Skeletor: A DSL for Describing Type-based Specifications of Parallel Skeletons

David Castro  Kevin Hammond
School of Computer Science, University of St Andrews, St Andrews, UK.
dc84@st-andrews.ac.uk, kh@cs.st-andrews.ac.uk

Abstract
Parallel computers are becoming increasingly common, and this trend will probably continue in the next years. This is a rapid change towards increasingly parallel hardware, which requires that software engineers adapt to the new situation. However, most of the common programming techniques are aimed at sequential or small scale parallel programming. Recent attempts to change this situation have appeared, providing tools for allowing an improved programmability of parallel systems. However, there are currently lacking tools for reasoning about parallelism. This paper describes a new approach that helps describing and checking interesting properties of parallel systems by using a type-based representation of their parallel structure. We believe that such a tool will ease the task of choosing a suitable and correct parallel structure for the implementation of parallel systems.

1. Introduction
The single-processor CPU with ever increasing clock speeds has become obsolete. Clock-speeds have stalled, and a new trend towards increasingly parallel multi-core/many-core hardware has appeared. However, software developing techniques have not caught up with this trend, and many software developers still use small-scale high-effort methods for parallel programming.

There are currently different attempts to change this state of affairs. Examples of this are pattern-based parallel programming models [17, 21, 28], the usage of algorithmic skeletons [4, 9, 15, 28], using refactoring techniques for parallel programming [6, 7, 12, 19, 30] or (semi-)automatic parallelization [14, 16, 27]. Some of the work related to this area focuses on creating an approach that helps the programmer to think parallel [17] so that they benefit from the massive amounts of parallelism which will soon be available. However, these approaches rely on properties and characteristics of parallel systems which are implicitly assumed and cannot be explicitly stated, checked or composed to describe more complex behaviours.

We believe that it is possible to overcome these limitations by pressing the point of think parallel and taking it a little bit further. This requires creating a way to reason about parallelism, i.e. properties of parallel programs and properties of transformations to introduce parallelism. This would allow software engineers to explore different possibilities/combinations for introducing parallelism with certain guarantees of correctness.

In this paper we present a new language-independent technique for achieving these objectives based on one key idea: using data types to capture parallel structure. A type system is basically a set of formal rules which assign a type to the various language constructs. Types are just a classification of these different elements according to their possible values. Type systems are well known tools that provide advantages such as error detecting, abstraction or serving as documentation [26]. We intend to take advantage of these characteristics for reasoning about parallelism. By creating a statically typed domain-specific language with expressive enough types for describing parallel structure, we can delegate the task of checking whether the properties we want to check hold or not. Moreover, it is possible to annotate the types with other kinds of useful information, such as cost estimates. The main contributions of this paper are:

1. we introduce a new way of representing parallel structure using a statically typed domain-specific language. The type system guarantees that the different components have the specified structure, and can be composed in a consistent way;
2. we show how to use a powerful type system to capture not only the parallel structure of a program, but also relevant properties of the domain;
3. we present a way to use this type representation to explore different parallel structures, and how the type system can be helpful to capture some kinds of errors and warnings in the specification of a parallel program. Moreover, we will show how to use this technique in a wider framework for specifying and implementing parallel systems.

2. Background
Parallel Patterns A parallel pattern [17, 21, 28] describes a common pattern of parallelism, which typifies a particular class of parallelism. They provide a design-time choice for parallelism, just as other kinds of software design patterns. Examples of parallel patterns are task farms, parallel maps, pipelines and parallel reduce operations.

Algorithmic skeletons An algorithmic skeleton [9, 17, 25] provides a parametric implementation of a specific parallel design pattern. It is usually implemented as a high-level function which re-
squeezes problem-specific code and any necessary skeletal parameters. Algorithmic skeletons can be nested.

2.1 Skeletons
These are some of the basic algorithmic skeletons:

**Seq** A wrapper for sequential computations to be computed as a unit. Given a function \( f : A \rightarrow B \) representing a piece of sequential code, we define the sequential skeleton as \( \text{Seq}(f) = f \) (Figure 1).

**Comp** This skeleton models the sequential composition of two computations represented by functions \( f_1 : A \rightarrow B \) and \( f_2 : B \rightarrow C \), denoted by \( f_1 \circ f_2 \) (Figure 2).

**Pipeline** The functional composition of two or more skeletons implementing a staged computation on a stream of independent inputs. It is denoted by \( (f_1 \| f_2 \| \ldots \| f_n)(x) \), where each stage \( S_i \) applies the function \( f_i \) to the output of the previous stage (Figure 3).

**Farm** A task farm (\( \Delta \)) models the application of a skeleton to a stream of independent inputs by a specific number of workers. The skeleton \( \Delta(nw, f, x) \) models the application of the function \( f \) to the input stream \( x_1, x_2, \ldots, x_n \) by \( nw \) number of workers (Figure 4).

**Map** A parallel map models those parallel computations in which a skeleton is applied in parallel to all the items building a collective data structure (e.g. the elements of a list). Given an input stream \( xS_1, xS_2, \ldots, xS_n \), where each \( xS_i \) is a collective data structure, \( xS_1 = \langle x_{11}, x_{12}, \ldots, x_{1k_1} \rangle \), the skeleton \( \text{Map}(f, p, c, x) \) applies for each input \( xS_i \), the function \( f \) to each partition \( \langle x_{i1}, \ldots, x_{ik} \rangle \) returned by \( p \). The result is combined by \( c \) back into a single data structure (Figure 5).

2.2 Skeleton Equivalences
There are a number of high-level structural equivalence relations for skeletons that can be rewritten from one form to another [1, 2, 6]. Here we show some of the well-known equivalences for the skeletons presented in this paper. From left to right these rules describe how to increase the parallelism degree, while reading them from right to left they describe how to decrease it.

1. **pipe intro/elim.** A sequential composition of \( n \) functions is structurally equivalent to a parallel pipeline with \( n \) stages.
   \[ S_1 \circ S_2 \equiv S_1 |\!| S_2 \]

2. **map fission/fusion.** A single parallel map over a sequential composition is equivalent to a composition where each stage is a parallel map.
   \[ \text{Map}(S_1 \circ S_2, p, c) \equiv \text{Map}(S_1, p, c') \circ \text{Map}(S_2, p', c) \]

3. **farm intro/elim.** A skeleton over a stream can be rewritten to a farm computation over the stream. It is important to realise that the order of the outputs might not be preserved by using a farm.
   \[ S \equiv \Delta(nw, S) \]

4. **data2stream.** A parallel map can be rewritten as a task farm, given that we provide two functions, one for partitioning the input into its different elements before the farm and other for combining them afterwards.
   \[ \text{Map}(S, p, c) \equiv \text{Partition}(p) |\!| \Delta(nw, S) |\!| \text{Combine}(c) \]

5. **map intro/elim.** A skeleton can be rewritten to a map given that the input for this skeleton can be partitioned and combined, and that the skeleton can be rewritten to work on the different parts of the map.
   \[ S_1 \equiv \text{Map}(S_1', p, c) \]

2.3 Type Systems

*Type systems* are a well known tool for assigning a property, a *type*, to the different language constructs by using a set of inference rules. They classify the different elements of the language according to the valid range of values they can take [26]. Types are present in most of the common programming languages, such as C, C++, Java, Haskell, OCaml, etc. An example of a simple type system can be seen in the C programming language. We can think of a C program (roughly) as a list of function definitions. Each function definition has a signature which specifies the type of the arguments received by the function, and the type of the output. Examples of such types in C are `int` or `float`.

Statically typed languages make use of this type system to check at compile time that there are no type errors, i.e. that no language constructs of a particular type are used where a different type is expected. For example, in the functional language Haskell [18], if we write the addition of an integer with a string the compiler would give a type error.

More complex type systems include dependent type systems. Examples of languages with dependent types include Agda [23], Epigram [22] or Idris [5]. A dependent type is a type which depends on a value. For example, in languages such as Idris we can write \( v : \text{Vect} \text{Int} 10 \), which means that the vector \( v \) has type \( \text{Vect} \text{Int} 10 \), or a vector of 10 integers. These type systems allow to express complex properties about the different elements of a program, e.g. we can express in the type of a function for sorting a list that the list is sorted.

The expressiveness of dependent types comes at the cost that the type inference is undecidable. Moreover, constructing well-typed expressions in a dependently typed language may be a hard task since the programmer might need to provide proofs for some properties specified in the types. A full description of the different type systems and dependent types is beyond the scope of this document.

3. Types for Parallel Skeletons

In this paper we describe a new small and expressive *domain-specific language*, Skeletor, aimed at reasoning about different parallel skeletons and the equivalences between them. We define a small set of constructs to describe different parallel structures, and provide a number of transformations between them inspired by their structural equivalences described in Section 2. A set of types and combinators are provided to allow the programmer to build their own abstractions about different patterns, experiment with different transformations to introduce new parallel skeletons and get estimates of the costs of using different parallel structures in different architectures. Moreover, we believe that this tool could be integrated in a wider framework (Figure 6) which would allow to provide as an input an already existing parallel algorithm. By providing a set of different desired transformations, the tool will use the type system to inform the user about incorrect transformations, i.e. transformations between non-equivalent parallel structures or
transformations that would break some required property, and give cost estimates about the valid outputs. One example of such properties would be that the order of the output of the skeletons involved in a transformation must be preserved.

In Figure 6 we give the overall structure of one possible framework using the tool described here. The main idea is that an interactive interpreter for Skeletor will be used to allow the specification of different parallel structures for a given input program, and explore all the different possibilities of parallelism. Then, by feeding the desirable transformations to a refactoring tool, it will be possible to generate different alternatives, using execution and profiling information to change the parallel specification or exploring other, more complex, rewritings.

3.1 The Skel Type

For the sake of simplicity, we will provide only a high-level description of the types instead of giving a full description of the implementation details. The basic type that we use is the type for parallel skeletons, \( \text{Skel} \). This type can be thought as an alias for a function transforming an input stream into an output stream.

\[
\text{Skel} \ a \ b = \text{Stream} \ a \to \text{Stream} \ b
\]

\( a \) and \( b \) are type variables, and \( \to \) is the type of a function. Intuitively, a Stream is a type representing a (possibly infinite) sequence of input/output tasks.

An example is a skeleton that might perform some filtering to a stream of input images, \( \text{Skel} \ \text{Img} \to \text{Img} \). This will be equivalent to a function from \( \text{Stream} \ \text{Img} \) to \( \text{Stream} \ \text{Img} \).

In Section 4 we describe possibilities for extending this type to deal with a wider range of situations.

3.2 Creating Skeletons

We provide a way to create these skeletons through a set of functions.

**Seq** A sequential skeleton applies the function \( f \) sequentially to a stream of inputs.

\[
\text{seq} : (f : a \to b) \to \text{Skel} \ a \ b
\]

**Comp** This is a sequential composition of two skeletons, \( s_1 \) and \( s_2 \). The first receives a stream of values of type \( a \) and returns a stream of elements of type \( b \), the second receives a stream of values of type \( b \) and returns a stream of values of type \( c \). The result is a skeleton receiving input streams of values of type \( a \) and returning streams of values of type \( c \).
Figure 6. Possible scenario for a framework using this tool: interactive framework.

comp : (s₁ : Skel a b) → (s₂ : Skel b c) → Skel a c

Pipe A pipeline is created by composing two skeletons, s₁ and s₂. Although in Section 2 we described a pipeline with many stages, for the sake of simplicity this function considers only pipelines with two stages. Given this basic construct, it is straightforward to deal with pipelines of more functions by adding more pipes as inner skeletons. Note that the types of comp and pipe are the same, since both skeletons compute the composition of their inner stages, comp sequentially and pipe in parallel.

pipe : (s₁ : Skel a b) → (s₂ : Skel b c) → Skel a c

Farm The function for creating a task farm will receive as inputs the number of workers (n), an emitter (e) which distributes the elements of the input stream to the set of workers, a collector (c), which merges the results of the different workers into a single output stream and a worker (w) on streams of elements of type a to b. This function will return a worker receiving a stream of elements of type a and returning a stream of elements of type b. It is important to note that the emitter and the collector are scheduling and collecting policies. This language provides some basic scheduling policies, such as a round robin or an on-demand strategy. For collecting policies, we provide two different policies: one collecting the results as they become available (non-ordering) and another enforcing in-order delivery of the results.

farm : (n : Nat) → (e : Emitter n a) → (c : Collector n b) → (w : Skel a b) → Skel a b

Note that we can use this function to create a default farm using a round robin scheduling and a non-ordering collection. Using the functions roundRobin as the emitter and the defColl as the default collector, this function can be defined as follows:

defaultFarm : (n : Nat) → (w : Skel a b) → Skel a b
defaultFarm n w = farm n roundRobin defColl w

Map Creating a parallel map requires as input a function (e) splitting each element of the input stream into the different parts, a function (c) for combining the outputs from the different set of chunks and a worker (w) on each partition.

map : (e : Partition a) → (c : Combine b) → (w : Skel a b) → Skel a b

3.3 Example
This example represents the parallel structure of an image processor. This program processes input images in parallel by using the default task farm, and, for each image, it applies a filter using a parallel map on the different partitions of this image. The functions partImg and combImg are the partitioning and combining functions for images respectively. The function filterImg is the sequential code for processing images.

For this example, we define two functions, procImg and filterImgs. procImg is a skeleton that applies a filter sequentially to a stream of input images. This filter is applied in parallel to each chunk of the image, which is later combined. We use this skeleton as a worker for a farm of 5 workers, which is defined in function filterImgs.

procImg : Skel Img Img
procImg = map partImg combImg (seq filterImg)

filterImgs : Skel Img Img
filterImgs = defaultFarm 5 procImg
4. Properties of Parallel Skeletons and Rewrite Rules

The types presented so far just allow the specification of parallel programs by using nested parallel skeletons. We need more information in the type to allow the description of different desirable properties of the parallel skeletons.

The main limitations of the current representation are twofold:

1. we cannot use the type system to check for structural equivalences using these types. If we define a function pipeIntro, we will have to check that the input is a Comp skeleton, and give a run-time error in case it is not. We need to add information about the parallel structure at the type level if we want to delegate this task to the type checker; and,

2. using the current types we cannot represent behavioural properties. For example, the task farm collector will merge the results of the different workers without necessarily preserving the order of the input list. That means that if we want to introduce a farm for a given skeleton, we have to check whether this property is required or not. For example, if we are describing a program performing some filtering to an input stream of video frame images, this property will be required.

4.1 Rewriting Skeletons using Types

In order to statically check that structural equivalences hold for our rewriting functions, we need to add a type annotation about the nested parallel structure of the skeleton. This type will only contain information about the parallel pattern, omitting the details: emitter function, collector function, partition, etc.

The type annotations that represent patterns are: Seq, Farm p, Map p, Pipe p1 p2, Comp p1 p2, where p1 and p2 are nested patterns. We modified the Skel type to include this information, so that when we create a sequential skeleton we give it the following type:

\[
\text{Skel} \quad \text{Seq} \quad \text{a} \quad \text{b}
\]

Likewise with a farm

\[
\text{Skel} \quad \text{(Farm p)} \quad \text{a} \quad \text{b}
\]
and with the rest of the skeletons. To illustrate this, consider the example given in Section 3. The type of procImg describes that it is a skeleton that corresponds to a parallel map with an inner sequential computation. The type of filterImg s describes a skeleton that corresponds to a task farm with an inner parallel map that applies a sequential computation.

\[
\text{procImg} \quad : \quad \text{Skel} \quad (\text{Map}\quad \text{Seq}) \quad \text{Img} \quad \text{Img} \\
\text{filterImg} \quad : \quad \text{Skel} \quad (\text{Farm}\quad (\text{Map}\quad \text{Seq})) \quad \text{Img} \quad \text{Img}
\]

Note that although these types seem too verbose, they can be easily inferred. This can be useful to allow the programmer to write only certain relevant type annotations, so that the types of the parallel skeletons become more readable. However, we believe that it is a good coding practice to provide type annotations, as they are a very valuable documentation.

4.1.1 Basic Rewriting Functions

We provide a set of type-safe functions that are used to introduce and eliminate parallel skeletons given some specific inputs. These functions are based on structural equivalences given in Section 2. We provide the alias Rw for the type of the rewrite functions. This type has four parameters, p1 and p2 are parallel annotations of skeletons, and a and b are the types of the values of the input and output streams. \(\text{Rw} \quad p1 \quad p2 \quad a \quad b\) is a function that receives a skeleton annotated with the parallel pattern p1 and returns a parallel skeleton annotated with p2.

\[
\text{Rw} \quad p1 \quad p2 \quad a \quad b \quad = \\
\text{Skel} \quad p1 \quad a \quad b \quad \rightarrow \quad \text{Skel} \quad p2 \quad a \quad b
\]

Note that these functions make use of the information contained within the types to check if a valid input is provided. For each structural equivalence rule we define an introduction function and an elimination function.

1. \text{Farm intro}. Farm introduction requires the same arguments as the basic combinator for creating a farm. That is, the number of workers, nw, the emitter, e, and the collector, c. The type of the output skeleton of the rewrite function is a Farm, where the original parallel structure p is used as the annotation for the inner parallel structure.

\[
\text{farmIntro} \quad : \quad (\text{nw} : \text{Nat}) \\
\rightarrow \quad (e : \text{Emitter}\quad \text{nw}\quad \text{a}) \\
\rightarrow \quad (c : \text{Collector}\quad \text{nw}\quad \text{a}) \\
\rightarrow \quad \text{Rw} \quad p \quad (\text{Farm}\quad p) \quad a \quad b
\]

2. \text{Farm elim}. Farm elimination requires a farm skeleton as input, and returns the inner skeleton.

\[
\text{farmElim} \quad : \quad \text{Rw} \quad (\text{Farm}\quad p) \quad a \quad b
\]

3. \text{Pipe intro}. This function rewrites an input skeleton of type \text{Comp}, i.e. a sequential composition into an output skeleton of type \text{Pipe} with the same inner parallel structure p.

\[
\text{pipeIntro} \quad : \quad \text{Rw} \quad (\text{Comp}\quad p) \quad (\text{Pipe}\quad p) \quad a \quad b
\]

4. \text{Pipe elim}. The \text{pipeElim} function works just as \text{pipeIntro}, but rewriting a skeleton of type \text{Pipe} into a \text{Comp}.

\[
\text{pipeElim} \quad : \quad \text{Rw} \quad (\text{Pipe}\quad p) \quad (\text{Comp}\quad p) \quad a \quad b
\]

5. \text{Map fusion}. This function will transform a skeleton annotated with type \text{Comp}, i.e. a sequential composition into a map skeleton of type \text{Map} with the same inner parallel structure p.

\[
\text{mapFusion} \quad : \quad \text{Rw} \quad (\text{Comp}\quad (\text{Map}\quad p1)) \quad (\text{Map}\quad p2)) \quad (\text{Map}\quad (\text{Comp}\quad p1\quad p2)) \quad a \quad b
\]

6. \text{Map fission}. This function splits a parallel map of a sequential composition of two skeletons to a sequential composition of two parallel maps. Two functions for composing the two newly created map skeletons are required. Specifically, p and c are the partition and combine functions required for passing the intermediate results of the maps.

\[
\text{mapFission} \quad : \quad (\text{p} : \text{Partition}\quad t) \rightarrow \quad (c : \text{Combine}\quad t) \\
\rightarrow \quad \text{Rw} \quad (\text{Map}\quad (\text{Comp}\quad p1\quad p2)) \\
(\text{Map}\quad (\text{Comp}\quad p1\quad (\text{Map}\quad p2))) \quad a \quad b
\]

By describing the structural equivalence rules in the types of the rewriting functions, it is impossible to apply them to skeletons with the wrong structure. We illustrate this by showing a simple example where the type system enforces these structural equivalences. Let \(s\) : \text{Skel} \quad (\text{Farm}\quad \text{Seq}) \quad \text{Img} \quad \text{Img} be a task farm processing images. Suppose we want to modify the structure using \text{pipeIntro} s.
As pipeIntro requires a Comp as the type, we will get a type error informing us that it cannot unify Comp p₁ p₂ with Farm Seq. The type system ensures that the skeletons described this way are consistent.

4.1.2 Composing Rewritings

These functions can be composed to create more complex transformations. For example, a straightforward composition of those functions would split a map of a sequential composition into two maps that are combined using a pipeline of two stages. To do so, we use the . operator, which is a normal function composition: \((f₁ \cdot f₂) x = f₁(f₂ x)\). We define the function mapFP by a normal function composition of pipeIntro and mapFission:

\[
\text{mapFP} : (p: \text{Partition } a) \\
\rightarrow (p: \text{Combine } a) \\
\rightarrow \text{Rw (Map (Comp p₁ p₂))} \\
\rightarrow \text{Pipe (Map p₁) (Map p₂)} \Rightarrow a \Rightarrow b \\
\text{mapFP} p c = \text{pipeIntro . (mapFission p c)}
\]

Additionally, we provide functions performing traversals on the different structures. They can be used for describing more complex rewritings by applying other rewritings to an inner skeleton. They are useful for modifying the structure of the inner workings of the parallel skeletons.

For example, we provide the functions rwCompFst and rwCompSnd to apply a rewriting to the skeletons that correspond to the first and second stages of a sequential composition.

\[
\text{rwCompFst} : \text{Rw p₁ p₂ a b} \\
\rightarrow \text{Rw (Comp p₁ p) (Comp p₂ p) a c} \\
\text{rwCompSnd} : \text{Rw p₁ p₂ b c} \\
\rightarrow \text{Rw (Comp p p₁) (Comp p p₂) a c}
\]

Likewise, we provide the functions rwPipeFst, rwPipeSnd, rwFarm and rwMap for traversing the components of pipelines, farms and parallel maps.

4.2 Properties of Parallel Skeletons

There are two main questions which need to be answered in order to describe how we would check that different kinds of properties about the parallel skeletons hold. The first is which kinds of properties we are interested in, and the second is how do we add them to our language for describing parallel skeletons. The answer to those questions is still not clear, and we are still experimenting with different prototypes. In this section we will just give a very high-level description of an example of property we want to check and discuss a possible way to add this information to our types.

We give two examples of such properties. The first one states that given a worker skeleton, \(S\), for each input element, \(x_i\), an output element, \(y_i = S x_i\), such that size\((x_i)\) = size\((y_i)\) is produced. We can call this property size preserving (Size). The only thing that is checked is that the size of the elements of the input stream is the same as the size of the elements of the output of applying the skeleton to the input. Specifically, we want to define a function Size, which receives an input skeleton, \(S\), and returns a proof that for all input stream, \(s\), for all element \(x\) of \(s\), the size of the input is the same as the size of the output of applying the skeleton to this input. We use \(\ast:\ast\ast\) as an operator describing the application of a skeleton to one single element.

\[\text{Size: (f : Skel a b)} \rightarrow \forall \ i : \text{Stream a}, \forall \ x \in i, \text{size } x = \text{size } (f \ast : x)\]

Another property is related to the ordering of the output elements. Given a worker skeleton, \(S\), and an input stream, \(x_1, x_2, \ldots, x_n\), if the output is a stream \(y_1, y_2, \ldots, y_n\) such that \(y_i = S x_i\), we say that it preserves the ordering (Ord). A task farm does not necessarily have this property. We can state this property as follows: the sequential execution of the inner worker on each element of the input stream, \(s'\) returns the same output as the execution of the parallel skeleton on the same input. Again, we want to define a property receiving a skeleton, \(S\), returning the described proof.

\[\text{Ord: (f:Skel a b)} \rightarrow \forall \ i : \text{Stream a}, (f' \ i = f \ i)\]

Note that this is still a work in progress and the exact details of how we represent this may change. We intend to use these properties by adding them as constraints to the functions returning skeletons so that later they can be checked for inconsistencies and errors, but we are considering other alternatives.

4.3 Annotating Types with Cost Information

We want to annotate the skeletons with cost information obtained by applying cost models. Given the type-level description of the parallel structure of our programs, we can annotate the skeletons with an estimate obtained by applying cost models [6, 8, 25] for parallelism. As an example, we show the cost model of a task farm:

\[
T_{\text{farm}}(N_w, L) = \max \{ T_{\text{emitter}}(N_p, N_w, L), T_{\text{collector}}(N_w, L), \left( T_{\text{farm}}(L) \right)_{\min(N_p, N_w)} \}
\]

The cost model of a farm receives as parameters the number of workers, \(N_w\), and the maximum size of the input tasks, \(L\). By using these parameters, it estimates that the farm run-time is the maximum of the costs of the emitter, \(T_{\text{emitter}}\), the collector, \(T_{\text{collector}}\) and the time it takes to compute the entire sequential computation, \(T_{\text{farm}}\) divided by the minimum of the number of workers, \(N_w\), and the number of processors, \(N_p\). This is assuming that each worker has a similar granularity and that all workers are fully occupied.

We intend to annotate the skeletons with this information using their types. This information can be very useful for informing the user when a transformation might give worse run-time behaviour than the original. We could be tempted to add as a constraint for applying the different skeletons that some benefit in terms of run-time is obtained. However, as this information will only give an estimate which might not always be accurate, this would be not a reasonable approach. Also, applying successive transformations might lead to a better performance, even if intermediate transformations produce skeletons with worse performance, posing a limitation to the composability of this DSL.

Since some parameters of the cost models depend on the available physical resources or the implementation details of the skeletons, we will not calculate run-time predictions for a given parallel structure. Instead, we will associate each skeleton with a function which, when applied to specific architecture details and profiling information, will return a run-time estimate.

As an example, we will explain how the farm skeleton can be annotated its cost model. The first type we need is the type of the cost models, \(CM\), which is a function that receives a description of the architecture, \(\text{Arch}\), and returns an integer which represents the predicted run-time in milliseconds.
5. Case Study

In this section we illustrate the basic usage of the types and the skeleton DSL on a simple example of denoising a stream of satellite images. This is an adaptation of a case study described in [6]. The basic program comprises a two-stage function composition. The first stage, geoRef, applies an algorithm to each image received to consolidate georeferencing information. The second stage, filter, is in charge of denoising the images. So we will start by explaining how we represent the basic structure of the program. It is basically a skeleton representing a sequential composition of two stages. The second stage can be split into a map over the images.

\[
\text{geoRefSkel} : \text{Skel Seq Img Img} \\
\text{filterSkel} : \text{Skel (Map Seq) Img Img} \\
\text{denoise} : \text{Skel (Comp Seq (Map Seq)) Img Img} \\
\]

Given these types, we can start applying the pipeIntro function on the denoise function. Since we have a Comp skeleton, it is possible to do that.

\[
\text{pipeIntro (denoise)} \\
\]

We can also apply a farm into the first stage of the pipe. This is done by using traversal functions, in this case rwPipeFst. Note also that we use the function defFarmIntro to avoid passing the arguments for the emitter and the collector; the default policies will be used. Also, the number of workers, \( nw \), is specified elsewhere. In this example, we apply a rewrite function to the first stage of a Pipe created by applying pipeIntro to the original denoise function.

\[
\text{pipeDenoise2} : \text{Skel (Pipe (Farm Seq) (Map Seq)) Img Img} \\
\text{pipeDenoise2} = \text{rwPipeFst (defFarmIntro nw)} (\text{pipeIntro (denoise)}) \\
\]

Another possibility within this language is to compose these rewriting functions to create new ones. This can be done by passing the skeleton as an argument of a function which combines the different rewriting functions.

\[
\text{pipeFarm} : \text{Rw (Comp Seq (Map Seq)) (Pipe (Farm Seq) (Map Seq)) a b} \\
\text{pipeFarm} = \text{rwPipeFst (defFarmIntro nw)} . \text{pipeIntro} \\
\]

The functions farmPipe and pipeFarm are prototypes of different ways of rewriting a skeleton to produce the same kind of parallel structure, and this is shown by their types. Note that if we had used a wrong rewriting or parallel skeleton, the type checker would provide a type error with enough information to help locate it. For example, consider the definition of pipeFarm. If we do not apply pipeIntro, we will get the type error “Cannot unify Pipe (Farm Seq) (Map Seq) with Comp (Farm Seq) (Map Seq)”, as well as the line of code where this error was detected.

Note that these types do not provide enough information for reasoning about these different parallel structures. If we want to allow this, the next step is to add the cost models to the DSL. By doing this, the tool will allow to reason about desirable parallel structures for different architectures, providing a means to make informed decisions about parallelism.

6. Related Work

The ParaPhrase project. The paraphrase project is a novel approach which aims to introduce parallelism through the usage of advanced refactoring techniques to introduce high-level parallel design patterns. The main idea behind this project is to use these high-level design patterns to restructure programs into other forms which are more suited for parallel execution [17]. A key goal of this project is that the different parallel components are intended to match heterogeneous architectures, e.g., combinations of CPU/GPU.
By using this high-level approach they provide a framework which eases the design and implementation of different parallel systems by allowing the programmer to focus on the high-level details. Although our approach draws many of the key concepts and ideas from this project, we focus on providing a tool for specifying and exploring different parallel patterns on a language-independent tool. Moreover, it is our belief that this tool could complement the framework and methodology of the ParaPhrase project by allowing the description and specification of more complex patterns and rewrite rules, composed in terms of the simpler constructs which we provide and using the information described in the types for guiding all the process.

**Algorithmic Skeletons.** Algorithmic skeletons abstract common patterns of parallel computation. They present a top-down structured approach where parallel programs are described using nested skeletons. This is known as structured parallelism [9, 15]. This form of structured parallelism provides a clear and consistent structure across platforms by distinctly decoupling the application from the structure in a parallel program. Although algorithmic skeletons cannot be used to describe all parallel and distributed programs, there is a growing number of applications [10, 28]. Furthermore, algorithmic skeletons possess a predictable communication and computation structure, since they capture the structure of the program, that provides a foundation for performance modelling and estimation of parallel applications. Our approach highly depends on algorithmic skeletons. In fact, we believe that our approach can provide the advantages of both algorithmic skeletons and type systems. By relying on the type system, this tool can provide a way to describe and transform the parallel structure of a program, allowing also to reason about the performance of different parallel structures for the same problem in a consistent way.

**Semantics of skeletons.** Several works provide a formal description of the semantics of the skeletons. Aldinucci and Danelutto [3] propose an operational semantics schema that can be used to describe both functional and parallel behaviour of skeletal programs in a uniform way. They show how the semantics enables several interesting analysis of Lithium [11] programs, such as the description of functional semantics, the comparison in performance and resource usage between functionally equivalent programs and the analysis of maximum parallelism achievable with infinite or finite resources. Falcou and Séro [13] show how generative and metaprogramming techniques can be applied to the implementation of a skeleton-based parallel programming library. Quaff. This implementation is derived directly from a set of explicit production rules, in a st-parallel-orientated style. Our approach highly depends on algorithmic skeletons. In fact, we believe that our approach can provide the advantages of both algorithmic skeletons and type systems. By relying on the type system, this tool can provide a way to describe and transform the parallel structure of a program, allowing also to reason about the performance of different parallel structures for the same problem in a consistent way.

**Rewriting Rules.** Roughly, rewriting systems consist of a set of objects with some relations on how to transform those objects. Rewriting rules for transforming different parallel skeletons into other kinds of parallelism have been used for the implementation of different refactoring techniques [1, 2, 6]. Our approach is related to this, as we try to provide the different rewriting rules as functions with expressive types that capture the relations between the different objects, so that no wrong rewriting can be applied.

**Formalisation of Refactorings.** Opdyke [24] proposed set of seven invariants to preserve behaviour for object-oriented programming language refactorings. Opdyke’s refactorings were accompanied by proofs demonstrating that the enabling conditions he identified for each refactoring preserved the invariants. However, Opdyke did not prove that preserving these invariants preserves program behaviour. Tip et al. [29] used type constraints to verify the preconditions and to determine the allowable source code modifications of generalisation-related refactorings in an OO programming language context. Li and Thompson [20] investigate the formal specification of refactorings as well as proving their functionality preservation. Although they claim that refactorings should preserve behaviour, by applying refactoring to introduce parallelism we may change the behaviour and still have a correct program. For example, for certain kinds of problems where a list of output values is produced we might not be interested in preserving the ordering of the output. What we need is to identify key properties, and guarantee is that they still hold after the refactoring.

7. Conclusions and Future Work

This paper has described a new design for a DSL, Skeletor, for specifying parallel systems in terms of their parallel structure. The novelty of this DSL is that it relies on a type-based representation of the parallel skeletons to expose the parallel structure at type level. We also described how to perform different transformations between parallel structures that ensure that the arguments and the outputs have the required structural equivalence in a consistent way. We described also how the Skeletor DSL could be used in a wider framework, where the specification of the parallel programs could be extracted from an already existing implementation in different programming languages. Our main goal with this work is to provide a means of reasoning about different parallel structures and equivalences between them, so that we can use the information obtained by using this tool for a better decision about which parallel structure to choose for certain problems. We expect to achieve this objective by adding cost models to the DSL.

Since the Skeletor DSL is a new prototype, there is still much room for improvement. We need to identify more of the key properties of the parallel skeletons. A challenging future task is to find a better representation for these properties, so that they have certain desirable characteristics. For example: different properties have to be easily composed when creating different skeletons, it must be easy to reason and work with them and they should not overcomplicate the types. Since these properties are implicitly derived from the semantics of the skeletons, another possibility is to formally describe this semantics within the DSL to provide a sound means to prove them. We intend also to extend this approach with a wider range of parallel skeletons, and explore the possibility of composing and creating more complex skeletons and rewrite rules for them. Additionally, we intend to find representative case studies to show the advantages of using this tool, identify its limitations and find a solution for them.

The description of the framework in which the Skeletor DSL can be integrated is still vague. We need to provide a precise description of each of the steps and components of this framework. One of the possibilities is to draw inspiration from the methodology of the ParaPhrase project, and possibly describe an adaptation of it that includes the usage of a toolchain for the Skeletor DSL. Finally, we need to build this toolchain to fulfill the requirements of the framework. Since this toolchain heavily depends on the framework in which we intend to use this DSL, we still cannot provide more details about it. However, we speculate that integrating type-based skeletons into a general-purpose, mainstream, language can be an interesting possibility.

8

2013/12/9
References


Measuring and modelling energy consumption for a CPU/GPU conjugate gradient method in an adaptive FEM

Jens Lang  Gudula Rünger
Department of Computer Science, Chemnitz University of Technology, 09107 Chemnitz, Germany
jens.lang@cs.tu-chemnitz.de  gudula.ruenger@cs.tu-chemnitz.de

Abstract
In scientific computing, energy models for algorithms or specific implementations support energy-aware applications, enable ensuring power limits or generally help improving the energy efficiency. In this article, an energy model for the conjugate gradient method (CGM) used in an adaptive finite element method (FEM) is developed. Furthermore, the article investigates software-based methods for measuring the energy consumption of the CGM when executed on both, the CPU and the GPU. The influence of the CPU performance state, i.e. the frequency, on the computational power and on the energy consumption is shown. The case study draws conclusions for an energy-efficient execution of the application and an energy-aware workload distribution between CPU and GPU.

Keywords conjugate gradient method, energy consumption, energy awareness, frequency scaling, GPU

1. Introduction
Minimising energy consumption has been a minor design goal for applications in scientific computing, for a long time. However, for highly parallel high-performance applications, energy consumption will gain a significance similar to the execution time, for example for achieving Exascale computing.

The scientific code considered in this article is an adaptive finite element method (FEM) [6] which, e.g., is applied to deformation problems. The adaptive FEM refines its mesh adaptively at the most critical points in contiguous iterations, i.e. adds more elements to the mesh at the points with the largest deformation gradients. The most time-consuming step in the FEM implementation is the solution of a system of linear equations of the stiffness matrices, which is solved by a conjugate gradient method (CGM).

In [17], we have investigated the execution time of the CGM in the FEM and determined an optimal distribution of the computational workload between CPU and GPU with respect to time. In contrast, the present work investigates the energy consumption of the CGM on the most common kinds of processing units, i.e. CPU and GPU. Energy measurements as well as an evaluation on how to perform the most energy-efficient execution are given. The contribution of the article is to present detailed experiments covering different aspects of the energy consumption of the CGM. The experiments include a direct comparison of the energy consumption on the CPU and on the GPU, the influence of the CPU frequency, and the energy needed for transferring data between GPU and main memory. Furthermore, an energy model for the CGM is developed.

Energy models for applications are needed if the machine which executes an application has a certain power limit which must not be exceeded, or might help to decide on which hardware an application should be executed to achieve the best energy efficiency. The last aspect could include different CPU models, CPU-GPU hybrids or accelerators.

A general definition of energy efficiency is the ratio of “useful output of a process” to the “energy input into a process” [24]. In computing, such a metric can be “records sorted per joule” [26] or “algorithmic steps per joule” [13]. In this article, the reciprocal value $\varepsilon$, “energy needed for processing one ‘item’ of computational work”, is taken, which is, e.g., also used in [30]:

$$\varepsilon(E,n_{item}) = \frac{E}{n_{item}}$$

where $n_{item}$ denotes the number of “work items” processed and $E$ denotes the total energy needed by the processing units during the execution. Each work item comprises a similar amount of computation. In this article, one work item is defined to be the processing one finite element of the FEM, unless stated otherwise. The lower the value $\varepsilon$ is, the more energy-efficient is the implementation.

The rest of this article is structured as follows: Section 2 gives an overview on the conjugate gradient method in its element-wise version. Section 3 describes the methods used for measuring the energy consumption of the CPU and the GPU. In Sect. 4, the experiments are presented. In Sect. 5, an energy model is developed, and Sect. 6 discusses the results and their implications. Sect. 7 concludes the article.

2. Element-wise conjugate gradient method
One component of the FEM considered is a parallel solver for linear systems of equations using the conjugate gradient method [20]. This method solves the linear system of equations

$$Au = b \quad (1)$$

by testing for a potential solution $u^k$ if $A u^k \approx b$ holds. More precisely, it is tested whether the residuum $r^k = |A u^k - b|$ is below a given error bound $\varepsilon$, i.e. $r^k < \varepsilon$. If the condition is not fulfilled, the next solution $u^{k+1}$ is computed and tested in a further iteration. The first potential solution $u^1$ is chosen arbitrarily. Using a preconditioner for choosing $u^{k+1}$ accelerates the convergence and hence reduces the number of iterations needed.

However, the present implementation does not use the overall vectors $u$ and $y$ and the overall matrix $A$, but performs an element-
wise matrix-vector multiplication of the form
\[ y_{el} = A_{el} u_{el} \]  
(2)
for all finite elements \( el \) of the FEM. While the size of the overall vectors grows proportionally to the number of elements and may grow to some hundreds of thousands, the size of the element vectors remains constant. Depending on the number of nodes per element and the number of degrees of freedom, it is between 8 and 81. The data needed for processing one element, i.e. the element data structures \( A_{el}, u_{el} \) and \( y_{el} \), is extracted from the overall data structures and converted back by dedicated functions \( o2el \) and \( e2o \):
\[ A_{el} = o2el(A,el), \quad u_{el} = o2el(u,el), \quad y_{el} = \sum_{el} e2o(y_{el}) \].
(3)

The routine performing the CGM calls the sub-routine AXMEBE to perform the element-wise matrix-vector multiplication according to eq. (2) for all elements \( el \). AXMEBE is shown in pseudo code in Alg. 1.

Since the equations (2) to (4) are independent from each other for each element, AXMEBE can be executed in parallel, as shown in Lines 3 to 9. In Lines 4 and 5, the element data structures are extracted from the overall data structures according to Eq. (3). The matrix-vector multiplication of Eq. (2) is performed in Line 6, and the element data structures are converted back according to Eq. (4) in Line 7. The writing into the overall vector \( y \), which is shared between the processors, is performed in Line 9. The synchronisation of the write accesses is ensured by the critical section.

Algorithm 1: Pseudo code of AXMEBE

```
begin
  for each el do in parallel
    A_{el} := o2el(A,el)
    u_{el} := o2el(u,el)
    y_{el} := AXMEBE(A_{el},u_{el})
  begin synchronised
    y := y + y_{el}
  return y
```

The parallel section in Lines 3 to 9 is suitable for being executed on both, CPU or GPU. For an execution on a GPU, the required data from \( A \) and \( u \) has to be transferred to the GPU memory beforehand and the result \( y \) has to be fetched from the GPU memory afterwards. For a detailed description of the implementation, see [5] and [17].

The code used for this work is based on a Fortran code [6]. The main parts of the code are parallelised using a shared-memory parallelisation with OpenMP [5]. For the CGM, there is additionally a GPU implementation in CUDA [17]. The method presented in [17] minimises the execution time by utilising both, CPU and GPU, for executing the workload of the FEM concurrently: The execution time model allows to predict the execution times on both kinds of processing units enabling to distribute the workload in a way that the execution is finished on both processing units at the same time. Using this method, roughly half of the elements is processed on the CPU cores and the other half of the GPU is processed with the hardware described in the next section.

3. Measuring method

This section gives details on how the energy consumption measurement of the CPU and of the GPU for executing the routine AXMEBE was measured. All measurement is performed using software interfaces, no additional measurement instruments have been added.

The hardware used is a 16-core machine, consisting of 2 octocore CPUs Intel Xeon E5-2650 with a nominal clock rate of 2.0 GHz. The machine has 32 GiB of RAM, which is composed of 8 GiB Samsung M393B1K70DH0 DIMMs with a clock rate of 1.6 GHz and distributed on 2 NUMA nodes. Additionally, the machine contains an Nvidia Tesla C2075 GPU. On the software side, the machine runs a Linux operating system with kernel version 3.2.46, the Nvidia GPU driver version 304.88 and the CUDA 5.0 environment.

3.1 CPU

With the CPU architecture code-named Sandy Bridge, Intel introduced machine-specific registers (MSRs) which are intended for “Running Average Power Limiting” (RAPL) [7]. Among these RAPL registers are “energy status” registers providing an energy metering interface [15, vol. 3B, ch. 14-28]. This functionality is used to measure the energy consumption of the CPU in the present work. On Sandy-Bridge server CPUs, there are three different energy metering registers which measure the following values:

- **MSR_PKG_ENERGY_STATUS** for the energy consumption of the CPU package (i.e. one socket),
- **MSR_PPD_ENERGY_STATUS** for the energy consumption of processor cores including their caches [25], except the last-level cache [10],
- **MSR_DRAM_ENERGY_STATUS** for the energy consumption of the memory modules (DIMMs) of this package [28].

Each register is updated roughly every 1 ms and its value contains “the total amount of energy consumed” since the last time the register was cleared [15, vol. 3B, ch. 14-28]. In this work, the register MSR_PKG_ENERGY_STATUS is used as it hints at the power consumption of everything needed for computation on the CPU, i.e. including caches, un-core energy, etc. RAPL is used for measuring the energy consumption of algorithms for example in [11, 14, 25, 31].

The MSRs are read via the kernel module msr, which enables access to MSRs of the CPU in the user space. The interface is provided by the file /dev/cpu/k/msr, where \( k \) is the number of the CPU core. The executable reading the interface file is required to have the capability CAP_SYS_RAWIO. The capability has to be set by the root user after each compilation of the source code, which is very uncomfortable for everyday development. So for this work, the msr kernel module has been altered not to check the capabilities.

For the measurement, the registers are read immediately before the call and after the return of AXMEBE in an adapted version of the real FEM application. In this way, all effects that occur in “real life” execution, e.g. cache misses, are included.

3.2 GPU

For measuring the power consumption of the GPU, its integrated power meter is used. The principal purpose of the power meter is to ensure that the GPU does not consume more power than 225 W as specified in the PCIe standard. The current value of the power can be accessed via the function nvmiDeviceGetPowerUsage of the Nvidia Management Library (NVML) [23]. The value comprises the power consumption of the whole GPU being drawn from both, the PCIe socket and the additional power supply.

As the on-board power meter updates its value only every 20 ms, the method presented in [18] is used to obtain a power profile of the GPU routines with a higher temporal resolution. This method executes the routine to be evaluated a large number of times (50 to 100), always with the same parameters. The function is called at random phases of the sensor update interval so that the power consumption is measured at different times during the execution. Integrating the values obtained yields the energy consumption of the routine on the GPU.
Due to linear [29] or even super-linear [32] dependency of the power from the temperature of the circuitry, it is necessary to ensure that the temperature of the GPU is always in the same range, which is 54 °C for this work. If the temperature is lower, then the GPU is heated by executing some workload. If it is higher, the GPU is left idle for some moments in order to give it the possibility to cool down. For measuring the temperature, the internal thermometer accessible via the function `nvmlDeviceGetTemperature` from the NVML is used. A temperature of 54 °C results in a static power consumption of 77...79 W for the GPU in short-idle mode.

4. Experiments

This section describes the elements conducted. The results build the foundation for the energy model developed in Sect. 5. The test cases `c8Zyl` and `bohrung` from [6] have been used for the experiments. Each finite element consists of 27 nodes, each having 3 degrees of freedom, which results in an element data structure size of 81.

4.1 CPU-only and GPU-only execution

At first, the energy consumption of `AXMEBE` is investigated depending on its execution time. Figure 1 shows the results of the experiment. Each point represents one call of `AXMEBE`. The experiment has been conducted several times for each number of elements. As the machine contains two CPUs, the values obtained for the MSRs `MSR_PKG_ENERGY_STATUS` for both CPUs have been summed up to the total value. The regression lines for the measurement results have been obtained using the Levenberg-Marquardt least-squares method. They show that the energy consumption $E$ is roughly proportional to the execution time $t_{ex}$:

$$E \sim t_{ex} \quad (5)$$

The power values have been obtained as follows: $P_{CPU} = 126$ W for the CPUs and $P_{GPU} = 138$ W for the GPU. The values correspond well to the thermal design powers of 95 W per CPU package [11] and 215 W for the GPU [22]: Both kinds of processing units need roughly two thirds of their maximum power.

As [17] showed, the execution time $t_{ex}$ is proportional to the number of elements $n_{el}$ processed:

$$n_{el} \sim t_{ex} \quad (6)$$

From the relations in eq. (5) and in eq. (6), it can be concluded that

$$E \sim n_{el} \quad (7)$$

holds also true, i.e. the amount of energy consumed is proportional to the number of elements processed.

4.2 Frequency scaling

In order to reduce their power consumption, modern CPUs can be set into an operational mode with lower clock frequency and lower voltage. For each mode, a P-state with a distinct frequency is defined. This technique is often referred to as dynamic frequency and voltage scaling (DVFS) [8]. In this section, it will be investigated whether setting a specific P-state for the operation of the CPU allows to execute the CGM in a more energy-efficient way.

In Linux, the kernel module `cpufreq` is responsible for setting the P-state considering the current CPU load. Different governors are available for this task, e.g. `performance`, which always chooses the highest frequency possible, or `powersave`, which always chooses the lowest frequency possible. The governor `userspace` enables the P-state to be set from the user space by writing the desired clock frequency to the file `scaling_setspeed` in the directory `/sys/devices/system/cpu/cpu0/cpufreq/`. The Intel Xeon E5-2650 used for the experiments in this work offers P-states with the following clock frequencies: 1.2 GHz to 2.0 GHz in intervals of 0.1 GHz, and 2.001 GHz, which is the Turbo Boost mode [4]. Turbo Boost runs the CPU cores with a higher than the nominal clock rate as long as certain conditions are met, including the current power consumption and the processor temperature do not exceed the manufacturer-specified values [2]. Thus, 2.001 GHz does not represent the actual frequency which is determined internally.

The experiments have been conducted by setting all CPU cores to each of the P-states available, executing `AXMEBE` and measuring execution time and energy consumption for different numbers of elements. The results are shown in Fig. 2. The chart shows the behaviour of the power consumption $P$ and the execution speed $v_{ex}$ depending on the CPU clock frequency $f$. The execution speed defines how many elements $n_{el}$ can be processed in a given time $t$, i.e.

$$v_{ex} = \frac{n_{el}}{t} \quad (8)$$

The two vertical axes have been scaled so that the power and the execution speed are drawn on top of each other for the last P-state. The measurement results have been approximated by the regression curves

$$P(f) = af^3 + bf + c \quad \text{and} \quad v_{ex}(f) = df + e \quad (9)$$

![Figure 1. Energy consumption of the CPU and the GPU for executing AXMEBE depending on its execution time. The CPU uses the cpufreq governors performance (clock rate 2.0 GHz) and powersave (clock rate 1.2 GHz).](image1.png)

![Figure 2. CPU power consumption and execution speed for performing the CGM with different CPU clock frequencies](image2.png)
in the range of 1.2...2.0 GHz. The Turbo Boost mode with 2.001 GHz has not been considered for the regression analysis as its actual frequency does not correspond to the clock frequency assigned to the P-state.

That the power consumption is best represented by the cubic equation in (9) is motivated by the equation

\[ P = CV^2 f + V \cdot I_{\text{leak}} \tag{10} \]

which is generally used for modelling the power consumption of CMOS circuits [21]. In this equation, \( V \) represents the voltage, \( C \) the capacitance of the transistors and wires, \( f \) the frequency and \( I_{\text{leak}} \) the leakage current. The frequency \( f \) depends linearly on the voltage \( V \) [33], which is also indicated by [3] for the Pentium M processor. Therefore, the first summand in eq. (10) is proportional to \( f^3 \) and the second is proportional to \( f \). For reference, the values of the parameters are: \( a = 2.2 \pm 0.4 \), \( b = 29 \pm 4 \) and \( c = 24 \pm 4 \).

The linear relation of execution speed and clock frequency results from the compute-boundness of the AXMEBE code. The number of instructions in the routine is fixed and the duration of their execution is a fixed number of clock cycles, hence the execution time of AXMEBE is roughly proportional to \( f^{-1} \). The parameters found by the regression are \( d = 222 \pm 4 \) and \( e = -70 \pm 6 \). The constant offset might result from code sections needing the same fraction of execution time as 70 elements per millisecond, but whose execution times are invariant with respect to the clock frequency. Such code sections might be memory-accesses in AXMEBE or interrupt service routines triggered from the exterior. Also systematic errors in the experimental setup cannot be ruled out.

The charts in Fig. 2 clearly indicate that the execution speed is increasing faster than the power consumption and hence the P-states with higher frequencies allow a more energy-efficient operation.

### 4.3 Per-element energy

For an energy-efficient execution of the CGM, the questions to be answered are on which processing unit, CPU or GPU, the energy needed to process one element, the per-element energy, is smaller, and whether this energy varies with the CPU clock frequency. Figure 3 shows the per-element energy of the CPU and the GPU. The top chart shows its dependency from the CPU clock speed. The results have been obtained by dividing the power values \( P \) by the execution speed values \( v_{\text{ex}} \) from Fig. 2:

\[ E_{\text{el,cpu}}(f) = \frac{P(f)}{v_{\text{ex}}(f)} \tag{11} \]

The regression curve shown in this chart is also the ratio of the regression curves in Fig. 2. The results show that higher clock frequencies allow a more energy-efficient operation: With a frequency of 2 GHz, the per-element energy \( E_{\text{el,cpu}} \) is 267 µJ, compared to 321 µJ with a frequency of 1.2 GHz. In the Turbo Boost mode, the energy is 265 µJ.

The per-element energy for the GPU \( E_{\text{el,gpu}} \) is shown in the bottom chart in Fig. 3. Each point results from one experiment with a specific number of elements. A clear dependency from the total number of elements is not visible. \( E_{\text{el,gpu}} \) ranges from 220 µJ to 250 µJ. The mean of the values is 233 µJ. The results indicate that the GPU processes the workload of the CGM in a slightly more energy-efficient way than the CPU.

### 4.4 Theoretic calculation of the per-element energy

The main operation in the AXMEBE routine, taking roughly 90 % of its execution time on the CPU, is the matrix-vector multiplication routine \textsc{smamvek}. Its semantics is similar to the \textsc{spmv} BLAS routine: A symmetric matrix, stored as a triangular matrix, is multiplied by a vector. The pseudo code, which has been derived from the source code [6], is shown in Alg. 2. The size of \( n \) is never above 81 as this is the maximum size of the element data structures. Experiments have shown that this implementation is as time-efficient as a BLAS implementation for the matrix and vector sizes used.

The following floating-point operations are performed in the code: The outer loop iterates \( n \) times and has one addition and one multiplication per iteration. The inner loop has \( n(n+1)/2 \) iterations each consisting of two additions and two multiplications. This yields \( 2n(n+2) \) floating-point operations in total.

Tolentino and Cameron [30] estimate the energy consumption of operations on current machines as \( 2 \ldots 10 \text{ nJ} \) per floating-point operation. With a value of \( n = 81 \), an upper bound for the energy consumption for computation is 134 µJ per element. This value
seems to be realistic compared to 265 µJ for the whole AXMEBE, if one bears in mind that the following aspects have not been taken into consideration:

- Data has to be read from memory and stored back.
- Code has to be read and decoded.
- Per SMAMVEK call, there are \( n(n + 3) \) arithmetic integer operations and further operations controlling the programme flow, such as compare or jump operations.
- There are further operations to be performed by AXMEBE which contribute roughly 10% to its total execution time.

4.5 Throttling down one CPU

As the GPU allows a more energy-efficient execution of AXMEBE than the CPU, it might be considered processing all elements on the GPU and meanwhile throttling down the CPUs. Experiments showed that the throttling down does not work to the full extend: The first CPU package, CPU 0, does always draw the full amount of power, even when it is only waiting for the GPU to finish. Therefore, it is reasonable that this CPU performs some computation during the waiting period. In contrast, the second package, CPU 1, is throttled down automatically if there is no thread running on this CPU. Running the 8 threads on the cores belonging to CPU 0 and no thread on CPU 1 can be ensured with OpenMP by setting the environment variable `OMP_NUM_THREADS` to 8 and `GOMP_CPU_ Affinity` to 0-7.

The top chart in Fig. 4 shows the energy consumption of the CPUs when executing 16 threads: Both CPUs are working under full load and have a similar power consumption: \( P_{CPU0} \approx P_{CPU1} \approx 63 \) W. The bottom chart shows that, in contrast, CPU 1 needs much less energy when only 8 threads, which are all assigned to CPU 0, are running in total. Then, CPU 1 can power down unused components which leads to a power of \( P_{CPU1} \approx 18 \) W. CPU 0 has a power of \( P_{CPU0} = 65 \) W. Considering, of course, that one CPU with 8 threads needs double the time compared to 2 CPUs and only power, but not energy is saved.

4.6 Data transfer

For an extensive study of the energy consumption of the GPU execution, the energy needed to transfer the data to the GPU must not be disregarded. For the transferring data from the host to the device memory and vice versa, CUDA offers the function `cudaMemcpy`. Figure 5 shows the time needed for copying a given amount of data between the two memories. The regression lines have been determined using the Levenberg-Marquardt least-squares method, the outliers have been left out of consideration for the regression. The results show that the data transfer rate from the host to the device memory is 3241 MiB/s, from the device to the host memory 2859 MiB/s. These values are in the same order of magnitude as the results in [9] for the Tesla C2050. The host memory has not been pinned to specific memory pages as this would be impractical in the application, the FEM.

The power consumption of the GPU has been determined to be 83.2 W during the data transfer. Writing to the device memory needs total 25.7 J/MiB and reading needs 29.1 J/MiB, which is 3.06 µJ/bit and 3.47 µJ/bit, respectively. However, it is not possible to isolate the power consumption of the GPU memory, so these values do also comprise the energy needed by the GPU processor.

For measuring the energy consumption of accesses to the main memory, the RAPL register `MSR_DRAM_ENERGY_STATUS` has been read for both CPU packages before and after the data transfer. The data is stored in both memories as the NUMA policy `interleave` [16] was used for the allocation. For a comparison measurement, the CPUs have been left idle for exactly the same time as needed for the data transfer using `nmap asleep` and the energy consumption of the memory has been measured in the same way. This method allows to determine the amount of energy needed for reading or writing one data word from or to the memory. In order to avoid that memory accesses are served by the cache, an array twice as big as the caches has been written before each measurement. The measurements have been performed for data sizes which are in the same range as they
with respect to the CPU, i.e. without needing any considerable CPU time. Hence, the total energy consumption for transferring one bit is between 180 and 270 µJ, which is considerably higher than the value of 100 µJ shown as the axes are cropped.

For exact results the measurement method needs some refinement. With the method used, only a rough indication of the real conditions can be given. In fact, the memory access energy should be constant, i.e. independent from data sizes. The energy for reading one bit is between 100 and 250 pJ. The energy for writing one bit is a little higher: it is between 180 and 270 pJ. Despite the deficiencies of the experiment, its results are in the same order of magnitude as the value of 100 pJ/bit estimated in [30]. Nevertheless, the question of how to accurately measure memory access energy using RAPL has to be investigated further.

For a finite element consisting of 27 nodes with 3 degrees of freedom each, 27 kB of data have to be transferred from the main memory to the GPU memory before starting the CGM. This data remains constant during all CGM iterations, i.e. all AXMEBE calls. The data transfer needed for each individual AXMEBE call is already included in the values measured in Sect. 4.3. The energy needed by the DRAM is 22...54 µJ per element; the energy needed by the GPU is 41 µJ per element. The energy needed for running the CPU can be disregarded for the data transfer as the function \texttt{cudaMemcpyAsync} allows the data to be transferred asynchronously with respect to the CPU, i.e. without needing any considerable CPU time. Hence, the total energy consumption for transferring one element for PPCGM is at maximum \( E_{\text{copy}} = 95 \mu J \). As there are generally 10 to 20 iterations before the CGM stops, the energy is divided by the respective number to get the amount of data transfer energy to be attributed to one AXMEBE call.

5. Energy Model

The total energy consumption \( E_{\text{tot}} \) for AXMEBE consists of the energy consumption of the CPU \( E_{\text{cpu}} \) and the energy consumption of the GPU \( E_{\text{gpu}} \):

\[
E_{\text{tot}} = E_{\text{cpu}} + E_{\text{gpu}}.
\]

The energy consumption of each processing unit can be expressed by the per-element energy consumption \( E_{el} \) multiplied by the number of elements processed \( n_{el} \) on the respective processing unit. For the GPU, the costs for the data transfer \( E_{\text{copy}} \) have to be added:

\[
E_{\text{cpu}} = n_{\text{el,cpu}} \cdot E_{el,cpu}
\]

\[
E_{\text{gpu}} = n_{\text{el,gpu}} \cdot \left( E_{el,cpu} + E_{\text{copy}} \right)
\]

The per-element energy on the CPU depends on its frequency \( f \) and the execution speed \( v_{ex} \) at frequency \( f \):

\[
E_{el,cpu}(f) = \frac{P(f)}{v_{ex}(f)}.
\]

This results in an overall formula for executing with both, CPU and GPU:

\[
E_{\text{tot}} = n_{\text{el,cpu}} \cdot \frac{P(f)}{v_{ex}(f)} + n_{\text{el,gpu}} \left( E_{el,cpu} + E_{\text{copy}} \right).
\]

The model is valid only for times when the processing units are executing workload, it does not capture idle times.

6. Discussion

The frequency scaling experiments in Sect. 4.2 showed that the power consumption of a CPU grows cubically with increasing clock speed. Furthermore, the common rule for calculating the power consumption of CMOS circuits, eq. (10), could be confirmed. The execution speed depends linearly on the frequency, which is intuitive for compute-bound parts of the code. Additionally, a small fraction in the code could be identified whose execution speed is frequency-invariant. Combining both laws leads to the following formula for computing the energy needed by the CPU for process one element with AXMEBE:

\[
E_{el,cpu} = \frac{P(f)}{v_{ex}(f)} = \frac{a f^3 + b f + c}{d f + e}
\]

where \( a \) through \( e \) are variables determined by regression analysis.

The results in Sect. 4.3 show that performing the calculations on the GPU saves roughly 13% of energy compared to the CPU. The per-element energy on the GPU is \( E_{el,cpu} = 235 \mu J \) compared to \( E_{el,cpu} = 265 \mu J \) for the CPU running with the highest frequency and \( E_{el,cpu} = 321 \mu J \) for running with the lowest frequency. Also taking into account the energy needed for transferring the data from the main memory to the GPU memory as investigated in Sect. 4.6 changes the situation substantially. The total amount of data-transfer energy to be attributed to one AXMEBE call is roughly \( E_{\text{copy}} = 63...95 \mu J \), i.e. approximately one third of the computation energy \( E_{el,cpu} \).

The conclusion is that under consideration of the data transfer energy, it depends on the number of iterations of the CGM whether AXMEBE is executed more energy-efficiently on the CPU or on the GPU. If the number of iterations is less than 3, executing AXMEBE completely on the CPU is more energy-efficient. With

![Figure 6. Energy needed for accessing the main memory](image)
greater numbers of iterations, transferring the data to the GPU and executing AXMEBE there is more energy-efficient.

If the data is already situated in the GPU memory, possibly due to previous steps of the FEM executed on the GPU, the data transfer energy can be neglected. Table 1 shows the energy which would be needed for executing AXMEBE with 100 elements for different distributions of the elements to the processing units neglecting the data transfer energy. The distributions assume that all CPU cores together process as many elements as the GPU in a given time. Consequently, if only half of the CPU cores are operating, one third is processed on the CPU and two thirds on the GPU. The more energy-efficient execution on the GPU suggests that all computation could be off-loaded to the GPU and the CPU meanwhile switches to a power-saving idle mode. Experiments showed that the first CPU, whose first core is responsible for controlling the GPU in the implementation, does not throttle down. So, if it is impossible to save energy, the energy used anyway should be used in a productive way: for performing computation. In contrast to the first CPU, the second CPU does throttle down when it does not execute any thread. With this configuration, a small amount of energy (below 2%) can be saved compared to utilising both CPUs. If the GPU is removed from the machine, i.e. it does not need any power, the resulting energy consumption will be higher by approximately 6%. To prove the significance of the results, further investigations would be needed, especially with respect to the small differences.

### Table 1. Energy needed for processing 100 elements with AXMEBE neglecting the data transfer energy

<table>
<thead>
<tr>
<th>Percentage of elements</th>
<th>Energy</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 %</td>
<td>23.5 mJ</td>
<td>theoretical value</td>
</tr>
<tr>
<td>33 %</td>
<td>24.9 mJ</td>
<td>CPU 1 throttled down</td>
</tr>
<tr>
<td>25 %</td>
<td>25.3 mJ</td>
<td>both CPUs running</td>
</tr>
<tr>
<td>50 %</td>
<td>27.0 mJ</td>
<td>GPU switched off</td>
</tr>
</tbody>
</table>

For supporting application developers, a tool automating the process and the experiments described in this article would be helpful. Such a tool could help finding laws for the energy consumption of the application considered or for identifying code sections offering potential for energy-efficiency improvements. Despite the existence of tools that automate the construction of performance and power models, e.g. by regression and significance analysis such as [27], modelling without any human intervention is impossible. A software tool could, for example, offer the possibility to add calls to measuring functions at the crucial points of the application or perform the evaluation even more independently such as a execution-time profiler does nowadays. If this tool supplied the developer with charts showing the influences of the various parameters on the energy consumption, he or she could easily derive an energy model for the application and constrain him- or herself to the “creative” work for which machines are unsuitable.

For more exact results, improvements in the measurement capabilities are needed. Desirable would be a higher temporal resolution of the energy MSRs in the CPU and the power meter of the GPU. Furthermore, energy measurement would be simplified on the GPU if it offered an interface for retrieving the energy, compared to the power interface currently existing. Such an interface could be implemented by integrating the power in hardware or in the GPU driver.

The findings of this work are supported by [19] and [12]. [19] sets up an execution time and power model for a sparse CGM and also finds a linear relationship between frequency and execution time as well as a super-linear relationship between frequency. [12] implements an energy-efficient CGM and finds that the execution on the GPU is more energy-efficient than an execution on the CPU.

### 7. Conclusion

This article has presented experiments investigating the energy consumption for executing a conjugate gradient method on both, CPUs and GPUs. The results show that the total energy used is proportional to the number of elements processed on both kinds of processing units. The frequency-scaling experiment shows that decreasing the clock frequency of the CPU leads to a decrease of the power consumption, but to an increase of the per-element energy on the CPU. The per-element energy for computation on the GPU is approximately 13% lower than on the CPU. The energy needed for transferring the data from the main to the GPU memory for the whole CGM is roughly 40% of the computational energy of the GPU for one iteration of the CGM.

From the experimental results, an energy model has been derived and conclusions for an energy-aware distribution of the workload between CPU and GPU have been drawn. For the system used for the experiments, the best strategy seems to be executing the CGM in parallel on the first CPU and on the GPU allowing the other CPU to go into idle mode. The energy model developed allows to predict the energy consumption depending on the number of elements processed on the GPU and on the CPU and the clock frequency of the CPU. From the energy prediction, the most energy-efficient distribution of the workload can be inferred.

The experiments show the necessity for further refinement of energy measurement facilities which are accessible by software. The results of this article lay the foundation for a more extensive energy analysis of the FEM resulting in an energy model comprising the whole application.

### Acknowledgments

This work is supported by the cluster of excellence Energy-Efficient Product and Process Innovation in Production Engineering (eniPROD) funded by the European Regional Development Fund (ERDF) and the Free State of Saxony.

### References


Abstract
During the last decade, parallel processor architectures have become a powerful tool to deal with massively-parallel problems that require High Performance Computing (HPC). The last trend of HPC is the use of heterogeneous environments, that combine different computational power units, such as CPU-cores and GPUs. Performance maximization of any GPU parallel implementation of an algorithm requires an in-depth knowledge about its underlying architecture, becoming a tedious task only suited for experienced programmers. In this paper, we present TuCCompi, a multi-layer framework that not only transparently exploits heterogeneous systems, but automatically tunes the GPU capabilities by choosing the optimal values for their configuration parameters, using the kernel characterization provided by the programmer. This model is very useful to tackle problems characterized by independent, high computational-load tasks with none or few communications, such as embarrassingly-parallel problems. We have evaluated TuCCompi in different, real-world heterogeneous environments using the APSP problem as a case study.

Categories and Subject Descriptors D. Software [Programming techniques]: Concurrent Programming

Keywords APSP, Auto-Tuning, CUDA, GPU, Heterogeneous system, HPC framework, MPI, OpenMP, Parallel model

1. Introduction
Some computing-intensive problems are divided into many independent tasks that can be executed in parallel, and that do not require any communication among them. They are called embarrassingly-parallel problems [7]. Many problems are included in this category, such as index processing in web search [8], bag-of-tasks applications [3], traffic simulations [21], some molecular physics computations [2], biomedical-domain problems [12] or Bitcoin mining [22].

Although the parallelization of embarrassingly-parallel problems does not require a very complex algorithm to take profit of parallel computing environments, their high amount of computational work requires High Performance Computing (HPC). In order to give support to the massive demand of HPC, the last trends focus on the use of heterogeneous environments that include computational units of different nature. These computational units include common CPU-cores, graphic processor units (GPUs) and other hardware accelerators. The exploitation of these environments offers a higher peak performance and a better efficiency compared to the traditional homogeneous cluster systems [1]. Due to these advantages and to the low cost of building heterogeneous systems, they are being incorporated into many different computational environments, from small academic research clusters, to supercomputing centers.

Despite of the wide use of heterogeneous environments to execute massively-parallel algorithms, there are two issues that limit the usability of these environments. The first one is the lack of global computing frameworks that easily schedule the workload in such complex environments. Some works have tried to ease the jointly use of parallel programming languages, such as MPI or OpenMP, by the creation of different tools. For example, a source-to-source compiler that translates C annotated code to MPI + OpenMP or CUDA code is presented in [19]. However, in that work CUDA can not be jointly used with the other parallel models. Another example is OMPICUDA [13], a framework to develop parallel applications on heterogeneous clusters by mixing OpenMP and MPI. In this work OpenMP code is translated to CUDA, however, this code has serious programming limitations. Moreover, these works do not exploit all computational capabilities of the GPUs.

The second limitation is the lack of a tuning methodology that efficiently unleashes the power of GPU devices. There is not known a parallel model that automatically selects the optimal values for CUDA configuration parameters, such as the threadBlock size, or the state of L1 cache memory, for each kernel. These optimization techniques significantly enhance the GPU performance. Although languages such as CUDA aim to reduce the programmer’s burden in writing parallel applications, it is a difficult task to correctly tune the code in order to efficiently exploit all underlying GPU resources. Several studies [23, 24] have shown that in some cases the values recommended by CUDA do not lead to the optimum performance, leaving to programmers the task of searching for the best values through time-consuming, trial-and-error tests.

In this paper we present TuCCompi (Tuned, Concurrent CUDA, OpenMP and MPI), a multi-layer computing framework that transparently exploits heterogeneous systems and squeezes the GPU capabilities by automatically choosing the optimal values for important configuration parameters. Each layer represents a level of parallelism (see Fig. 1). The first layer handles the distributed-memory environment, coordinating the nodes. The second layer manages the computational units which belong to the shared-memory environment inside a node. The third layer automatically deploys...
the execution in hardware accelerators such as GPUs. The fourth layer automatically handles concurrent works inside these GPUs. Finally, an internal Tuning mechanism automatically selects the optimal values for GPU configuration parameters for each kernel and each GPU architecture. We have developed a prototype framework to test this model, allowing any user to transparently take advantage of all computational capabilities of both, CPU-cores and GPU devices, distributed in different shared-memory systems, without having a deep knowledge of parallel programming methods. The case study used to evaluate the model is the All-Pair Shortest-Path (APSP) problem. Experiments have been run in an academic heterogeneous environment.

The contributions of this work are: (a) Mechanisms to automatically choose the optimal values for important CUDA configuration parameters, based on provided programmer kernel characterization, for any current kind of GPU architecture; (b) The automatic exploitation of a modern GPU feature such as the concurrent-kernel execution as a new dimension of parallelism; (c) The creation of a specific prototype framework that combines the use of these two novel layers and the traditional ones, whose use leads to performance improvements up to 12% in a test case.

The rest of this paper is organized as follows. Section 2 introduces our conceptual approach. Section 3 describes the use of the model through some code snippets. Section 4 explains the case study used. In Sect. 5 we present the experimental environment and the results obtained. Section 6 describes some related work. Finally, Sect. 7 summarizes our conclusions.

2. TuCCompi Architecture

This section gives a description of the different layers defined in our model. The use of a multi-language framework provides the user with mechanisms to obtain a good performance, tuning the devices in an optimal way. A graphical representation is depicted in Fig. 1.

The 1st layer (distributed environment) Nowadays, one of the most economic ways to assemble a heterogeneous system is to interconnect a set of individual machines, also called nodes, such as personal computer, laptops, complex virtual host machines or even other supercomputing systems composed in turn by other machines. The nodes found in these heterogeneous environments usually consist in shared-memory systems, with very different computational capabilities. It is necessary to apply communication and synchronization mechanisms in order to coordinate these machines for the parallel resolution of the problem. The first layer of TuCCompi (see Fig. 1) is responsible of managing the coordination of these nodes without taking into account the specific hardware details and features of each machine. In order to communicate and synchronize these nodes, we use MPI (Message Passing Interface) as message passing tool.

The 2nd layer (shared-memory systems) Most computers nowadays are composed by several processing units (we will name them CPU-cores) that share a global address space. Additionally, there are other accelerator devices, such as GPUs, FPGAs and Xeon Phi, that are usually controlled by a host system (CPU) and are capable of executing kernels independently. Although the use of these devices implies the computational sacrifice of a CPU-core, their performance is higher than the one obtained by this CPU-core. In this layer of TuCCompi we use the concept of “computational unit” for any CPU-core or device that shares the global memory hosted in a node. This second layer is responsible of the coordination of all computational units inside the node. If there were such accelerator devices in the machine, like the GPUs present in Fig. 1, this layer would automatically deploy the parallel version of the algorithm to the CPU-cores responsible of the management of those devices, and the sequential version to the rest of CPU-cores. In order to manage these resources we use OpenMP as thread-management environment.

The 3rd layer (GPU devices) An emerging way of parallel computing includes the use of hardware accelerators, such as GPUs, FPGAs and Xeon Phi. Their powerful capability have triggered their massive use to speed up high-level parallel computations. For certain problems, the use of a parallel implementation of an algorithm in these hardware accelerators can offer huge speedups against the sequential algorithms deployed in the CPU-cores. However, their management is much more complicated than any multi-core system. If these kind of devices are found in a shared-memory system, the third layer automatically deploys the parallel-algorithm execution into them, in addition to the sequential execution of the remaining node CPU-cores. In our prototype implementation we have included CUDA to manage GPU devices.

The 4th layer (concurrent GPU kernel execution) The most recently NVIDIA GPUs support concurrent-kernel execution [15, 16], where different kernels of the same application context can be executed on the GPU at the same time (see Fig. 2). If the number of resources needed to execute a kernel does not reach the total available resources, the remaining ones can be used to concurrently execute another kernel. Thus, the number of kernels that can be executed at the same time depends on the total hardware resources required by each kernel and the corresponding GPU hardware characteristics. This feature is very helpful when small kernels are launched, allowing a concurrent execution that exploits all device resources. Although at first sight this feature seems to be profitable only when small kernels are launched, the concurrent execution of bigger kernels also gives performance improvements. This occurs because several kernels of the same application context that, work on the same memory areas take advantage of the L1 data-cache, originating less number of cache-misses and therefore alleviating the global memory bottlenecks. Additionally, the threadBlock-warp dispatcher schedule kernels faster if they have been previously launched [17].

The fourth layer of TuCCompi (see Fig. 1) is responsible of the automatic launching of many concurrent kernels in modern GPUs, squeezing their computational resources. The different tasks that are scheduled to these kind of accelerators can be executed in parallel in the same device, adding a new level of parallelism.

The Tuning layer While correctness of an NVIDIA CUDA program is easy to achieve, the optimal exploitation of the GPU computational capabilities is much more complicated than in traditional CPU cores. Usually, it requires an extensive CUDA programming experience. Some examples of code tuning strategies are the choice
Table 1. TuCCompi kernel-characterization classification. The def choice can be used when the user does not know the kernel characterization.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Global memory-access pattern</td>
<td>scatter/medium-coalesced/coalesced/def</td>
</tr>
<tr>
<td>B</td>
<td>Ratio of arithmetic instructions per thread compared to the global-memory accesses</td>
<td>high/low/none/def</td>
</tr>
<tr>
<td>C</td>
<td>Ratio of L1 cache memory lines evictions compared to the size of this memory</td>
<td>high/medium/low/def</td>
</tr>
<tr>
<td>D</td>
<td>Ratio of global memory data reutilization compared to the number of arithmetic instruction per thread</td>
<td>high/medium/low/def</td>
</tr>
</tbody>
</table>

of an appropriate threadBlocks size and shape, the coalescing maximization of the memory accesses, or the occupancy maximization of the Streaming Multiprocessor, among others. Moreover, the resource differences between each GPU architecture and release, such as the number of computational units, cache-sizes, and other features, make even more difficult to find the optimal configuration for each GPU. Besides this, the optimal values also depend of the access-memory pattern and the characteristics of the code of each executed kernel. The Tuning layer automatically selects some important parameter values for each kernel and GPU, based on a kernel characterization provided by the application programmer.

3. TuCCompi Model Usage

TuCCompi users should provide to the model the following (see Fig. 3): (a) The sequential-CPU and the parallel-GPU code applications, that are named as Plug-In_CPU and Plug-In_GPU respectively, (b) the kernel characterizations used in the GPU code, and (c) the main C language program with TuCCompi primitives and macros.

In this way, the application programmer does not have to provide: (a) the values of GPU configuration parameters for an optimal execution, (b) the code implementation for concurrent kernel deployment, (c) the code implementation for the management of the distributed and shared computational-unit resources, nor (d) the initial communication between all the cluster nodes involved.

3.1 Kernel characterization

The user has to provide a general characterization of her kernels along with its definition. This information is easily expressed in our

Figure 2. Sequential (left) vs concurrent execution (right) of several kernels. When the total needed threads of a kernel surpass the maximum GPU active threads (number of concurrent threads supported by each Streaming Multiprocessor (SM) x number of SMs) CUDA driver is the responsible of splitting it into two sequential steps.

Figure 3. TuCCompi model usage. Elements in the dashed box are provided by the programmer. Note that the user can develop different versions of each plug-in (Code A, Code B, ...) but only one at a time will be deployed into TuCCompi framework.

Figure 4. Kernel characterizations and implementations. The programmer adds the boxed primitive before the kernel implementation to characterize it.

Figure 5 shows the interface of the sequential code that will be inserted the code to implement the algorithm that solves a single task (line C01, Cpu user code).

3.2 User-code Plug-ins

Figure 4 shows some examples of the code used to characterize the kernels. Lines K00 and K04 describes the characterization of kernels k1 and k2 respectively, indicating the kernel name, the number of dimensions of the threadBlock, and the classification criteria described in Tab. 1. In the case that the user does not know how to classify her kernels, she can use the default (def) values provided by the model. The primitive used for this default case is TuCCompi_KERNELCHAR(kernel_name, num_dims, A, B, C, D) primitive. The values for parameters A, B, C and D have to be chosen from the kernel-characterization classification shown in Table 1. TuCCompi model automatically optimizes the use of all underlying hardware resources of GPU devices, following the guidelines and optimizations proposed in [24] for each possible combination of these parameters.
The process has to wait until all computational units of the cluster node (described below). In line M05 the programmer specifies that the macro represents the kind of scheduling policy desired by the user. Listing computational units in parallel. The first parameter of this macro in the corresponding plug-ins, for CPUs and GPUs, using all existing elements that compounds it. As we said before, the algorithm schedules the same number of task to each computational unit of the cluster node, these asking-for-tasks requests are issued through the computational units work as slaves. The slaves enter into a working loop, requesting tasks from the master until it sends a termination signal to them. As the master can be located at any cluster node, these asking-for-tasks requests are issued through the hardware nature of the computational unit.

The third one, MS, follows a master-slave model. One computational unit is dedicated to act as the master, and the rest of the computational units work as slaves. The slaves enter into a working loop, requesting tasks from the master until it sends a termination signal to them. As can be located at any cluster node, these asking-for-tasks requests are issued through the distributed-environment communications.

4. Case study

In order to check TuCCompi framework prototype, we have chosen the All-Pair Shortest-Path (APSP) problem for sparse graphs as our case study because it is a representative example with good characteristics to evaluate the model features. Being an embarrassingly parallel problem, it suits perfectly with TuCCompi philosophy for the first three layers. Additionally, the GPU solution for this kind of problems involves three kernels of very different nature, size, and characterization. This variety allows us to check the behaviour of the fourth layer, and the tuning layer.

In this section we explain this problem in more detail and we describe the corresponding plug-ins developed for the TuCCompi model.

4.1 All-Pair Shortest-Path (APSP) problem

The APSP problem is a well-known problem in graph theory whose objective is to find the shortest paths between any pair of nodes. Given a graph $G = (V, E)$ and a function $w(e) : e \in E$ that associates a weight to the edges of the graph, it consists in computing the shortest paths for all pair of nodes $(u, v) : u, v \in V$. The APSP problem is a generalization of the classical problem of optimization, the Single-Source Shortest-Path (SSSP), that consists of launching the corresponding kernels into the GPU. Line G02 shows the TuCCompi macro that carries out a kernel launch, with the name of the kernel as first parameter, and followed by other user variables that have been previously allocated in the GPU. 

Figure 6 shows the code that will be executed in a CPU to manage the associated GPUs. The user should define the code that handles the logic control of the algorithm that comprises the use of one or several GPU kernels. This code will be the responsible of launching the corresponding kernels into the GPU. Line G02 shows the TuCCompi macro that carries out a kernel launch, with the name of the kernel as first parameter, and followed by other user variables that have been previously allocated in the GPU. 

Figure 5. Plugin_Cpu interface. The programmer adds to her code the boxed arguments to deploy the Cpu plugin in TuCCompi.

Figure 6. Plugin_Gpu interface and internal structure. The programmer has to replace the typical CUDA kernel launch primitives for the boxed TuCCompi macros.

Figure 7. User implementation of the TuCCompi main-program. The programmer has to add to her code the boxed primitives.
in computing the shortest paths from just one source node \( s \) to every node \( v \in V \).

An efficient solution for the APSP problem in sparse graphs is to execute a SSSP algorithm \( |V| \) times selecting a different node as source in each iteration. The classical algorithm that solves the SSSP problem is due to Dijkstra \cite{dijkstra1959note}. Crauser \textit{et al.} in \cite{crauser2000parallel} introduces an enhancement that tries in each iteration \( i \) to augment the threshold \( \Delta_i \) as more as possible to process more nodes in the next iteration.

### 4.2 Plug-ins for our example

Both sequential and parallel GPU codes are implementations of the Crauser algorithm. Their implementation for this problem has been taken from \cite{tucompi}. Algorithm 1 shows the GPU parallel pseudo-code of Crauser’s algorithm. Figure 9 shows the TuCCompi implementation for the pluginGPU. This implementation repeatedly launches three kernels (relax, minimum and update) with different features. Following the classification criteria described in Sect. 3.1, the kernels are characterized in Table 2.

Regarding to the scheduling issue, due to the parallel nature of the problem we have defined each SSSP computation as a single independent task. We have implemented our own master-slave scheduling plug-in (see Fig. 8). The master differentiates the nature of the slave that is requesting a task. Depending on the slaves computational power, the master will send more or less tasks. The TuCCompi model is better exploited if the master gives more tasks to the modern GPUs (Fermi, Kepler and so on) due to their multi-kernel execution feature. For our particular master, we decided to dispatch four tasks for each modern GPU, and only one for the Pre-Fermi architectures and the CPU cores.

Figure 8 (top) shows the master implementation. The master will manage the task distribution while there are tasks to be executed (lines 01-16). To do so, the master waits for a task request from any slave (line 3). If the slave is a modern GPU (Fermi or Kepler) (line 04), the master checks if there are MK available tasks to be sent. In this case, it sends the pack to the corresponding slave using its identifier, and updates the task counter (lines 05-07). However, if there are not enough tasks for this type of slave, the master sends to it the termination signal and updates the counter of slaves that have already finished (lines 08-11). If the requesting slave is an old GPU (pre-Fermi) or a CPU-core, the master only sends a single task to the slave (lines 12-15). Afterwards, the task counter is updated. When all tasks have been scheduled and carried out, the master sends to the finishing slaves the termination signal and updates the corresponding counter (lines 17-21).

Figure 8 (bottom) shows the slave implementation. First, the slave notifies the master that it is idle (line 1). Then the slave receives the task(s) to be executed (line 2). Finally, the slave returns the task identification (line 3).

<table>
<thead>
<tr>
<th>Kernel</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relax</td>
<td>scatter</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Minimum</td>
<td>coalesced</td>
<td>low</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>Update</td>
<td>coalesced</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>

### 5. Experimental evaluation

This section describes the methodology used to test the TuCCompi model, the platforms used, and the input set characteristics for the case study (the APSP problem). Finally, the experimental results and a discussion is presented.

#### 5.1 Methodology

In order to evaluate TuCCompi for heterogeneous environments, we have tested the APSP problem as a case study (see Sect. 4) in different scenarios. Each scenario was designed with the aim to check the use of the layers involved in each scenario in an incremental fashion. Architecture details are shown in Table 3:

- A single GPU, that uses the 3rd, 4th and the tuning layer.
Two GPUs, that involve the 2nd layer in addition to the previous ones.

**Pegaso:** A shared-memory system with two GPUs and eight CPU-cores (two for handling the GPUs and six for computing), in order to test the 2nd layer by mixing two different kinds of computational units.

**Small HC:** Small heterogeneous cluster, that uses all layers of TuCCmpi.

**Big HC:** Big heterogeneous cluster, with the aim to evaluate the scalability of the model.

The workload scheduling used for these environments was the particular master-slave policy described in Fig. 8.

Finally, with the aim of testing the performance gain offered by the innovative 4th and Tuning layers, we have compared the execution of a single GPU without these layers with respect to the concurrent kernel execution one kernel per time, together with the threadBlock values recommended by CUDA, with respect to the concurrent kernel execution.

We can observe that the execution times have been reduced as more computational resources are used. Moreover, the use of this model has a communication overhead lower than 1 percent. The overhead of the Small-HC have never surpassed 0.589% of the total execution time. Figure 11 represents the task distribution because we do not have to use padding techniques to avoid buffer overrun errors.

### 5.4 Experimental results

#### GPUs vs the heterogeneous environments

Figure 10 shows the execution times for the single GPU, the multi-GPU system and the two heterogeneous cluster scenarios. Although the GPUs are the most powerful devices, and their combined use significantly decreases the execution times, the addition of many less-powerful computational units enhances even more the total performance gain. We can observe that the execution times have been reduced as more computational resources are used. Moreover, the use of this model has a communication overhead lower than 1 percent. The overhead of the Small-HC have never surpassed 0.589% of the total execution time. Figure 11 represents the task distribution between the Big HC nodes for the executed master-slave policy. Furthermore, the figure shows the theoretical distribution for each cluster node if the equitable policies, EQ1 and EQ2, were used.

### 5.5 Input Set Characteristics

The input set is composed by a collection of graphs randomly generated by a graph-creation tool used by [14] in their experiments. The graphs have been created adding seven adjacent predecessors to each node of the graph. Afterward, graphs are inverted in order to store the node successors sequentially. These graphs are represented through adjacency lists, with the nodes numbered from 0…[|V| − 1], and integers that randomly range from 1…10 for edge weights.

We have used four different graph-sizes, whose number of vertices are 1 049 088, 1 509 888, 2 001 408 and 2 539 008. These sizes have been chosen because they are multiple of the threadBlock sizes considered. In this way the GPU algorithm is easier to implement because we do not have to use padding techniques to avoid buffer overrun errors.

### 5.2 Target Architectures

Table 3 describes the heterogeneous platforms used for our experiments. For each node we indicate the number of CPU-cores and the GPUs used. This heterogeneous cluster contains a total of 180 CPU-cores and 3 GPUs. However, each GPU device is governed by a single CPU core, thus, the total number of real computational units is 180. The multi-GPU systems include the two devices present in the Pegaso machine, and the single GPU scenario uses the most powerful of them, the GeForce GTX 480.

The different nodes shown in Table 3 run the Ubuntu Desktop 10.04 (64 bits) operating system. The CUDA toolkit release used is 4.2 with the 295.41 64-bit driver.

## 6. Related work

IICoMP [19] is a source-to-source compiler that translates C annotated code to MPI + OpenMP or CUDA code. The user needs to specify the sequential code that she wants to parallelize. The
The performance improvement of the 4* layer

Figure 11. Number of executed tasks per cluster node with different distribution policies in the big heterogeneous cluster.

Figure 12. Performance improvements obtained by the 4th and Tuning layers with respect to CUDA recommended configuration values.

Programmer Application

Figure 13. Usage of TuCCompi with code-transformation modules.

authors are only focused in parallel-loop problems. This compiler does not support the jointly use of CUDA with any other parallel model, therefore, it is not appropriate to be used in heterogeneous environments. Besides this, the ICoMP compiler does not easily support a new GPU architecture or other kind of accelerators.

The authors in [13] propose a framework called OMPICUDA to develop parallel applications on the hybrid CPU/GPU clusters by mixing OpenMP, MPI and CUDA models. Besides this, they include a compiler that translates automatically OpenMP source code to CUDA. This framework presents serious programming limitations. First, it does not support any recursive function. Second, the 1.X CUDA architectures can not be used because of the way pointers are used. Third, the critical OpenMP sections are not fully translated to CUDA. Fourth, this framework can not be easily modified to support a new parallel model. Finally, they do not develop any policy to select the proper values of CUDA configuration parameters. A parallel programming approach using hybrid CUDA, MPI and OpenMP programming is presented in [25]. The authors only focus on the model to solve iterative problems and they do not take into account any CUDA optimization technique. The proposed model does not support any mechanism to include new work distribution policies.

The authors in [9] have created an hybrid tool that includes the same parallel models used by the previous mentioned works to solve raycasting volume rendering algorithm. They test the system scalability when the input data size is increased. This tool is only focused in a single parallel application and does not include any CUDA optimization technique. Moreover, this work does not include any automatic mechanism to efficiently exploit all available hardware devices in heterogeneous environments. Additionally, the proposed models can not be used for other kinds of parallel problems.

Another work in this field is StarPU [10], a task programming library for hybrid architectures supporting GPUs. However, to the best of our knowledge, StarPU does not include the concurrent-kernel feature of modern GPUs, nor our tuning techniques for better exploiting GPU computational capabilities.

With respect to sequential-to-parallel code transformation, proposals in this field include accULL [20], that receives a sequential code of an algorithm and automatically transforms it to parallel code that can be deployed into GPU devices. Another example of code transformation is Ocelot [6], that works in the opposite way. Given a GPU implementation, Ocelot transforms it to sequential code. TuCCompi model does not aim to deal with sequential-to-parallel code transformation. However, both proposals described above and many others can be easily attached to our multilayer model (see Fig. 13). Additionally, in order to solve the automatic GPU kernel characterization, it is also easy to attach a module that analyzes the GPU implementation and connects its output to the Tuning layer.

7. Conclusions and future work

We propose TuCCompi, a multilayer deployment model that helps the programmer to easily obtain flexible and portable programs that automatically detect at run-time the available computational resources and exploits hybrid clusters with heterogeneous devices. This model offers to the programmer a transparent and easy mech-
anism to select the optimal values of GPU configuration parameters just characterizing the nature of the kernels. Any parallel application that can be devised as a collection of non-dependent tasks working on shared data-structures can be exploited with the current model of TuCCompi.

The use of new layers to exploit a concurrent kernel execution in GPUs and optimal parameters adds a novel parallel dimension and a new automatic optimization compared with previous works, representing in our test case a performance gain up to the 12% for the GPUs usage. Therefore, these new layers turn out to be very important for heterogeneous environments that include these GPU devices.

The model is designed to provide a mechanism of plug-ins, in order to easily change: (1) The algorithms to be deployed; (2) the scheduling policies of the tasks; and (3) the parameter values for GPUs optimal configurations without making any change in the model. Furthermore, it is easy to exploit tools generated by other research works related with parallel code transformation in order to give a complete single programming environment to the user. The use of this model exploits even the less powerful devices of a heterogeneous cluster and correctly scales if more computational units are added to the environment, with a communication overhead less than one percent of the total execution time.

As future work, we plan to devise other uses for TuCCompi, including massively parallel problems such as Bitcoins currency mining or molecular computations, as well as other kind of parallelizable problems.

Acknowledgments

The authors would like to thank Javier Ramos López for his support with technical issues. This research has been partially supported by Ministerio de Economía y Competitividad (Spain) and ERDF program of the European Union: CAPAP-H4 network (TIN2011-15734-E, MOGECOPP project (TIN2011-25639); and Junta de Castilla y Len (Spain): ATLAS project (VA172A12-2).

References


Abstract

GP-GPUs have been considered since long time as suitable accelerators for data parallel computations. Recent improvements in the supported features make this kind of devices amenable to support other kind of parallel patterns. In this work we discuss preliminary experiences aimed at supporting stream parallel computations on GPUs. In particular, we consider the possibility to execute streams of data parallel tasks on a GPU, we introduce analytical performance models for these kind of computations and we show how these models may be used to predict the “optimal” parameters to be used in the execution of the task stream on GPUs.

Keywords: Stream parallel patterns, GPU, structured parallel programming models

1. Introduction

Since their introduction, GP-GPUs have been used as accelerators of data parallel computations in high performance computing contexts. The huge number of cores included in high end GP-GPUs demonstrated particularly efficient in the execution of a number of different data parallel kernels, not necessarily “embarrassingly parallel”. Moreover, the FLOP per Watt sported by GP-GPUs make these devices even more interesting in the context of HPC, as they de facto promise to lower the power footprint of HPC computations with respect to the one relative to the usage of more classical shared memory multi core architectures.

As a result, GP-GPUs equip different supercomputers in the top500.org list and—as of June 2013—they are key components of the two top positions in the green500.org list. At the other end of the computer systems—namely in tablets, laptops and desktop systems—GP-GPUs are used as graphic accelerators but the current state-of-the-art GPU development tools allow owners of these machines to accelerate their data parallel applications using the very same techniques adopted—with a completely different scale—in top500 systems.

Recent advances in GP-GPU technology introduced some features that can be used in a radically different context. nVidia GPUs, for instance, include two different DMA engines in addition to control processors and GPU cores, such that data transfers from host to GPU memory and from GPU to host memory may be overlapped with kernel computations on the GPU. Moreover, the CUDA framework introduced the concept of GPU operation stream, that is of sequence of data transfers (to and from GPU memory) and kernel operations that primitively supports overlapping of data transfer and computation times. As a final step, the recent GPU control logic supports the concurrent execution of different kernels on the single GPU.

All these features inspired us the idea that GP-GPUs may be used also to improve the performance of the execution of different parallel patterns, namely those typical in stream parallelism. In stream parallel applications parallelism comes from the concurrent execution of different input tasks appearing at different time on an “input stream” rather than coming from the decomposition of a large computation into sub-computations as it happens in data parallelism. Therefore we investigated the possibility to use concurrent kernel execution provided by modern GPUs to support the parallel execution of different tasks in stream parallel patterns such that the stream parallel computation time is eventually shortened. In this work we will therefore discuss some preliminary experiments with GP-GPUs cores partitioned for the execution of different tasks (kernels) coming from a stream of tasks rather than using the same cores to execute single data parallel tasks. Our contribution may be summarized as follows:

- we developed an analytical performance model suitable to figure out the optimal number of CUDA streams to be used to implement the computation of a stream of data parallel tasks on a GPU
- we implemented stream parallel computations on GPUs, using FastFlow as stream parallel computation orchestrator and nVidia Tesla or Kepler GPU as target accelerator. This implementation demonstrates the feasibility of the implementation of stream parallel applications on GPUs as well as the suitability and accuracy of the performance model.

The rest of this paper is organized as follows: Sec. 2 introduces the parallel patterns used in stream parallel applications, Sec. 3 introduces the features of GPUs we used to accelerate stream parallel applications, Sec. 4 discusses our analytical performance model for stream parallel applications on GPUs, and eventually Sec. 5 and Sec. 7 draws conclusions and summarizes ongoing and future work.

2. Stream parallel patterns

Parallel design patterns may be classified according to different principles. The authors of [12] introduce classes related to the different usage made of the patterns, namely to expose concurrency of the application at hand, model the algorithm used, model the implementation and the mechanisms used in the implementation. In more classical parallelism books [17] parallel patterns are simply classified as either data parallel or stream parallel. In particular:
• Data parallel patterns are those patterns where parallelism arises from the concurrent computation of sub collections (partitions or overlapping subsets) of the input task. Each of the concurrent computations computes a partial result. Eventually the partial results are combined to produce the actual result of the computation.

• Stream parallel patterns, instead, are those patterns where parallelism arises from the concurrent computation of different input tasks appearing on the input stream at different times.

Common data parallel patterns include:

• The map pattern, applying the very same computation to all the items in an input collection and giving in output the collection of the sub results.

• The reduce pattern, “summing up” all the items in an input collection using a binary, associative and commutative function.

• The stencil pattern, computing a output collection out of an input collection, where the components of the output collection derive from the application of a function over overlapping subsets of the input collection.

whereas stream parallel patterns usually include:

• The pipeline pattern, applying in sequence a set of computations on the same input task. We will denote a pipeline pattern as pipeline \( f_1, \ldots, f_n \) where the different \( f_i \) are the pipeline stages. This pipeline applied to input stream tasks \( \langle x_1, \ldots, x_m \rangle \) computes the output stream \( \langle f_1(x_1), \ldots, f_n(x_m) \rangle \). The parallelism degree of the pipeline is \( n \), as stage \( f_i \) will compute in parallel to all the other stages, with all the stages working on (partial results relative to) different input tasks \( x_j \).

• The farm pattern (also known as master/worker) where the same computation is applied on all the items of the input stream. We will denote the farm pattern as farm\( n_w \) \( (f) \) where the \( f \) denotes the farm “worker” and \( n_w \) denotes the parallelism degree used to compute the farm. This farm applied to same input stream considered for the pipeline will compute the output stream \( \langle f(x_1), \ldots, f(x_m) \rangle \).

Typical stream parallel computations may be easily defined using pipelines and farms. As an example an application “cleaning” a video stream applying in sequence a couple of filters \( F_1 \) and \( F_2 \) may be defined as the pipeline pipeline \( v2is, F_1, F_2, is2v \) where the two stages \( v2is \) and \( is2v \) transform a video stream into a stream of image frames and vice versa. In case the two filter cost much more—in terms of the execution time per single frame—than the first and last pipeline stage, a better parallel structuring of the application can be structured as

\[
\text{pipeline}(v2is, \text{farm}_{n_w}(F_1), \text{farm}_{n_w}(F_2), is2v)
\]

with \( n_w' \neq n_w'' \) in case of different times in the computation of the two filters.

Different structured parallel programming frameworks implement the pipeline and farm patterns [2, 7, 10] and demonstrate efficiencies close to ideal for applications exploiting these patterns [5]. However, all these frameworks provide implementations of the stream parallel patterns only targeting multi cores and COW/NOW architectures, as these patterns—although very useful to model a number of different applications in different applicative domains—are not suitable to target modern accelerators such as GPUs. This comes from the fact that GPUs are mostly conceived to speed up data parallel computations with their limited number of “control processors” (in the range of units or tens) controlling a huge number of “cores” (in the range of thousands) only suitable to execute the same instruction(s) scheduled—better, broadcasted—by the control processors.

3. GPU features supporting stream parallel pattern acceleration

Recently, classical GPU architecture has been extended with additional features that in our opinion are definitely interesting to support stream parallel computations in addition to data parallel ones, namely ¹:

• GPUs have been equipped with multiple DMA engines suitable to support data transfers to and from GPU memory in parallel to the computation managed by the GPU control processors on the GPU cores. Initial implementation of GPUs only supported sequential data transfers w.r.t. to the execution of kernels or had just a DMA engine, such that only partial overlap of data transfers with kernel computation may be achieved.

• CUDA provides a stream concept that can be used to direct kernels and data transfer operations to the GPU in such a way that items of the same stream are executed in order but items from different streams can be executed in arbitrary order.

• The GPU hardware now support concurrent kernel execution, that is the concurrent execution of kernels requiring only a partition of the GPU resources (memory and cores).

In particular, according to the nVidia documentation, these features may be used to overlap data transfers (input tasks to GPU and results from GPU memory) and kernel computations but also computations of different kernels from different CUDA streams. Fig. 1 outlines these possibilities ("par-comm" and "par-kern" in the figure), compared with the standard (old style) usage of GPUs that requires the copy of input data to the GPU memory, the execution of the kernel and the copy of the results back to the main (CPU) memory in sequence (“seq” in the figure) [15].

Having these possibilities offered by recent GPUs (Tesla and Kepler ones, as an example) in mind, we tried to devise proper methodologies suitable to support the execution of stream parallel applications on GPUs. In particular, we concentrated on applications where: i) farm stages are present, ii) and the farm workers compute data parallel tasks iii) the data parallel task code may be implemented by a GPU kernel. As an example, we considered an application whose parallel structure may be described in terms of pipeline, farm and map patterns as follows:

\[
\text{pipeline}(S_1, \text{farm}_{v}(\text{map}(f)), S_3)
\]  

The worker of the farm, in this case, applies a function \( f \) over all the items in the input data collection (e.g. all the items in a vector on in a matrix). The first and last stage of the pipeline just create a stream of tasks and store results from a stream of results, respectively. A number of applications from different domains sport this parallel structure, indeed, included the video filtering application mentioned in Sec. 2. It is worth pointing out that:

• We focused our attention on farm stages as these stages are aimed at processing in parallel the tasks appearing onto the farm input stream. We aim at executing in parallel these tasks using the GPU resources.

• We focused on data parallel farm workers, as the data parallel computations are naturally suited to be executed on the GPU. In case the farm workers are not data parallel, we’ll consider the possibility to await a given amount of time collecting the

¹ We refer here to nVidia GPUs and to the CUDA development tools, although most of the things mentioned also apply to GPUs from other vendors and to the OpenCL tool suite.
farm tasks appearing on the input stream in vector and then executing the computation of the vector of tasks through a GPU map, mapping the worker computation on each one of the vector elements\(^2\).

The methodology we propose to run these kind (meta-pattern) of applications can be summarized as follows:

i) we replace the farm pattern in the pipeline by a sequential stage, such that eventually the pattern actually executed is a pipeline\((S_1, \text{seq}(\text{map}(f)), S_3)\) (2)

ii) in the sequential stage, we direct the map\((f)\) tasks to the GPU using an appropriate number of independent CUDA streams, such that the computation of the different kernels appearing onto the input stream may happen in parallel. This means when the program is started, a suitable number of CUDA streams are created in the seq\((\text{map}(f))\) stage, then each new task appearing onto the seq\((\text{map}(f))\) input is directed to the “next” stream\(^3\).

iii) we use proper analytical models to figure out the optimal number of streams to be used in the sequential stage, depending on the features of the data parallel tasks (kernel execution time, amount of input data to be transferred to the GPU, amount of data (results) that have to be transferred back to the CPU after kernel execution.

In the next sections, we will outline how the models used in the last step may be derived and how an application may be implemented following this methodology on a machine equipped with a GPU using FastFlow [1] as the stream parallel programming framework.

4. Analytical performance model

In this Sec. we discuss how we derived a performance model suitable to figure out the optimal number of CUDA streams to be used to direct data parallel tasks to GPU for execution when implementing the pipeline stage replacing the farm pattern, as discussed in Sec. 3. In the following, we will denote with the term “synchronous version” the sequential execution of data parallel kernels as depicted in Fig. 1 (seq). We will also use the term “asynchronous version” while referring to the CUDA stream based execution of data parallel kernels according to the model shown in Fig. 1 (par-comm).

First of all, let us state clearly the problem we want to solve. An arbitrary number of CUDA streams may be used to direct kernels to GPU. However, creating too few streams doesn’t benefit much from streams and creating too many has an associated overhead of managing such number of streams since the overhead of stream creation and deletion increases linearly with the number of streams. Hence, the programmer has to ask the question “What is the optimal number of streams that result in the minimum completion time that could be achieved due to the introduction of streams?”

To answer this question, we have to consider applications with different timing behaviour. One is when the data transfer times to/from the device dominates the overall completion time of an application and the other is when the kernel execution time is the dominant time (see Fig. 2). Moreover, the amount of data transferred from the host to the device is not always equal to the...
amount transferred from the device to host which implies varying transfer times depending on the transfer direction.

Let’s start with the case where the transfer time dominates the completion time, that is, kernel execution time is significantly smaller than input data and output result transfer time.

We consider our GPU computations modelled according to the schema shown in Fig. 1 (par-comm) as a three stage pipeline, where the first stage implements data transfers of input data from host to device memory, the second stage actually computes the kernel on the input data and the third stage implements the device to host result transfer.

The service time of this pipeline is given by

\[ T_{S\text{-pipe}} = \max\{T_{S\text{-h2d}}, T_{S\text{-k}}, T_{S\text{-d2h}}\} \]

where, \( T_{S\text{-pipe}} \) represents the total pipeline service time, \( T_{S\text{-i}} \) represents the service time of pipeline stage \( i \).

The optimal service time of the pipeline can be achieved when the stages are balanced, i.e.,

\[ T_{S\text{-h2d}} \approx T_{S\text{-k}} \approx T_{S\text{-d2h}} \]

\[ \frac{T_{S\text{-h2d}}}{T_{S\text{-d2h}}} \text{ is actually the time to transfer } I \text{ bytes of data to/from device memory at the rate of } B \text{ bytes per time unit, and each such transfer has an additional overhead, } T_{\text{setup}}. \]

Similarly, the Kernel stage has two costs. The first one is associated with the overhead of kernel launch and the second is associated with the actual execution of the kernel; thus,

\[ T_{S\text{-k}} = T_{K\text{-launch}} + T_{K\text{-exec}} \]

For this particular pipeline structure, there is no cost associated with the inter-stage links that would otherwise be if there were data that actually moves among the stages. The stages are simply CUDA operations that are scheduled to execute concurrently by the CUDA runtime system.

When considering computations where the input data communication time dominates the kernel time, the CUDA Best Practice [13] estimates the completion time of the asynchronous version of the application to be:

\[ T_{C} = T_{tr} + \frac{T_{E}}{n} \]

where \( T_{tr} \) represents the data transfer time, \( T_{E} \) the kernel time and \( n \) the number of CUDA streams relative to the asynchronous version of our application.

Equation 6 assumes \( T_{tr} \) and \( T_{E} \) are comparable so that the loads on the stages are balanced and satisfy the constraint for the optimal pipeline service time. \( T_{tr} \) assumes the value of the dominant transfer time.

Similarly, the estimated completion time for the case where the kernel time dominates the communication times is given by

\[ T_{C} = T_{K} + \frac{T_{tr}}{n} \]

The above formula actually neglects the overhead associated with stream management and the overhead of multiple data transfers and kernel launches. Stream management has an associated overhead which linearly increases as the number of streams increases. Too much of it affects the overall performance of the application. Taking these observations into consideration, the following equations better estimate the completion time.

\[ T_{C} = \begin{cases} T_{tr} + \frac{T_{E}}{n} + T_{oh}, & \text{case 1 (} T_{tr} \text{ dominates } T_{E} \text{)} \\ T_{E} + \frac{T_{tr}}{n} + T_{oh}, & \text{case 2 (} T_{E} \text{ dominates } T_{tr} \text{)} \end{cases} \]

The overhead \( T_{oh} \) needs to be decomposed further; hence,

\[ T_{oh} = \begin{cases} nT_{tr\text{-oh}} + (n-1)T_{tr\text{-oh}}, & \text{case 1} \\ nT_{tr\text{-oh}} + T_{k\text{-oh}} & \text{case 2} \end{cases} \]

where \( T_{tr\text{-oh}} \) represents the overhead relative to the creation of a stream, \( T_{tr\text{-oh}} \) represents the overhead of the data transfer and \( T_{k\text{-oh}} \) represents the overhead relative to the kernel launch.

To calculate the optimal stream count, let us consider the above cases. In case where the dominating operation is the data transfer operation and for now if we consider it to be the host-to-device transfer we have:

\[ T_{C}(n) = T_{tr\text{-h2d}} + \frac{T_{E}}{n} + \frac{T_{tr\text{-d2h}}}{n} + T_{oh} \]

In this case, the kernel execution times and the device-to-host transfer times of all tasks except for the last one are overlapped by the host-to-device transfer times of the tasks, including the overheads due to multiple kernel invocation and multiple data transfers. In equation \( 10 \), \( T_{tr\text{-h2d}} \) and \( T_{tr\text{-d2h}} \) account for the kernel execution time on the last task and the device-to-host transfer time of the last output task, respectively. Expanding equation 10, we have:

\[ T_{C}(n) = T_{tr\text{-h2d}} + \frac{T_{tr\text{-h2d}}}{n} + nT_{tr\text{-oh}} + (n-1)T_{tr\text{-oh}} \]

where \( T_{tr\text{-h2d}} \) represents the transfer time of the input data in a synchronous version, \( T_{tr\text{-d2h}} \) represents the transfer time of the output data in a synchronous version, \( T_{tr} \) represents the kernel time in a synchronous version and eventually \( n \) is the number of CUDA streams used in the asynchronous version.

The coefficient \( n - 1 \) in equation 11 is to account for the additional overhead of sending the last \( n - 1 \) tasks. Equation 11 is a hyperbolic function and solving for the local minimum for the positive value of \( n \) gives the optimal stream count.

For the case where the device-to-host transfer time is greater than the host-to-device transfer time and the kernel execution time, we have to interchange the places of \( T_{tr\text{-h2d}} \) and \( T_{tr\text{-d2h}} \); thus,

\[ T_{C}(n) = T_{tr\text{-d2h}} + \frac{T_{tr\text{-h2d}}}{n} + nT_{tr\text{-oh}} + (n-1)T_{tr\text{-oh}} \]

Now consider the case where the kernel execution time dominates the input/output task transfer time; hence, we have:

\[ T_{C}(n) = T_{E} + \frac{T_{tr\text{-h2d}} + T_{tr\text{-d2h}}}{n} + nT_{tr\text{-oh}} + T_{k\text{-oh}} \]

Note that in equation 13 the kernel launch overhead is not multiplied by \( n \). At first this may seem counter intuitive but it is known that kernel launch overhead depends on the number of blocks. Let \( B \) be the number of blocks the kernel is launched with in the synchronous version and \( b \) be the launch overhead. With \( n \) streams, every launch is configured with \( B \) blocks and the kernel launch overhead becomes \( \frac{b}{n} \) for each launch; hence, the total overhead will be the sum of the individual overheads, i.e.,

\[ T_{k\text{-oh}} = \sum_{i=1}^{n} \frac{t}{n} = t \]

The following equations summarize all cases;

\[ T_{C}(n) = \begin{cases} T_{tr\text{-h2d}} + \frac{T_{tr\text{-d2h}}}{n} + nT_{tr\text{-oh}} + (n-1)T_{tr\text{-oh}}, & \text{case 1} \\ T_{tr\text{-h2d}} + \frac{T_{tr\text{-d2h}}}{n} + nT_{tr\text{-oh}} + (n-1)T_{tr\text{-oh}}, & \text{case 2} \end{cases} \]
where the first two cases refer to the situations where the transfer times dominate the kernel time and the third one refer to the case where the kernel time dominates the transfer times.

The overhead of kernel launch can be obtained by launching the empty kernel and timing the elapsed time. Stream creation time is available for each device architecture. One can evaluate the transfer time overhead by solving the equation.

\[ T_{tr-oh} = 2 \cdot T(M) - T(2M) \] (16)

where \( T(x) \) is the transfer time of \( x \) byte of data. It has to be noted here that the data transfer must be performed from a pinned host memory as this is the case when using CUDA streams. Large value of \( x \) gives a better estimation.

Equation 15 is a family of functions which can be generalized as

\[ f(n) = \frac{A}{n} + n \cdot B + C \] (17)

where \( A, B, \) and \( C \) are real numbers. This is a hyperbolic function, for the positive values of \( n \), the function decreases monotonically as \( n \) increases and starts to increase monotonically after some value of \( n \), as \( n \) increases. For our model, \( n \) denotes the number of streams and solving for \( n \) that results in the minimum value of \( f(n) \) gives us the optimal stream count.

5. Experiments

In order to verify the feasibility of our approach, we implemented different versions of a synthetic application using FastFlow [1]. FastFlow is a structured parallel programming framework that efficiently provides and supports—among others—primitive stream parallel patterns, including the pipeline and farm we discussed in Sec. 2 and targets shared memory multi core architectures.

A first version of the application implements the pipeline pattern (2) outlined in Sec. 3. In particular, the second stage wraps in a FastFlow ff_node all the activities needed to direct the data parallel task computations to the GPU. The highly efficient mechanisms sported by FastFlow relative to the inter-thread communication [3] and to the parallel thread orchestration guarantee that no measurable additional overhead is introduced to the overhead required to run the data parallel tasks on the GPU cores. This has been verified running the GPU kernel within the FastFlow stream parallel program and within a standard main, measuring the execution time of the GPU kernel in the two cases and verifying that the times were actually the same.

With this first version of the application we performed different experiments aimed at validating the performance model introduced in Sec. 4 by comparing the predicted completion times with the measured ones.

A second version of the application implements the pipeline pattern with a farm stage (1) outlined in Sec. 3. This version has been used to evaluate the effect of different application parameters on the speedup achieved using the CPU.

All the results discussed in this Sec. are relative to experiments performed on a 24 core AMD Magny Cours architecture with an nVidia Tesla C2050 GPU (CUDA capability 2.0), running CentOS Linux, kernel 2.6.18-274.el5 and using gcc 4.6.2 and CUDA SDK 5.5 and on a 16 core Intel Sandy Bridge architecture with a GeForce 660 GPU (CUDA capability 3.0), running CentOS Linux, kernel 2.6.32-279.el6 and using gcc 4.4.6 and CUDA SDK 5.5.

5.1 Validating the performance model (optimal stream number)

We measured the completion time in three different use cases, using the AMD/Tesla C2050 architecture. The three uses case applications all implement the pattern (2) outlined in Sec. 3 but differ in the kind of map implemented on the GPU in the second pipeline stage:

- The first use cases implemented uses a second stage computing on each input task—a vector of float—a simple map computation.
- The second one instead implemented a simple vector reduce computation.
- The third one implemented a string search algorithm.

In the first two cases—these are synthetic applications, indeed—the time spent in the function used (the map function \( f \) and the reduce binary function \( \oplus \)) has been artificially varied to stress the GPU without needing to include actual, complex kernel code.
The kind of results achieved are outlined in Fig. 3 and 4. The model succeeds predicting the “shape” of the completion time curve and fairly approximates the minimum actually measured with the experiments.

Fig. 5 shows quite a different case. In this case, the time spent sending data and receiving results is much less than the time spent computing the kernel on the GPU and therefore there is no advantage using the stream mechanisms to overlap communication and computations time. In fact, the model and the measured times clearly indicate that the “best” stream count is 1, which in turn means the “synchronous” version is actually better than the “asynchronous” one.

5.2 Evaluating impact of application parameters on application speedup

We used the second prototype to direct to the GPU very small data parallel kernels in a stream and we evaluated the kind of changes in the speedup due to the variation of different parameters such as the stream length, the amount of data parallelism in a single task or the amount of computation necessary to compute the single data parallel sub-task. In this Section, all the speedups are computed with respect to the execution of the computation on a single CPU core.

Fig. 6 plots the speedup relative to the execution of a variable number of data parallel tasks. Each data parallel task computes a map on a vector of 32 floating point items. The streams considered hosted 4K, 8K and 16K tasks. It is evident that longer streams achieve better speedups and succeed exploiting a larger number of streams. In fact, each one of the workers in the pipeline\((S_1, \text{farm}_{n_w}(\text{map}(f)), S_3)\) farm use own stream to direct tasks to the GPU.

The kind of effect achieved on the GPU is the one shown in Fig. 7. The Fig. is a snapshot of the \texttt{nvvp} profiler running a 4 worker farm (i.e. with 4 CUDA streams directing tasks to the GPU) and clearly shows how the execution of kernels from different workers is actually performed concurrently.

Fig. 8 shows what happens when the stream tasks have different “map” sizes. We considered streams of 8K tasks in all cases, but in the first case we have 16 items per vector—less than the CPU warp size—in the second we have 32 items per vector and in the third one 64 items. In all cases, the “optimal” speedup is achieved in between 2 and 4 workers/stream. This seems to indicate that the fairly different “map” sized do not impact the optimal stream count, while guaranteeing different speedups.

Eventually, Fig. 9 shows what happens when different amounts of time are required to process the single item in a map. The small, medium, coarse grain in the legend refer to computations taking time \(t\), \(2t\) and \(4t\) time, respectively. By increasing the time spent in
the computation of the single map sub-item, the speedup increases then lowers again. This indicates that there is a kind of “optimal” grain size for computations with respect to the speedup achieved.

6. Related work

nVidia documentation suggests the correct way to direct multiple data parallel tasks to the GPU in such a way concurrent execution of the data parallel tasks is actually achieved on GPUs with computing capability at least 2.0 [14, 15]. This is actually the mechanism we exploited to implement concurrent execution of tasks appearing onto the input stream. Different structured parallel programming environments support data parallel pattern implementations targeting GPUs, including Skepu [8], Meusli [9] and SkeCL [16]. However, to be best of our knowledge no one of these parallel programming frameworks supports the implementation of task parallel patterns on GPUs.

Marrow, is a new structured parallel programming framework based on algorithmic skeletons and targeting GPUs. In addition to traditional—for GPUs—skletons, it adds pipeline, stream and loop skeletons and exploits GPUs through OpenCL. From the documentation available, it seems that Marrow actually succeeds overlapping at least the communication and computation phases of a stream parallel computation [11].

Spoc is a skeleton library written in Ocaml that also addresses the execution of data and stream parallel skeletons on GPUs. The library and run time perform a lot of optimizations, especially relative to the data transfers to and from GPU memory, but as far as we know the library does not support parallelism in the execution of tasks appearing on the input stream [6].

7. Conclusions and ongoing work

We investigated the problem of executing stream parallel computations on GPUs, which are usually exploited with data parallel computations. We provided a performance model suitable to predict the optimal number of CUDA streams to be used to implement stream parallel computations on state-of-the-art GPUs. We discussed experiments that a) assess our performance models and b) demonstrate how speedup is affected by the different parameters in a stream parallel computation of small data parallel tasks (i.e. tasks not using the whole amount of resources of the GPU).

We are currently performing more experiments assessing the feasibility of directing stream parallel computations to GPUs to improve the performances of applications with respect to the execution on CPU cores. In particular, we are experimenting “clustering” of stream parallel tasks into data parallel tasks and performing more experiments aimed at optimizing the streaming of tasks to GPUs.

References

Performance Enhancement Infrastructure for Skeleton-based Parallel Programming Frameworks

Mehdi Goli  Michael T. Garba  John McCall
IDEAS Research Institute, Robert Gordon University, Aberdeen, Scotland, UK
Email: {m.goli,m.t.garba,j.mccall}@rgu.ac.uk

Horacio González–Vélez
Cloud Competency Centre, National College of Ireland, Dublin, Ireland
Email: horacio@ncirl.ie

Abstract
Algorithmic skeletons—or simply skeletons—have long been considered a viable approach to introduce high-level abstraction to parallel programming that hides the complexity of recurring patterns of coordination and communication logic behind generic reusable application interfaces. However, existing skeleton frameworks do not incorporate clean interfaces to provide information and expose control parameters that may be required by performance optimizers for analysis, reporting, and scheduling. In this paper, we propose a Performance Enhancement Infrastructure (PEI) exploiting a language-neutral protocol that facilitates the development of performance measurement and enhancement tools while allowing for portability between protocol-compliant frameworks. We compare the performance of FastFlow with and without PEI, demonstrating that instrumenting FastFlow with PEI can achieve speedups of up to 260% for heterogeneous applications and up to 200% for homogeneous applications.

Keywords: Performance Enhancement Infrastructure; Heterogeneous Skeletons; GPU; Algorithmic Skeletons; Parallel Patterns; Parallel computing;

1. Introduction
Efficient heterogeneous parallel computing with CPUs and GPUs remains an open area of study due to the intrinsic complexity and potential trade-offs associated with the communication and non-linear performance characteristics that exist in GPU-accelerated nodes despite the presence of high performance interconnects. Current trends indicate that, with major hybrid architectures in development, techniques and tools for managing these heterogeneous platforms are of increasing significance.

Algorithmic skeletons—or simply skeletons—have long been considered a viable approach to introduce high-level abstraction to parallel programming that hides the complexity of recurring patterns of coordination and communication logic behind a generic reusable application interface [12].

Algorithmic skeletons allow the separation of computational from coordination concerns. The execution of a skeleton-based application involves the orchestrated execution of a set of components implementing the parallel pattern (or the parallel pattern composition) modelling parallelism within the application. From the functionality point of view such components include:

- “Computing” components: aimed at executing the original business code encapsulated into the sequential wrappers to serve as pipeline stage, farm, or map worker, etc.
- “Service” components: are those orchestrating the activities of the computing components and routing necessary input data to these components and collecting the result data from them (e.g. farm and map emitters and collectors).
- “Homo”-components: are computing components which have similar functionality but are running on different device type in shared memory system (e.g. an OpenCL based component can be compiled both as a GPU kernel and as a CPU thread).

Skeleton-based frameworks retain access to information about parallel patterns that is typically rendered unavailable by the compilation process [20]. The presence of this high-level information makes it practical to create integrated tools for application configuration, profiling, reporting, refactoring, scheduling, and dynamic load-balancing.

In this paper, we propose a Performance Enhancement Infrastructure (PEI) for skeleton frameworks based on integrated component instrumentation and an open communication protocol. This can potentially enable the achievement of improved performance by dynamic coordination mechanisms integrated with the framework. Moreover, it can improve the portability of these tools for platforms with different performance profiles. By testing with the FastFlow framework, we demonstrate the extraction of structural patterns, runtime profiling information and the use of scheduling and load-balancing policies determined by external tools that analyse this data.

Crucially, these instrumentation mechanisms must meet the requirement of low performance overhead. However, other desirable characteristics that would increase the effectiveness of this approach include language and framework neutrality, and seamless integration with external tools.

This paper is organised as follows. In section 2, we briefly describe the related approaches. In section 3 we explain the proposed performance infrastructure in details. Section 4 presents the experimental infrastructure used for evaluating the results, followed by the result of our evaluation. Finally, section 6 provides some concluding remarks and future work.

2. Background
As the subject of extensive research, several tools are currently available for the computational problem including automatic par-
allelisation, performance visualisation, instrumentation and debugging as summarised in [9, 10]. One trend is to generate language-based tools for auto-tuning parallel applications using a static parsing mechanism to analyse the program source code, determine the tunable parameters, such as number of threads and identify parameter dependencies using `pragma`-based approaches [14, 23] or the generation of entirely new programs [11, 15]. Typically, these approaches are limited to a single language or are highly specialised for a certain domain [21, 22].

An alternative problem is improving performance by optimising coordination. This may be approached with solutions based on runtime information monitoring and reconfiguration. In this paper, we focus on the coordination problem for achieving application performance objectives and meeting non-functional requirements. In [2] dynamic reconfiguration of grid-aware applications in the ASSIST programming environment has been proposed to capture and meet the quality of service requirements in Grid applications.

A framework for programming self-managing component-based distributed applications (self-* ) is presented in [1]. It is organized as a network of management elements (MEs) interacting through events. Methods are categorised as sensors and actuators. The sensors determine environment changes through events generated by the management platform or by other application specific sensors. The actuators apply architectural changes, add, remove and reconfigure components and bindings between them via MEs.

The absence of clean separation of coordination from computation in these approaches requires the direct involvement of programmers in the performance tuning that should exist outside their program. However, it is highly desirable for optimisation of these coordination mechanisms, or non-functional concerns [3], to be handled transparently from the user perspective. Skeleton based frameworks provide an opportunity to achieve this clean separation.

In [3–5] a behavioural optimisation technique captures non-functional concerns in an independent activity or the autonomic manager. These concerns are:

- parallelism degree;
- set-up and tuning;
- dynamic load balancing; and,
- adaptation of parallelism patterns to different features of the target architecture.

Limitations of these parameters are specified by user applications in the form of a contract. Each component of the skeleton is equipped with its own autonomic manager that executes a control loop in one of two modes: passive or active. In the passive mode a manager monitors the status of the current computation and awaits new contracts from its parents. Contracts are received in the active mode from the user application or a parent manager in the hierarchy and the appropriate autonomic actions are taken as required to maintain the contract. Semi-formal models based on autonomic management are proposed for component based parallel and distributed program development [6]. This programmer-oriented methodology is based on formal tools that permit reasoning about program and refinement.

Adaptive Structured Parallelism [19] (ASPARA) is a generic methodology to incorporate structural information at compilation into a parallel program, which facilitates adaptation at runtime. The four phases are: `programming` during which API calls to ASPARA are parameterised, `compilation` when the structured parallel program is instrumented, `calibration` which extrapolates node performance and selects the most appropriate for the given application, and `execution` which is responsible for selection and deployment on the chosen node.

Whilst these solutions have been developed in the context of CPU environments, they lack full support to multi-core CPUs and general-purpose computation on graphics processing units (GPGPU). In fact, higher-level approaches for performance tuning of heterogeneous multicore CPU/GPU applications have become an area of active research in computational science.

A queue monitoring heuristic is introduced in [16] for increasing resource utilisation for divisible workloads performing numerical linear algebra on CPU and GPU resources that fit the pipeline parallel architectural pattern. Stochastic allocation of heterogeneous resources to pipeline components is proposed that maximises throughput subject to queue stability. Following this work, a heterogeneous streaming pipeline implementation of these numerical linear algebra kernels over the FastFlow skeletal library [7] introduces adaptive throttling based on memory usage to coordinate the streaming pipeline and dynamically allocate CPU and multi-GPU resources in a distributed memory cluster environment [18].

The novelty of our approach lies in further analysing, visualising, and optimising the coordination mechanism of the Heterogeneous(CPU/GPU) skeleton-based frameworks, transparently from the user applications. Providing a new component interface protocol for interaction between the skeleton-based frameworks and the proposed infrastructure increase the infrastructure portability and generality of the design. This will allows us to instrument any arbitrary skeleton frameworks running on different heterogeneous platforms with the proposed infrastructure as long as they can speak via the protocol.

3. Proposed Approach

Our performance framework provides an interface to external tools through a Dynamic Skeleton Runtime Interface (DSRI) that is able to communicate via a standard Virtualisation Interaction Protocol (VIP). Figure 1 shows an architectural overview when applied to a general framework. The following sections detail these components.

![Figure 1. The Architecture of Performance Enhancement Infrastructure.](image)

3.1 VIP: Virtualisation Interaction Protocol

VIP provides a standard protocol for interaction among all parts of an instrumented skeleton application. The protocol covers four classes of interaction:

1. **Structural Information** that captures the high-level parallel patterns.
2. **Behavioural Information** relating to performance parameters of the skeleton components.

3. **Instructional Information** that provides an interface to set or modify control parameters.

4. **Architectural Information** relating to the underlying hardware execution platform.

VIP uses Javascript Object Notation (JSON) [13] as the low-level encoding format. JSON’s conciseness, broad language support, convenient API, and human-readability has led to its widespread adoption as a communication format in distributed systems and the World Wide Web.

### 3.2 DSRI: Dynamic Skeleton Runtime Interface

Using VIP, the DSRI serves as a bridge between performance enhancement tools and an executing application, providing access to structural and behavioural information after retrieval from the instrumented components and forwarding instructions from external controllers to these components. We refer to the information extracted from a running application as **sensors** and the control parameters as **actuators**. The interaction with each framework is via the two types of methods called actuator and sensor. These methods are interfaces which must be implemented by the framework.

Detailed explanation of the actuator and the sensor are as follows.

**Actuators**

Actuators are parameters that affect non-functional behaviours and configuration choices such as the mapping or allocation of components to specific resources (CPUs and GPUs), the component execution states (online or offline) or the load balancing policy governing workload distribution among alternative components. These may optionally be set statically at program initiation or dynamically at runtime.

**Sensors**

Sensors provide information gathered during the execution of the application. This information can be categorised as:

1. **Structural Metadata** that describes the structure of the skeleton tree and component features including the component name and the pattern type.

2. **Profiling Information** that captures service time frequency distribution, utilisation, throughput, and the status of the components.

To mitigate the performance overhead, two levels of information monitoring are necessary:

**Aggressive Sampling Mode:** Comprehensive information is gathered on every individual task completion by a component. Aggressive monitoring introduces some performance overhead and is intended for initial data gathering in the absence of prior data or after phase change events that alter the runtime characteristics of a program.

**Sparse Sampling Mode:** Statistical sampling is performed at a specified sampling rate for only a certain fraction of completed tasks. This is intended to be the default mode.

### 3.3 Performance Enhancement Tools

The performance enhancement tools include analysis, visualisation, source refactoring and pattern transformation utilities for sensor information, and heuristic controllers that control the actuator states for optimising performance metrics. The DSRI may incorporate these controllers or act as a proxy for VIP- compliant external programs that provide these capabilities.

In the following we outline the currently available heuristic controller tools for PEI.

#### 3.3.1 Heuristic controllers tools

Given a parallel skeleton-based application, when parallelism may exploit arbitrarily nested parallel patterns over Heterogeneous(CPU/GPU) architectures, the aim is:

- To find an optimised allocation of skeleton components to underlying available heterogeneous resources

and

- To tune a flow of independent units of tasks in a divisible workload across multiple computing components

in order to improve the performance with respect to optimising the throughput, energy consumption, and resource utilisation.

The problem is addressed by providing a set of performance enhancement tools using heuristics that consider the different alternative controlling mechanisms. These tools evaluate the applications with respect to the following: the expected performances as derived from the performance models associated to the applied parallel patterns and monitoring of service time of the different components of the application at the runtime.

While considering the above problem, different aspects influencing the execution time of a parallel skeleton-based application must to be taken into account. In the following, we will outline these aspects used as base policies for the provided controlling strategies.

**Adaptive workload distribution policies:** When homo-components are available in a skeleton tree, a key challenge is to spread the computation among CPU and GPU cores using those components such that both resources contribute to the computation of the final result and the CPU and the GPU allocated sub-computations terminate as simultaneously as possible. This will reduce any inefficiency caused by a component waiting for the other computations to complete.

In such cases structured computations may be automatically managed by means of heuristics that exploit the models to split the available (data parallel) and the GPU cores (compiled as GPU kernels). In order to satisfy this policy an embedded tool, called DLT, is introduced aiming to present an optimised solution for workload distribution. It uses the divisible load theory technique [8] to provide a fraction of input data in proportion with the effective service time of each component.

**Components allocation policies:** Modern multi-core architectures specially with provide the possibility of executing multiple threads per core with an almost negligible drop in performance in comparison with executing them on separate cores.

The drop in performance occurs as a result of switching the register set, integer ALUs and memory access components in different contexts. However, the more expensive floating point hardware is shared among different threads. Generally, computing components require floating point resources while service components which require more replicated per context resources. Given that there are enough number of cores, in order to maximise the floating point resources for each computing component the two following heuristics can be considered.

---

1. However, this depends on the provided program features.
I. Allocating computing components to dedicated cores.  

II. Allocating service components to different contexts of the same core.

Therefore, a heuristic tool, called Mapper, embedded within the DSRI node applies the component allocation policies when applicable. However, there are cases where the above conditions are not met, especially given that an application can exploit the arbitrary nested level of parallel pattern, the number of components can easily surpass the number of cores. In these cases, the mapper tries to allocate the CPU/GPU component into available devices where all processors receive a fair amount of workload based on components effective service time.

Energy consumption policies: The key challenge here is to obtain maximum utilisation of the resources for any CPU slot allocated to a component. The optimised usage of the allocated slot of each component depends on the availability of a task on the component queue. If the task is not available the component must wait for it. This waiting can be either a busy wait loop or a thread sleeping until the data becomes available. In the former case, specially when the number of the threads running the application is less than or equal to the number of available cores, the performance of the application does not drop. However, when the number of threads running the application is more than the number of cores, the sleeping mechanism can dedicate this time slot to those threads which have received their tasks but are waiting for resources to execute them. Moreover, the energy consumption of the former case is higher and less optimised than the latter, which is also against the optimal resource utilisation.

Therefore, the strategy here is to provide an extra information layer on top of the operating system scheduler by putting a thread which is executing a component to sleep, whenever the component input/output queue is empty/full. Applying such a strategy on frameworks supporting busy waiting mechanism can be made on demand and is useful when the number of threads is bigger than the number of available cores and also energy saving and optimal utilisation are of high priority.

4. Case-Study

While the proposed approach is intended to be generally applicable to different skeleton frameworks, we apply the performance enhancement infrastructure specifically to the FastFlow framework, a parallel programming framework for multi-core platforms which facilitates the development of shared memory parallel applications by providing abstraction layers, programming constructs, and composable algorithmic skeletons [7].

FastFlow

FastFlow streaming patterns are coordinating mechanisms that control the flow of work between multiple concurrent threads. Such flow control allows programmers to focus on the application-specific computations by efficiently abstracting the complex coordination and communication layers. It provides multiple variants of three pattern categories: farms, pipelines and maps that may be composed and arbitrarily nested to realise complex streaming topologies.

FastFlow default controlling strategy FastFlow provides a very simple and effective interface to pin threads onto available cores. Threads are numbered according to the order used to activate the corresponding components in the skeleton-based application. However, either the user must specify a vector of core identifiers \( v \) (as supported by the target hardware) for FastFlow to automatically pin thread \( ii \) to core \( v[i] \), or the threads supporting the execution of the program components are mapped by applying the round-robin technique to the existing cores. When dealing with arbitrary nested parallel pattern where the number of available cores in shared memory is less than the number of components, assigning threads to cores in a fair manner can be complicated. Moreover, FastFlow uses the busy waiting loop technique when the component queue is empty, which can affect the performance in such cases while having unoptimised resource utilisation and high energy consumption.

Moreover, GPU offloading is implemented by taking advantage of different kinds of GPU software tools, in particular OpenCL. When a given component is to be offloaded to GPU, the corresponding “kernel” is compiled according to the appropriate policies with respect to the OpenCL environment and offloaded through the mechanisms offered by the GPU programming environment [17]. The FastFlow skeleton-based programming environment supports active components written in OpenCL, which may be compiled to be executed on either CPU cores or GPUs. However, for farm/map patterns, unless the user provides its own scheduling policy, the default policy will allocate a fair amount of workload to each component without considering the component execution time for each unit of computation (component processing power). However, applying the adaptive workload distribution policies presented in section 3 to calculate the appropriate computation unit for each component based on its processing power, is not a trivial job, specially when dealing with arbitrarily nested level of parallel patterns containing different number of homo-components in each level.

Improving FastFlow default controlling strategy Instrumenting FastFlow with PEI automatises the above procedures and improves the default controlling strategies by using policies suggested in section 3. These policies include

I. Improving the CPU core allocations of different components exploiting the knowledge available w.r.t. skeleton-based patterns used in FastFlow. For example, the mapper tool can be applied to prevent mapping of two workers to the same multi-threaded core (to avoid competing for the non replicated floating point resources) when enough number of cores are available, or to map the emitter and the collector of a farm/map pattern and the first stage of a pipeline when it is service component, to different contexts of the same core.

II. Spreading fair amount of computation unit to each component in proportion with their service time for heterogeneous(CPU/GPU) farm/map pattern, by applying the DLT tools.

III. Applying the energy saving policies when the number of component are more than the number of cores which also optimises the energy consumption level.

Instrumenting FastFlow with PEI

FastFlow is designed around a uniform object oriented class hierarchy with components (which may be pipeline stages or farm workers) and skeleton constructs (pipelines, farms and maps) implemented as subclasses of the `ff_node` base class.

1. DSRI: We have introduced a VIP-compliant DSRI manager node in FastFlow that connects with the root of a skeleton tree and retrieves or passes structured data. This DSRI node executes in a separate thread and uses non-blocking locks during information retrieval to maintain the lock-free semantics of FastFlow - a central aspect of the architecture.
2. **ff_node** Instrumentation: This introduces methods to allow all subclasses to interact with the DSRI node and expose sensor and actuator interfaces.

3. Instrumented Coordination Patterns: The pipeline, farm and map patterns are updated to expose their own sensor and actuator interfaces as well as those of their child components. This enables recursive extraction and forwarding of sensor states and actuator controls from the root of the skeleton tree by the DSRI manager node in a single operation using a corresponding tree data structure.

4. Instrumented Structural Subcomponents: **ff_loadbalancer** and **ff_gatherer** are specialised structural components that perform work distribution, load balancing and recombination in FastFlow farms. These subcomponents may be subclassed in user applications.

   FastFlow incorporates tracing functions that collect output performance information at program termination. These performance counters are output as unstructured textual information. By exposing these performance counters, previously output in the form of unstructured textual information, as sensors, the DSRI node retains comprehensive information about the executing program.

   The available actuators include CPU/GPU component mappings and load balancer policies that previously required source code intervention to modify. An example of the implementation of actuator and sensor for FastFlow farm pattern, and a sample farm example instrumented with PEI, with two JSON formatted instructional and profiling VIP based informations are provided in appendices.

5. **Evaluation**

   Figure 2 is a visualisation of the component interconnection topology for Heterogeneous Convolution problem extracted from the runtime structural metadata (left) and instructional information generated from the extracted runtime profiling data (right). To evaluate our approach for sensor and actuator parameters, we compare the performance of FastFlow with and without a VIP-compliant controller tools that determines the DLT tool for load balancing, the mapper tools for resource allocation/mapping and the energy consumption optimiser tool. These controllers take structured information collected by the DSRI node on previous program runs and predicts the optimal parameters for the execution of the program.

5.1 Hardware/Software Specification and Benchmark Applications

Performance evaluations were carried out on tree hardware platforms. Tables 1, 2 and 3 provide the detailed specifications for each test machine. Eight different FastFlow benchmark applications were evaluated with parameters described in Tables 6, 5 and 4. The first column explains the name of the benchmark application, the second one determined the skeleton tree pattern used for designing the application. The third column explains the number of input tasks executed by each application, and the fourth column represents the number of workers generated for each farm in the skeleton tree pattern. The Xookik cluster and Titanic are large machines and are tested with correspondingly large inputs. The Lenovo X220 laptop is a relatively resource-constrained platform but is representative of low-end user platforms where performance is nonetheless significant.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>Intel(R) Core(TM) i5-2520M</td>
</tr>
<tr>
<td>No. of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Cores per CPU</td>
<td>4</td>
</tr>
<tr>
<td>CPU Clock</td>
<td>800 MHz</td>
</tr>
<tr>
<td>physical Memory</td>
<td>4 GB</td>
</tr>
<tr>
<td>OpenCL driver</td>
<td>AMD-SDK version2.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>AMD Opteron(tm) Processor 6176</td>
</tr>
<tr>
<td>No. of CPUs</td>
<td>2</td>
</tr>
<tr>
<td>Cores per CPU</td>
<td>12</td>
</tr>
<tr>
<td>CPU Clock</td>
<td>3.37 GHz</td>
</tr>
<tr>
<td>physical Memory</td>
<td>32 GB</td>
</tr>
<tr>
<td>No. of GPUs</td>
<td>1</td>
</tr>
<tr>
<td>GPU Model</td>
<td>NVIDIA Tesla C2050</td>
</tr>
<tr>
<td>GPU Memory</td>
<td>2.68 GB</td>
</tr>
<tr>
<td>GPU Cores</td>
<td>448</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>4.0 V0.2.1231</td>
</tr>
<tr>
<td>GPU Driver Version</td>
<td>290.10</td>
</tr>
<tr>
<td>OpenCL driver</td>
<td>AMD-SDK version2.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>Intel(R) Xeon(R)</td>
</tr>
<tr>
<td>No. of CPUs</td>
<td>2</td>
</tr>
<tr>
<td>Cores per CPU</td>
<td>6</td>
</tr>
<tr>
<td>CPU Clock</td>
<td>3.07 GHz</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>50 GB</td>
</tr>
<tr>
<td>No. of GPUs</td>
<td>1</td>
</tr>
<tr>
<td>GPU Model</td>
<td>NVIDIA Tesla M2090</td>
</tr>
<tr>
<td>GPU Memory</td>
<td>6 GB</td>
</tr>
<tr>
<td>GPU Cores</td>
<td>512</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>5.0 V0.2.1231</td>
</tr>
<tr>
<td>GPU Driver Version</td>
<td>304.54</td>
</tr>
</tbody>
</table>

5.2 Results

5.2.1 PEI Overhead

One of the main concerns for adding any extra tools to an existing framework is the overhead created by adding the tools. Since all performance enhancement tools analysing the data and providing actuator instruction are external, the expensive part in PEI is the extraction of information from the framework while it is running. In order to measure the overhead of executing the sensor function, we switched off all provided performance enhancement tools and ran the instrumented framework (for both aggressive and sparse sampling). Therefore, in this case, the execution time of the instrumented version is equal to the time spent for running the original FastFlow plus the time spent for extracting the sensor data. Comparing these times with the original Fastflow framework execution times for different benchmark applications, will indicate the PEI overhead. Figure 3 compares the execution time of the original FastFlow framework and the instrumented framework without any enhancement tools (for both aggressive and sparse sampling) on the Xookik cluster. For the quicksort and convolution programs where the input stream sizes are of the order of 10 million, the
Figure 2. Component interconnection graph for Heterogeneous Convolution (left) and its instructional information provided for Xookik Cluster (right).

Table 4. Software Specification For Xookik CLUSTER NODE.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Skeleton Tree</th>
<th>input stream</th>
<th>No. of Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATMUL</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>100 matrices of 512 \times 512</td>
<td>n=16</td>
</tr>
<tr>
<td>SIMPLE_MANDELBROT</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>resolution matrices of 1024 \times 1024 for 25,000 iteration</td>
<td>n=12</td>
</tr>
<tr>
<td>QUICKSORT</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>Array size of 50000000</td>
<td>n=12</td>
</tr>
<tr>
<td>FIBONACCI</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>\text{fib}(40)</td>
<td>n=12</td>
</tr>
<tr>
<td>NQUEEN</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>Board size of 18 \times 18</td>
<td>n=12</td>
</tr>
<tr>
<td>STENCIL</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>Matrice size of 16384 \times 16384</td>
<td>n=12</td>
</tr>
<tr>
<td>HETERO_CONVOLUTION</td>
<td>PIPELINE(STAGE1, FARM(Emitter,FARM(Emitter, Worker[n], Collector),OPENCL WORKER[m],Collector))</td>
<td>100 image matrices of 8192 \times 8192 and filter matrices of 8 \times 8</td>
<td>n=8, m=2</td>
</tr>
<tr>
<td>HOMO_CONVOLUTION</td>
<td>PIPELINE(STAGE1, FARM(Emitter, Worker[n], Collector))</td>
<td>100 input matrices of 8192 \times 8192 and filter matrices of 8 \times 8</td>
<td>n=12</td>
</tr>
</tbody>
</table>

Table 5. Software Specification For Titanic Machine.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Skeleton Tree</th>
<th>input stream</th>
<th>No. of Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>HETERO_CONVOLUTION</td>
<td>PIPELINE(STAGE1, FARM(Emitter,(FARM(Emitter, Worker[n], Collector),OPENCL WORKER[m]),Collector))</td>
<td>100 image matrices of 8192 \times 8192 and filter matrices of 8 \times 8</td>
<td>n=16, m=2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Skeleton Tree</th>
<th>input stream</th>
<th>No. of Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATMUL</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>40 matrices of 256*256</td>
<td>n=6</td>
</tr>
<tr>
<td>SIMPLE_MANDELBROT</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>resolution matrices of 1024*1024 for 8000 iteration</td>
<td>n=4</td>
</tr>
<tr>
<td>QUICKSORT</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>Array size of 500000</td>
<td>n=4</td>
</tr>
<tr>
<td>FIBONACCI</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>Matrix size of 2048*2048</td>
<td>n=4</td>
</tr>
<tr>
<td>NQUEEN</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>Board size of 16x16</td>
<td>n=4</td>
</tr>
<tr>
<td>STENCIL</td>
<td>FARM(Emitter, Worker[n], Collector)</td>
<td>Filter size matrices of 8x8</td>
<td>n=4</td>
</tr>
<tr>
<td>HOMO_CONVOLUTION</td>
<td>PIPELINE(STAGE1, FARM(Emitter, Worker[n], Collector))</td>
<td>100 matrices of 2048*2048 filter size matrices of 8x8</td>
<td>n=4</td>
</tr>
</tbody>
</table>

Figure 3. The upper-bound overhead of running the infrastructure over FastFlow framework on Xookik Cluster Node.

Figure 4. Performance evaluation of executing the convolution application with heterogeneous skeleton over different platforms. The worst case overhead is less than 3%. For most cases, the performance drop is less than 1%.

5.2.2 PEI Performance Enhancement Tools

The PEI performance enhancement tools are applied to two categories of FastFlow applications. These may be *Homogeneous applications* using multicore CPUs exclusively (Figures 5 and 6) or *Heterogeneous applications* that combine CPUs and GPUs (Figure 4).

Figure 5. Performance evaluation of running applications with homogeneous skeleton over Xookik Cluster Node.

Figure 6. Performance evaluation of running applications with homogeneous skeleton over IBM-Lenovo X220.
Figure 4 represents the performance evaluation of executing the convolution application with heterogeneous skeleton over different platforms. The heterogeneous FastFlow convolution problem introduced in [17]. In Titanic machine the OPENCL_Worker0 runs on GPU environment and the OPENCL_Worker1 runs on GPU environment, while in Xookik cluster both OpenCL workers runs on GPU environment and in IBM-Lenovo X220, both OpenCL workers run on CPU environment. This figure states the service time of the glob_farm workers and total execution time of convolution application with/without using the infrastructure.

The DLT technique, determines appropriate workload fractions for each component worker of the heterogeneous farm pattern in proportion to its relative service time and minimises resource idling. These service times are derived from historical data collected over previous program warm-up runs. A warm-up run for a program is an execution of the program for very small input stream size in order to acquire the necessary sensor information.

Figure 4 demonstrates that applying the adaptive workload distribution policies introduce in 3 the convolution application can achieve up to 260% speed up over an already optimised version which uses the heterogeneous skeleton pattern.

The traditional round-robin load balancer performs efficiently when there are not any homo-components in the applications. However, as figures 5 and 6 demonstrate, applying the component allocation and energy consumption policies can achieve up to 200% speed up over FastFlow default mapping policies using round-robin mapping policies and busy waiting loop technique.

6. Conclusions and Future Work

In this paper we have proposed an approach for integrating performance enhancement tools with skeleton frameworks and evaluated the suitability of the proposed infrastructure for instrumenting the FastFlow framework. By introducing the language-neutral VIP communication protocol (VIP), we contend that this approach may be applied to a broader range of skeleton frameworks. A consistent protocol would allow these frameworks to take advantage of common tools for analysis, runtime optimisation and control.

Our results demonstrate that it is possible to perform tuning and coordination through external tools that aim to improve performance measures such as application throughput and resource utilisation. Additionally, higher-level tools (e.g. for refactoring) can now access structural and profiling information for internal analysis and prediction or presentation to a human user.

The availability of these tools and a clean interface facilitates portability, by transparent tuning of the non-functional concerns to new target architectures. Further work will explore the extension of these static control mechanisms to dynamic environments, allowing online controllers to adaptively adjust execution parameters and reconfigure the actuators to suit changing performance profiles. Extensions to distributed memory environments and multi-tenant applications sharing the same execution resources are a logical next step.

A. Code Excerpts

A.1 actuator Implementation for farm

```cpp
virtual void actuator (mObject instObj) {
    mObject lbObj; 
    mObject gtObj; 
    mValue mv = find_value (instObj, "Emitter")
    if (!(mv.is_null()) ) { lbObj= mv.get_obj ( ) ;}
    lb->actuator (lbObj); 
    for (int i=0; i<nworkers; i++) {
        mv = find_component ( find_value (instObj, "Workers"), "Local_ID", i);
        mObject wobj;
        if ( !(mv.is_null()) ) { wobj = mv.get_obj ( ) ;}
        workers[i]->actuator (wobj); 
        gt = get_attribute (gtObj, "Collector");
        if ( !(gt.is_null()) ) { gtObj= gt.get_obj ( ) ;}
        gt->actuator (gtObj);
    }
}
```

A.2 sensor Implementation for farm

```cpp
virtual mObject sensor () {
    mObject fObj; 
    mArray farm_workers; 
    mObject lbObj; 
    mObject gtObj;
    lbObj= lb->sensor ( ) ;
    gtObj= get->sensor ( ) ;
    for (int i=0; i<nworkers; i++) {
        farm_workers.push_back (workers[i]->sensor ( ) );
    }
    int status;
    fObj["FF::Node_Subclass"] = abi::
    .cxa_demangle ( typeid (*this).name ( ) , 0,0,&status );
    fObj["Type"] = "FF::FARM";
    fObj["Local_ID"] = ff_node::get_my_id ( ) ;
    fObj["Emitter"] = lbObj;
    fObj["Workers"] =farm_workers;
    fObj["Collector"] = gtObj;
    return fObj;
```

A.3 Simple FastFlow farm application instrumented with PEI

```cpp
int main(int argc, char * argv[]) {
    ff_farm<adaptive_loadbalancer> farm; // farm object with DLT based load balancer
    Emitter E (streamlen); 
    farm.add_emitter (&E); 
    std :: vector<ff_node *> w; 
    for (int i=0; i<nworkers; i++) w.push_back (new Worker ());
    farm.add_workers (w); // add all workers to the farm
    Collector C;
    farm.add_collector (&C);
    PROFILE (&farm); // Instrumenting the application with PEI
    farm.run_and_wait_end ( ) ;
}
```

A.4 Json sparse profiling information provided by sensor for a simple farm with 2 worker

```json
"Collector": {
    "Assigned_Processor": 1,
    "Data_Size": 0,
```
A.5 JSON instructional information provided by actuator for a simple farm with 2 worker

```json
{
  "Collector": {
    "Assigned_Processor": 1,
    "FF::Node_Subclass": "ff::ff.gatherer",
    "Type": "FF::GATHERER"
  },
  "Emitter": {
    "Assigned_Processor": 0,
    "FF::Node_Subclass": "adaptive_loadbalancer",
    "Type": "FF::LOADBALANCER"
  },
  "FF::Node_Subclass": "ff::ff.farm<adaptive_loadbalancer, ff::ff.gatherer>",
  "Local_ID": -1,
  "Type": "FF::FARM",
  "Workers": [
    {
      "Assigned_Processor": 2,
      "Data_Size": 0,
      "FF::Node_Subclass": "Worker",
      "Local_ID": 0,
      "Maximum_SVC_Tick": 245,
      "Minimum_SVC_Tick": 50,
      "Pop_Delay_Count": 123,
      "Pop_Delay_Ticks": 123000,
      "Push_Delay_Count": 0,
      "Push_Delay_Ticks": 0,
      "Sampled_SVC_Tick_Distribution": [
        "Sampling_Rate": 0,
        "Total_Number_Of_Tasks": 5,
        "Total_SVC_Ticks": 500,
        "Type": "FF::NODE"
      ]
    },
    {
      "Assigned_Processor": 3,
      "Data_Size": 0,
      "FF::Node_Subclass": "Worker",
      "Local_ID": 1,
      "Maximum_SVC_Tick": 410,
      "Minimum_SVC_Tick": 65,
      "Pop_Delay_Count": 121,
      "Pop_Delay_Ticks": 121000,
      "Push_Delay_Count": 0,
      "Push_Delay_Ticks": 0,
      "Sampled_SVC_Tick_Distribution": [
        "Sampling_Rate": 0,
        "Total_Number_Of_Tasks": 5,
        "Total_SVC_Ticks": 750,
        "Type": "FF::NODE"
      ]
    }
  ]
}
```

Acknowledgments

This work has been funded by the European Commission FP7 through the project Paraphrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, under contract no.: 288570 (10/2011-9/2014). [http://paraphrase-ict.eu](http://paraphrase-ict.eu)

References


and Experience with pipelines and farms. Distributed heterogeneous architectures: A methodological approach


VMOpenCL: Enabling Resource Sharing of Heterogeneous Computing with OpenCL on VMware

Shaobo Luo, Yajun Ha, Akash Kumar, Qiang Wu, Mohammad Shihabul Haque, Heng Yu
National University of Singapore, Singapore
{shaobo.luo, elehy, akash, elewuqia, elemsh, eleyh}@nus.edu.sg

Abstract
Virtualization is a well-established concept for hardware resource sharing in cloud computing platforms. With the rapidly growing use of data intensive computations, interest in virtualizing of heterogeneous platforms including CPU, GPU and FPGA is also rising. However, unlike a CPU based homogeneous platform, heterogeneous platforms bring new challenges for virtualizing resources effectively. Nowadays, CPU and GPU have been fully integrated into the virtualization platform, but commercial virtualization support for FPGA is still lacking. To address this challenge, we present VMOpenCL, an OpenCL wrapper on VMware, to enable efficient heterogeneous resource sharing on commercial VMware platforms. VMOpenCL allows OpenCL applications running on the guest operating system in VMware to utilize the hardware resources in the host with negligible overheads of less than 0.5ms per API. With the VMOpenCL, the modern GPU, FPGA and other accelerators which are compatible with OpenCL can be integrated into the VMware for cloud computing.

Categories and Subject Descriptors D.2.2 [Design Tools and Techniques]: Software libraries

General Terms Libraries, Performance

Keywords GPU, FPGA, OpenCL, VMware, Heterogeneous Computing

1. Introduction
As an enormous revolution, cloud computing has infiltrated in company’s business model, government’s operation and daily personal lives [1]. By consolidating and managing a large number of computing resources over a real-time communication network, cloud computing is able to provide organizations or enterprise with the required computation resources in a cost-effective way. Virtualization is a fundamental enabling technology for cloud computing. By allowing the hardware resources to be shared, virtualization increases the resource utilization and in turn reduces energy and monetary costs. Virtual resources are dynamically allocated to handle increased computation demands, and the computation can also be migrated from one physical server to another for load balancing or data centric based computation like MapReduce [2].

Recently, heterogeneous cloud computing has been gaining attention in both industry and academia. Graphics Processing Unit (GPU), Many Integrated Core (MIC) Architecture [3] and Field-Programmable Gate Array (FPGA) as accelerators have come into the picture. The heterogeneity provides benefits in terms of performance and power efficiency. The GPU is a many-tiny-core and multi-threaded processor with tremendous computational horsepower and extremely high memory bandwidth. Today’s K20X with new Kepler [4] compute architecture can achieve 3.95TFLOPS in single precision and 1.31TFLOPS in double precision floating point calculation and the bandwidth of graphics double data rate (GDDR) memory can reach to 250 GB/sec. The MIC is a general purpose many-coarse-core coprocessor based on the Intel Architecture: traditional X86 program can be directly run on it without being modified. For Intel Xeon Phi coprocessor 5110P (MIC), they use 60 cores (at 1.053GHz) with 240 threads and the memory bandwidth up to 320 GB/sec [5]. The FPGA is a highly customizable fine-grained digital logic device with dedicated hardware blocks for high performance arithmetic functions. It can dynamically customize the computation pipeline and the interconnect of the processing elements. Multiple processing units can be organized to work in parallel or pipelined style by customer after manufacturing. Furthermore, it is extremely efficient in custom bit-level processing compared with the CPU, MIC and GPU [6]. Nowadays, XILINX VC709 Kit provides 3600 DSP slices [7] and about 30 GB/sec memory bandwidth. A computing device with GPU, MIC and FPGA can achieve the best energy efficiency when the computing algorithm is matched with the optimal computing pattern which the computing device presents. The key features of the above mentioned processing cores are summarized in Table 1.

<table>
<thead>
<tr>
<th>Model</th>
<th>Perf(Gflops)</th>
<th>Bandwidth (GB/s)</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU(K20X)</td>
<td>3950</td>
<td>250</td>
<td>Mass tiny cores</td>
</tr>
<tr>
<td>FPGA</td>
<td>360</td>
<td>29.9</td>
<td>Mass interconnected cores</td>
</tr>
<tr>
<td>MIC (Phi)</td>
<td>2022</td>
<td>120</td>
<td>Mass complex cores</td>
</tr>
</tbody>
</table>

VMware is the leading player in the global virtualization and cloud infrastructure market. A typical virtualization server contains supervisor and virtual machines. The supervisor or host operating system on the host machine provides the execution environment for the virtual machines and the management utilities. A group of virtual machines running on one or more host computers can work together via variable interconnecting architecture to form a virtualized cluster. GPU virtualization is an emerging technology for the cloud computing [8]. Currently, NVIDIA plays a substantial role in this area and has unveiled the first hardware virtualization platform in the newly released Kepler architecture. Meanwhile, VMware has
started to introduce GPU virtualization since the vSphere platform. However, there is no such software framework to support GPU, Intel MIC and FPGA, all together simultaneously with the virtualization on the cloud computing environment.

The main contributions of this paper are listed as follows:

- Propose and implement an OpenCL based framework – VMOpenCL on VMware to enable computational resources sharing on a heterogeneous platform;
- Implement the VMOpenCL client on VMware virtual machine to enable OpenCL support on the Virtual Machine.
- Implement the VMOpenCL Server on the host to handle the requests issued from the VMOpenCL clients and manage the underlying heterogeneous platform.
- Implement automatic client-service code generation for API calls generation.
- Optimize the performance of the VMOpenCL by using Virtual Machine Communication Interface (VMCI) [15].
- Benchmark the performance of the VMOpenCL with examples.

Our work originates from the consideration on how GPU, FPGA and Intel MIC can be enabled in the cloud computing with virtualization. There are many existing works in this topic, including rCUDA [9], gVirtuS [10], vCUDA [11], GridCUDA [12]. They use CUDA APIs as an interface and transparently redirect local CUDA API calls to remote GPGPU for execution. So the CUDA program can be distributed over multiple GPGPUs. However, their approach is not suitable for our goal because we want to integrate FPGA into the underlying hardware platform. There are also other solutions to integrate FPGAs which use OpenCL API as an interface such as Hybrid OpenCL [13], OpenCL Remote [14]. However they cannot be combined with so many devices with modern commercial Virtual Machine environment. Comparing to CUDA, OpenCL is a cross-platform open standard with wide spectrum of support from many companies like Intel, NVIDIA, AMD, ARM etc. Unlike previous approaches, our implementation combines VMware and OpenCL together, enabling the virtualization technology for OpenCL. Virtual machines can assist to virtualize the computation resources (GPU, FPGA, and Intel Xeon Phi). The low level service running on the host machine aims to balancing the load of the OpenCL’s requests from the virtual machines.

The rest of the paper is organized as follows: In Section II, we present background and related literature. Section III describes the system framework of how to design and implement the VMOpenCL. Section IV discusses the experiment test results in detail. In Section V we conclude this paper and give a brief overview of the future work.

2. Related Work

Traditionally processor architecture and design has followed Moore’s law to increase the performance. However, the physical constraints such as memory-wall [16], power-wall [17], ILP-wall [18] have blocked the way for further improvements. Heterogeneous architecture gives a more efficient way of using the transistors and reduce the power. Heterogeneous computing platform consists of different architectures, using application dedicated data-path to give the maximum performance and power efficiency. In 2009, Michael Showerman et al. reported in [19] a cluster that is built with compute nodes each containing 2 dual-core AMD Opteron CPUs, 4 NVIDIA Quadro FX5600 GPU cards and 1 NullaTech H101 FPGA accelerator card. In 2010, Kuen Hung Tsai and Wayne Luk introduced in [20] a heterogeneous cluster with 16 nodes each containing 1 AMD Phenom Quad-Core CPU, 1 NVIDIA Tesla C1060 card, and 1 ADM-XRC-5T2 FPGA card. There are many efforts made by the industry and academia to compare the performance and power consumption of the GPU, FPGA and other accelerators, like those in [21][22][23]. They showed that there is no definite winner between the GPU and FPGA for all applications, each has its advantage over the other for some particular application cases. Therefore, the industry needs a software framework to dynamically adapt the application to match the underlying hardware.

Heterogeneous platform combines different types of computation units increasing the burden for programmers. In order to support GPU and FPGA on a cloud computer, rCUDA [9], gVirtuS [10], vCUDA [11], GridCUDA [12] are introduced by the academia. rCUDA approaches to create a CUDA compatible virtual device and utilizes the GPU resource in remote machines, allowing fewer GPUs to be installed. gVirtuS and vCUDA allow an instantiated virtual machine to access GPUs in CUDA. Other implementations such as Hybrid OpenCL [13], OpenCL Remote [14] use a virtual OpenCL device to operate remote GPUs over the network and do not take the benefit of VMware and VMCI.

OpenCL [24] is an open, royalty-free standard for heterogeneous computing. It was initially developed by Apple Inc. On 16 June 2008, Apple initiated the Khronos Group with AMD, IBM, Intel and NVIDIA. OpenCL uses host and device model which is similar to CUDA. The host code runs on the host machine, while kernel code runs on the device node. The host code uses a set of APIs to manage the kernel. OpenCL kernel uses a subset ISO C99 with extensions. OpenCL is widely supported on Linux, Mac OS X and Windows with many vendors. For the GPU side, NVIDIA, AMD and Intel have the OpenCL support. For FPGAs, Altera has also released software SDK for OpenCL. For the limitations of hardware, OpenCL on Altera only implements mostly features of OpenCL. It is compatible with OpenCL but not fully supported [25]. For Intel MIC processor, Intel has also released the OpenCL SDK.

3. VMOpenCL Framework

As Figure 1 illustrates, VMOpenCL is a framework to enable computation acceleration devices such as GPU, FPGA and Intel MIC to be shared among the virtual machines via OpenCL. The VMOpenCL framework is organized in two parts. One is the VMOpenCL client (VMOpenCLc) and the other is the VMOpenCL server (VMOpenCLs). The VMOpenCLc is located at the virtual machine side and is implemented as a runtime library associated with a configuration file. The runtime library implements a subset
of the OpenCL v1.2 core APIs (Application Programming Interface), while the configuration file holds the information on how to connect to the VMOpenCL server on the host. On the host side (VMOpenCL server), the host collects all the OpenCL function call requests issued from different VMOpenCL clients on the virtual machines, and sends these requests to the real hardware by vendor drivers via ICD loader (the ICD loader is an OpenCL extension which allows multiple vendors to co-operate on one host system) in a balanced way. After the call is finished, it collects and sends the results to the corresponding VMOpenCL clients.

To have a good design of VMOpenCL, there are several design considerations across the entire design phase.

**Transparency:** VMOpenCL should be compatible with currently used OpenCL themes. Our library has the same architecture as the other vendors.

**Performance:** VMOpenCL should have very little overhead, making the application run at the same speed as it runs on the native platform. We address this by using VMCI.

**Functionality:** VMOpenCL should support the existing APIs that are currently being used by native OpenCL application. We implement a subset of OpenCL v1.2 (not including OpenCL extension). This is compatible with almost OpenCL applications for accelerator nowadays.

**Scalability:** The framework should adapt well with increasing number of users. Our design uses process pair, so the maximum number of users is only limited by the number of processes in the operating system. For our configuration, 32786 processes can be supported simultaneously. It should be noted that Linux configuration can be adjusted to support more processes.

**Modularity:** The software should use precisely defined interface with independent components. Our implementation is layered and based on the standard socket for the communication.

### 3.1 VMOpenCL Client

As mentioned above, the VMOpenCL client consists of a runtime library, OpenCL standard header files and a configuration file. The runtime library is named libVMOpenCLc which supports the OpenCL v1.2 core API functions. The header files include the standard definitions to support API. The configuration file contains the information for the communication between the VMOpenCL client and VMOpenCL server, which is merely the port number of the VMOpenCL server seen in the VMCI from the virtual machines. The port number is set during the installation of the VMOpenCL framework. The application programmers need not concern about the port number in the configuration file. The only thing they need to do is to ensure the application is linked to the VMOpenCL client runtime library. This is similar to the normal OpenCL application development process. Through the VMOpenCL client runtime library, the application can query and utilize the underlying OpenCL devices on the host side as if these devices reside in the guest system, thus achieving a transparent access from the guest system to the host platform.

Once an application creates and calls the OpenCL APIs for the first time, the runtime library named libVMOpenCLc load the host service port from the configuration file and creates a socket connection to the VMOpenCL server side. The VMOpenCL server verifies the command and returns a new random service port to this client. After this, the libVMOpenCLc library closes the first connection and connects to the random service port. After a connection is established, API calls can encapsulate the arguments from the application on the virtual machine side and send to a VMOpenCL server on the host side by VMCI for further execution. After the VMOpenCL server returns the information to the application in virtual machine from the API call, it gathers the information and returns to the caller. This virtualization can combine different requests from the virtual machines into one real host, allowing the resources to be shared. As mentioned before, our framework use VMCI as the basic communication interface.

The VMCI is an efficient communication interface between virtual machine and the host operating systems as shown in Figure 2. The VMCI software stack of VMWare avoids using the TCP/IP stack in the guest and host operating systems. Instead, the VMCI uses shared memory as an internal communication channel to circumvent unnecessary network stack and archive low overhead and high throughput. The VMCI is composed of four parts: the VMCI library, the VMCI driver, VMCI device and the VMCI core. The VMCI library is for the applications in the virtual machine and host which need to communicate with the other side. The VMCI driver and device is used by the VMware for emulating the communication device and supporting the communication demands from the applications. The VMCI core runs on the host machine to exchange the data between the applications in virtual machines and the host machine. Alternatively, it is also possible to use RPC (Remote Procedure Call) [26] and TCP/IP socket to implement the communication. One advantage of using RPC or TCP/IP is the flexibility since the VMOpenCL is bonded to VMware when using VMCI. However, the RPC and TCP/IP socket need to go through the TCP/IP stack, which incurs a large overhead. On the other hand, modern commercial data centers are mainly using VMWare to provide virtualization, and performance is the biggest priority. With that consideration, we select VMCI with VMware for performance instead of flexibility. By using VMCI, high throughput can be observed for larger socket buffer sizes, which is close to the native API execution on the host and is more than 3 times the maximum throughput obtained compared to using TCP/IP sockets.

### 3.2 VMOpenCLs

The VMOpenCL server reacts to the requests issued from the VMOpenCL clients and manages the underlying devices. The VMOpenCL server is composed of a service manager, agent processes and Virtual OpenCL service (VOCL service, VOCls) as depicted in Figure 3. The service manager listens on a designated service port after system boot-up. Upon receiving the request from the VMOpenCL client, the service manager creates an agent process and returns the agent’s service port number to the VMOpenCL client. One agent process responds to one application instance from the VMOpenCL client. Then the OpenCL call requests from the application instance of the VMOpenCL client are serviced by the corresponding agent process on host machine side. One agent process also manages the context information of the respective application instance of the VMOpenCL client. After that, all the API requests are sent to VOCL service. The VOCL service is based on the ICD
with an Intel Core i7 870 @2.93GHz, 8GB DDR3 memory, an

4. Evaluation and Results

4.1 Setups

We performed the experimental results on a workstation equipped with an Intel Core i7 870 @2.93GHz, 8GB DDR3 memory, an

4.2 Results and Discussions

To evaluate the overhead of our framework, we use several benchmarks to compare the overhead of APIs among the native host execution, TCP/IP socket and VMCI based wrapper on VMware. The native host execution means that executing the API on the native host OS purely without any virtual machine support. We also employ TCP/IP socket and VMCI APIs call based benchmarks with virtual machine support. Figure 4 demonstrates time cost via the three types of APIs call, respectively named with the native (NATIVE, in blue color), TCP/IP based (VMTCP, in green color) and VMCI-based (VMCI, in red color) call time. Figure 5 illustrates the overhead after applying the virtual machine and the data transportation layer, by subtracting the native host execution time from the VMTCP and VMCI results in Figure 4.

The study results show that the overhead of VMOpenCL APIs call consumes a fraction of time compared to that via TCP/IP on the VMware. On the average, the function calls via TCP/IP socket from VMware to host cost 200ms more time, while it only takes 0.5ms for VMOpenCL framework via VMCI transport. Our framework demonstrate a strong advantage for the accumulated latency will tremendously impair the overall performance of massive parallelism in the virtual environment. Since the TCP overhead is overwhelmingly significant over the API actual execution time, in consequence that the VMTCP do not have noticeable variations. Exceptions are the functions related to offloading, uploading and executing kernels, which are essential for data-intensive and compute-intensive computing, like clCreateBuffer, clEnqueueReadBuffer, clEnqueueNDRangeKernel, because the performance is related to the payload sizes, exchange frequency and task computation density.

In Figure 6, we compare the data transfer time from host to device and device to host among the native host execution, TCP/IP socket and VMCI wrapper in the VMware on a native host machine. We stripped various payload sizes of the transactions from 128 bytes to 32M bytes on both uploading and offloading operations. To avoid measurement error in the small sized payload packages, we average the multiple execution time for each transaction. The result demonstrates that the performance of uploading and offloading in proportion to the payload size of each transaction, since transactions with larger payload sizes naturally increases the throughput for reducing the package split overhead. The uploading speed can reach up to 3Gbps and the offloading speed can reach up to 6Gbps when the payload size rises to 32M bytes. On the contrary to VMCI, the peak performance of VMTCP can only reach up to 900Mbps. Those can greatly increase the performance of data-intensive computing on today’s virtual machine based Big data computing platform.

<table>
<thead>
<tr>
<th>Application</th>
<th>MatrixMul</th>
<th>BoxFilter</th>
<th>NBody</th>
<th>DCT</th>
<th>oclFDTD3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATIVE</td>
<td>1.65</td>
<td>1.76</td>
<td>5.97</td>
<td>2.35</td>
<td>21.17</td>
</tr>
<tr>
<td>VMTCP</td>
<td>14.02</td>
<td>22.87</td>
<td>16.58</td>
<td>15.35</td>
<td>492.96</td>
</tr>
<tr>
<td>VMOCL</td>
<td>4.98</td>
<td>5.25</td>
<td>9.05</td>
<td>5.36</td>
<td>32.15</td>
</tr>
</tbody>
</table>

In the last set of experiment, we set up five selected applications for our benchmarks. The benchmarks range from simple to complex, including Matrix Multiplication, Box filter, N-Body,
Figure 4. API Call Time

Figure 5. APIs Call Overhead in VMTCP and VMCI
DCT and Finite difference. Table 2 illustrates the execution time of the applications running the kernel plus the timing of invoking the OpenCL APIs. The detail data list in The configuration of three version of benchmarks are same except the linked library. So the difference between the VMTCP and NATIVE columns is the API overhead for VMTCP, and the difference between the VMCI and NATIVE columns is the API overhead for VMCI. As can be seen, our VMOpenCL framework has on average 20% overhead impact on application performance running on a native host machine, compared to the VMTCP framework.

5. Conclusion and Future Work

Cloud computing is becoming essential today, and heterogeneous computing with virtualization is a key accelerator for cloud computing. In this paper, we present a VMOpenCL framework to integrate GPU, FPGA and Intel MIC into the commercial cloud computing environment. This framework utilizes the benefit of the virtualization on VMware, which virtualizes the CPU resources. By combining with OpenCL which virtualizes the GPU, FPGA, and other OpenCL compatible resources, it enhances more computation resource locality to the data store such that the power consumption is reduced. In this paper, we have discussed the architecture of our VMOpenCL framework and explained how the system works in VMOpenCLc and VMOpenCLs components. We stressed our main optimization on VMCI and presented experimental results. We demonstrate that the overhead of our framework is insignificant comparing to TCP based transport, the performance only drops slightly after the virtual machine involved and our implementation does not affect the final computation result. In the future, we intend to set up the Altera Stratix V card and Intel MIC cards to demonstrate the additional benefits on our future platform.

References

Using Erlang Skeletons to Parallelise
Realistic Medium-Scale Parallel Programs

Vladimir Janjic  Adam Barwell  Kevin Hammond
University of St Andrews, Scotland, UK.
{vj32,adb23,kh8}@st-andrews.ac.uk

Abstract
This paper shows how the Erlang skeleton library, Skel, can be used to parallelise three use cases, and the performance gains achieved through doing so. These use cases include: the Discrete Haar Wavelet Transform, the Single Machine Total Weighted Tardiness Problem using Ant Colony Optimisation, and an image merge. Using the parallelised versions of these applications, and a 24-core shared memory machine, we were able to achieve speedups of 16.63 for the Haar Transform, 11.77 for the ACO problem, and 14.51 for Image Merge. This demonstrates that, with relatively little effort, the Skel library can be used for the parallelisation of Erlang applications, whilst obtaining good speedups.

Keywords  Erlang, Parallelism, Skeletons

1. Introduction
The single-core processor, which has dominated for more than half a century is now obsolete. Machines with dual-, quad-, and even hexa-core CPUs are already common place in desktop machines, and CPUs with 50 cores as standard have already been announced 1. There has been a seismic shift between sequential and parallel hardware, but programming models have been very slow to keep pace. Indeed, many programmers still use outdated sequential models for programming parallel systems, where parallel concepts have effectively been bolted-on to the language, rather than high-level parallel constructs being a core feature. What is needed is an effective solution to help programmers think parallel. In the context of parallel programming, parallel design patterns represent a natural language description of a recurring problem, and of the associated solution techniques that the parallel programmer may use to solve that problem [13].

Porting existing sequential applications to large-scale shared-memory parallel systems usually comprises of identifying where in the application the potential parallelism lies, and subsequently implementing low-level parallel code that exploits this parallelism in a useful manner. This implementation is usually very tedious and error-prone, since the programmer must commonly explicitly handle thread creation, communication, and synchronisation. Algorithmic Skeleton Libraries [9] attempt to abstract away from these tedious details by providing a set of high-level skeletons as functions that capture common parallelism patterns. The user then only needs to provide the sequential code for the skeleton(s) he chooses, with the low-level parallel details (such as thread creation, communication and synchronisation) abstracted away.

In this paper we employ three use cases to demonstrate the effectiveness of Erlang as a parallel programming language. These use cases are built using a recent skeleton library for Erlang, Skel, which provides a set of classical, well-founded parallel implementations of the most well known, and useful, skeletons. For each use case, we describe how Skel is used to introduce parallelism, and evaluate its effectiveness in terms of performance. This evaluation has been undertaken using a 24-core shared memory system.

2. Background

2.1 Erlang
Erlang is a strict, impure, functional programming language with built-in support for concurrency. This concurrency model allows the programmer to be explicit about processes and communication, but implicit about placement and synchronisation. Erlang supports a lightweight threading model, where processes model small units of computation, or tasks. The scheduling of processes is handled automatically by the Erlang Virtual Machine, which also provides basic load balancing mechanisms. Erlang typically has three primitives for handling concurrency:

• spawn(), producing a lightweight Erlang process that executes a specified function;
• !, allowing messages to be explicitly sent from one Erlang process to another; and,
• receive, similarly allowing messages to be received from another process queue.

Furthermore, Erlang supports fault tolerance, by enabling groups of processes to be supervised, and new instances of processes spawned in the event of failure. Although Erlang supports concurrency, there has been little research into how Erlang can be used to effectively support deterministic parallelism.

2.2 Skeletons
An algorithmic skeleton [9] is an abstract computational entity that models some common pattern of parallelism. One example of which is the parallel pipeline — i.e. the parallel execution of the sequence of computations over the set of inputs, where the output of one computation is the input to the next.

Skeletons are typically implemented as high-level functions, the programmer supplying a skeleton with sequential, problem-

---

1 Intel’s Many Integrated Core Family
specific code, and any skeletal parameters. The skeleton itself handles the parallel aspects of the computation – the creation of parallel threads; communication and synchronisation between these threads; load balancing; etc.

Skel [7], is a Domain Specific Language implemented in Erlang for expressing parallelism using algorithmic skeletons. It currently provides a small number of classical skeletons that are considered the most useful. In this paper, we consider the following skeletons, each of which is implemented in Skel.

- **seq** is a trivial wrapper skeleton that implements the sequential evaluation of a function, \( f :: a \rightarrow b \), applied to a sequence of inputs, \( x_1, x_2, \ldots, x_n \).
- **pipe** models a parallel pipeline, applying each of the functions \( f_1, f_2, \ldots, f_m \) in turn to a sequence of independent inputs, \( x_1, x_2, \ldots, x_n \), where the output of \( f_i \) is the input to \( f_{i+1} \). Parallelism arises from the fact that, for example, \( f_1(x_k) \) can be executed in parallel with \( f_{i+1}(f_i(x_{k-1})) \). Here, each \( f_i \) has type \( a \rightarrow b \).
- The **farm** skeleton models the application of a single function, \( f :: a \rightarrow b \), to a sequence of independent inputs, \( x_1, x_2, x_3, \ldots, x_n \). Each application, \( f(x_1), f(x_2), f(x_3), \ldots, f(x_n) \), can be executed in parallel.
- **Feedback** is a wrapper skeleton allowing outputs of an inner-skeleton, of type \( f :: a \rightarrow b \), to be reused as inputs. Each result of the application of an input to the inner-skeleton is checked against a constraint-checking function \( f_c :: a \rightarrow \text{bool} \). Outputs are repeatedly passed back as inputs for each original input until the constraint-checking function fails.
- Finally, the **map** skeleton is a variant of a **farm**, where each independent input, \( x_i \), is partitioned \( p :: a \rightarrow \{ b \} \) into a number of sub-parts that can be worked upon in parallel. A worker function, \( f :: \{ b \} \rightarrow \{ c \} \), is then applied in parallel to each element of the sublist, followed by the combination \( c :: \{ c \} \rightarrow d \) of the results into a single result for each input.

Using a combination of higher-order functions, lists, and tuples, Skel is presented as a library from which one of two top-level functions is called. This invokes the library’s operations according to the functions and parameters passed by the programmer.

The requirements as to these functions and parameters are determined by the skeleton to be employed. The function \texttt{skel:run} takes two lists as arguments. The second is the input to be worked on. Where the first, referred to as a **workflow**, contains the tuples that determine which skeletons are to be used. These tuples have at least two elements, where the first is an atom – e.g. \texttt{seq}, \texttt{map}, \texttt{farm}, or \texttt{pipe} – specifying the skeleton to be used; successive elements specify functions to be used in the computation. Each workflow is an implicit pipeline, allowing the outputs of each tuple to be passed directly into the next.

We give an example of an invocation of Skel in Algorithm 1. Here, a task farm with ten workers applies each input in the list \texttt{Input} to a function, \( f \), whose output is then passed to a function, \( g \). Both functions are developer-defined. In this example, we observe that \( f \) and \( g \) are located in the same Erlang module as the call to Skel. Functions from other modules may be used by using the name of said module in place of the \texttt{?MODULE} macro.

**Algorithm 1** Exemplar invocation of Skel.

```
skel:run([[farm, [{seq, fun ?MODULE:f/1}],
            {seq, fun ?MODULE:g/1}], 10], Input).
```

The retrieval of results from work performed by Skel is dependent on the top-level function used in its invocation. In Algorithm 1, \texttt{skel:run} is used, requiring the programmer to explicitly fetch the result using a receive block. This should expect the tuple, \{sink_results, Result\}, where Result is the list of results produced by Skel.

Alternatively, results may be returned implicitly by calling the other top-level function: \texttt{skel:do}. Taking the same arguments as \texttt{skel:run}, \texttt{skel:do} acts as a wrapper for the above process, automatically returning \texttt{Result}.

3. The Discrete Haar Wavelet Transform

In mathematics, the **Haar wavelet** [14] is a sequence of “square-shaped” functions that can be used to approximate any square-integrable real function. An important example of the use of Haar wavelets in Computer Science is the **Discrete Haar Transform**, which is heavily used in image and signal processing. The Discrete Haar Transform consists of applying the operation

\[
y = T x T^T,\]

for an input vector \( x \), and a fixed Haar matrix

\[
T = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
\]

producing the output vector \( y \). A one-dimensional Discrete Haar Transform, where \( x \) is a pair of real numbers, is used in signal processing (e.g. sound compression). A two-dimensional Discrete Haar transform, on the other hand, where \( x \) is 2x2 matrix, is used for image compression, as for each 2x2 matrix of pixels \( x \), it gives a 2x2 matrix of pixels \( y \) as a result, where most of the energy of \( x \) is contained in the top left pixel of \( y \). Applying the Discrete Haar Transform to an image (alternatively a signal) consists of splitting the image (signal) into 2x2 subimages (pairs of elements) and applying the Haar Transform to each subimage (pair). See Figure 1 for the example of an image before and after the Discrete Haar Transform is applied.

We consider two use cases of applying the Discrete Haar Transform:

1. **audio compression**, where a 1D Discrete Haar Transform is applied to a stream of audio files; and,
2. **video compression**, where a 2D Discrete Haar Transform is applied to a stream of images captured by a camera.

In both cases we observe that the same operation – the Discrete Haar Transform – is applied to a set of independent inputs, i.e. a stream of audio files or images. Hence, parallelism may be introduced via a task farm.
3.1 Implementing the Haar in Skel

In this section, we illustrate the process of porting the sequential Haar use-case to the Erlang parallel skeleton library, Skel. The basic program comprises a three-stage function composition, as shown in Figure 2. In the first stage, a stream of files is read, where each file comprises a video or an audio file. These audio files/images are then passed to the second stage, where the Discrete Haar Transform is applied – 1D for audio files and 2D for images. Finally, in the third stage, the transformed audio files/images are sent across a network for further processing (if necessary). In order to parallelise this algorithm, we decided to introduce a task farm for the second stage (see Figure 3). In order to do this, we created a skeleton call to a Skel task farm, where the number of farm workers is controlled by the user via an input argument. A task corresponds to applying sequentially the Discrete Haar Transform to one audio file/image. The tasks \( X_1, X_2, \ldots, X_n \) are distributed to workers using a farm \textit{emitter}, in an on-demand fashion where tasks are sent to idle workers. The task results \( X'_1, X'_2, \ldots, X'_n \) are sent to the farm \textit{collector}. Both the emitter and collector are hidden from the user, and are provided by the basic library framework.

3.2 1D Haar Transform

The porting of the 1D Haar Transform proceeded in two stages, where the original algorithm is shown in Algorithm 2. Here, the sequential 1D Haar Transform, \texttt{sequential1d}, is defined as mapping a function, \texttt{haar_1d_wrapper}, over a list of \texttt{Vectors}.

\textbf{Stage 1: Introducing a Task Farm}    Introducing parallelism into the program was done by first identifying the sub-expression in the program that generated the output list, and where each operation on the list could be computed in parallel. In the 1D Haar Transform example, the operation \texttt{sequential1d} is converted into a task farm, where each worker computes the 1D Haar Transform for an input vector. The result of the task farm is a list of transformed vectors, as illustrated in Algorithm 3.

Here, the program can be broken down into a number of key components:

- \texttt{skel:run} denotes a call to the top-level \texttt{run} function in the Skel library, where the parameter to \texttt{run} is a (nested) Skeleton;
- \texttt{farm} denotes a farm skeleton;
- \texttt{seq} denotes the workers of the \texttt{farm} skeleton are the sequential function, \texttt{1d_haar_wrapper};
- 24 denotes the number of \texttt{farm} workers; and, finally,
- \texttt{Vectors} is the input list of tasks.

\textbf{Stage 2: Chunking}    While using a task farm for the 1D Haar Transform creates a reasonable amount of parallelism, the parallelism is too fine-grained and the program does not scale as we would typically expect. This is a common problem in the early stages of writing parallel programs. To combat this, we introduce chunking to the task farm, allowing us to group together a number of small tasks into one larger parallel task, where each parallel thread operates over a sub-list rather than just one element.

We want each worker to be busy, so we chunk by groups of 4 elements. (2048/4 = 512 tasks for each worker). By chunking in this way, we also decrease the communication costs, and reduce parallel overheads. Chunking can generally be achieved in a variety of different ways. In our example, we modify the task farm, manually refactoring it to a pipeline with a \texttt{partition} and \texttt{combine} stage, as illustrated in Algorithm 4.

\begin{algorithm}[h]
\caption{1D Haar Transform Using a Skel Task Farm with Partitioning and Combining}
\begin{algorithmic}
\State \texttt{theSkel1D(Vectors) -> skel:run(\{\{farm, \{\{seq, \texttt{fun}(V) \to}\}
\begin{align*}
&\texttt{haar_1d_wrapper(X) \to \texttt{seq_haar_1d_a(R, A, I, Lim))},} \\
&\texttt{24\},} \\
&\texttt{partition(Vectors, C, Len)),} \\
&\texttt{receive} \\
&\texttt{\{sink_results, Results\} \to combine(Results)} \\
&\texttt{end.}
\end{align*}
\end{algorithmic}
\end{algorithm}

\begin{algorithm}[h]
\caption{1D Haar Transform Using a Skel Task Farm}
\begin{algorithmic}
\State \texttt{theSkel1D(Vectors) -> skel:run(\{\{farm, \{\{seq, \texttt{fun}(V) \to}\}
\begin{align*}
&\texttt{haar_1d_wrapper(X) \to \texttt{seq_haar_1d_a(R, A, I, Lim))},} \\
&\texttt{24\},} \\
&\texttt{split(Vectors, C, Len)),} \\
&\texttt{collect(Results)),} \\
&\texttt{receive} \\
&\texttt{\{sink_results, Results\} \to Results} \\
&\texttt{end.}
\end{align*}
\end{algorithmic}
\end{algorithm}

In this way, different ants generally produce different (but similar) the pheromone trail for that component or (with the probability of a solution, an ant (with the designated probability of a solution to the problem, with the solution being partially guided by the pheromone trail). Once the iteration is finished and all ants have computed their schedules, the schedule that obtains the minimal total weighted tardiness is selected, and the pheromone trail is updated according to that solution. After that, the next iteration, where ants compute new solutions based on the new pheromone trail, is started.

An example ACO algorithm that we consider in this deliverable is computing a solution to a Single Machine Total Weighted Tardiness Problem (SMTWTP) [5]. In SMTWTP, we are given \( n \) jobs, whereas each job \( i \) is characterised by its processing time, \( p_i \), deadline, \( d_i \), and weight, \( w_i \). The goal is to schedule execution of jobs in a way that achieves minimal total weighted tardiness. The tardiness of a job, \( i \), in a schedule is defined by \( T_i = \max\{0, C_i - d_i\} \), where \( C_i \) is the completion time of the job \( i \) in that schedule. The total tardiness of the schedule is defined as \( \sum w_i T_i \).

Under the ACO solution to the SMTWTP problem, each ant independently computes a schedule in each iteration. The pheromone trail that guides the computation of schedules is defined by a matrix \( \pi \), where \( \pi[i,j] \) is the preference of assigning job \( j \) to the \( i \)-th place in the schedule. Therefore, in each step of the solution computation, an ant will either pick the job with the highest preference for that position, or will choose a biased random selection (again based on the pheromone trail). Once the iteration is finished and all ants have computed their schedules, the schedule that obtains the minimal total weighted tardiness is selected, and the pheromone trail is modified to increase chances of selecting job in the same order as in the currently found best solution.

```
Algorithm 5 Partition and Combine Functions in Erlang

combine([],) ->[];
combine([X|Xs]) -> lists:append(X, combine(Xs)).

partition([], ChunkSize, Len) -> [List];
partition(List, ChunkSize, Len) ->
    case (length(List) < ChunkSize) of
        true -> [List];
        false -> Chunk = lists:sublist(List, ChunkSize),
        NewList = lists:sublist(List, ChunkSize+1, Len),
        [Chunk | partition(NewList, ChunkSize, Len)]
    end.
```

```
Algorithm 6 Partition and Combine Functions in Erlang

sequential2D(Images) -> [ haar_2d_wrapper(I) || I <- Images].
```

```
Algorithm 7 Partition and Combine Functions in Erlang

theskel2d(Vectors) ->
skel:run([ {farm, [{seq, fun !MODULE:2d_wrapper/1}], 24}], Vectors),
    receive
        {sink_results, Results} -> Results
    end.
```

3.3 2D Haar Transform

The porting process for the 2D Haar Transform proceeded in a similar way to the 1D Haar Transform, where the sequential algorithm is shown in Algorithm 6.

Here, a function, `haar_2d_wrapper` is applied to each element, \( I \), of the list of input images, `Images`. Porting this code to use a Skel task farm comprised of rewriting the above code into a call to the Skel library, introducing a farm skeleton, with 24 workers, and each worker is a sequential function, `2d_wrapper`. In this example, it is not necessary to employ chunking, as the tasks are already large enough to give sufficiently large computation, without saturating the system with an abundance of parallel tasks.

4. Ant Colony

Ant Colony Optimisation (ACO) is a heuristic for solving NP-complete optimisation problems, inspired by the behaviour of ants living in real ant colonies. An ACO algorithm consists of a number of iterations. In one iteration, each ant independently computes a solution to the problem, with the solution being partially guided by a pheromone trail produced by ants. To compute one component of a solution, an ant (with the designated probability \( q \) follows the pheromone trail for that component or (with the probability \( 1 - q \)) it performs a biased random selection of the component. In this way, different ants generally produce different (but similar) solutions. After the iteration is finished, and all ants have computed solutions, the best solution is chosen and the pheromone trail is updated according to that solution. After that, the next iteration, where ants compute new solutions based on the new pheromone trail, is started.

An example ACO algorithm that we consider in this deliverable is computing a solution to a Single Machine Total Weighted Tardiness Problem (SMTWTP) [5]. In SMTWTP, we are given \( n \) jobs, whereas each job \( i \) is characterised by its processing time, \( p_i \), deadline, \( d_i \), and weight, \( w_i \). The goal is to schedule execution of jobs in a way that achieves minimal total weighted tardiness. The tardiness of a job, \( i \), in a schedule is defined by \( T_i = \max\{0, C_i - d_i\} \), where \( C_i \) is the completion time of the job \( i \) in that schedule. The total tardiness of the schedule is defined as \( \sum w_i T_i \).

Under the ACO solution to the SMTWTP problem, each ant independently computes a schedule in each iteration. The pheromone trail that guides the computation of schedules is defined by a matrix \( \pi \), where \( \pi[i,j] \) is the preference of assigning job \( j \) to the \( i \)-th place in the schedule. Therefore, in each step of the solution computation, an ant will either pick the job with the highest preference for that position, or will choose a biased random selection (again based on the pheromone trail). Once the iteration is finished and all ants have computed their schedules, the schedule that obtains the minimal total weighted tardiness is selected, and the pheromone trail is modified to increase chances of selecting job in the same order as in the currently found best solution.
Figure 5. Two images serving as input to Image Merge, and the resulting image

Figure 6. The image merge process employed

From these stages we derive which skeletons will be used in implementing ACO. For the core set of tasks – i.e. the ants computing their schedules – we use a map skeleton to divide ants equally across workers. For determining which of these schedules will update the pheromone trail, a pipe is used to pass the results of the map into a sequential function as in Algorithm 8. This pipe is itself nested within a feedback skeleton (Algorithm 9) so that the process may be repeated until an overall best solution is chosen. Figure 4 illustrates this idea graphically. Algorithm 9 also gives the skel:do call necessary to run ACO.

5. Image Merge

The third use case features the merging of two images, as demonstrated in Figure 5. Conceptually, this process may be considered in three stages:

1. the reading of a pair of images, and loading them into memory;
2. the merging of the pair; and
3. the writing of the resulting image to disc.

For this example, we choose to focus on the first two stages. Whilst writing the results to disc is a necessity in any truly practical application of an image merge, the performance of this phase is heavily dependent on hardware, and might confuse our results in evaluating the effectiveness of our approach.

As Figure 6 illustrates, the merge stage may itself be broken into three distinct stages. Firstly, and once the two images to be merged are received from the reading stage, all alpha regions within the images are removed. Secondly, all white pixels are turned black; thus prepared, the two images are then merged.

Image merge employs a pipeline of two task farms, one each for both reading and merging stages. The image pairs to be merged are input tasks. This is illustrated in Figure 7, and code given in Algorithm 10 where the number of workers to both task farms are specified by the parameters NW1 and NW2.

Alternatively, Image Merge might be made parallel manually. One method is to spawn a new process for every task. Under this method, an additional stage is added to Image Merge to collect the results of the merge. The function given in Algorithm 11, using the functions defined in Algorithm 12, performs the manu-
increased, therefore reducing the communication overheads.

version, the number of tasks is reduced, but the computation size is system is saturated with many fine-grained tasks. In the chunking task farm variant, there is an abundance of parallelism, where the investigation is needed at this stage, we speculate that in the simple line) shows a maximum speedup of 13.6, where the version with responds to processing 4 vectors. The simple task farm variant (blue chunked version still uses a task farm, but here one tasks corre-

For the 1D Discrete Haar Transform, we executed the application over 2048 audio samples, each with a sample size of 4400. This execution of a 1D Discrete Haar Transform operates over a single over 2048 audio samples, each with a sample size of 4400. This processing one vector. The red line shows the chunked version (where figure, the blue line corresponds to the speedups of the simple task farm without chunking. As can be seen, the application starts to scale reasonably well, tailing off with around 12 workers with a speedup of 7.2. This is due to a fact that there are 24 tasks, so with 12 workers, each worker gets exactly 2 tasks. Increasing the number of workers to 13 (and up to 23) results in imbalance in the number of tasks allocated to workers. Therefore, some workers may be idle for considerable time, waiting for workers that got more tasks to finish. This further results in only small improvements in speedups when there are between 13 and 23 workers. Balance is again restored when there are 24 workers, hence the much better speedup of 14.15.

6. Evaluation Results

In this section we evaluate the above use cases, where all measurements have been made on an 2.4GHz 24 core, dual AMD Opteron 6176 architecture, running Centos Linux 2.6.18-274.el5, and Erlang 5.9.1 R15B01, averaging over 10 runs. We report absolute speedups against the original sequential versions.

6.1 1D Haar Transform

For the 1D Discrete Haar Transform, we executed the application over 2048 audio samples, each with a sample size of 4400. This translates to 2048 vectors, each with 4400 elements, where a single execution of a 1D Discrete Haar Transform operates over a single vector. Figure 8 shows the speedup results for the parallel version of the 1D Discrete Haar Transform using 1–24 farm workers. In the figure, the blue line corresponds to the speedups of the simple task farm variant of the application, where a task corresponds to processing one vector. The red line shows the chunked version (where the input list of vectors is partitioned into groups of 4 vectors). The chunked version still uses a task farm, but here one tasks corresponds to processing 4 vectors. The simple task farm variant (blue line) shows a maximum speedup of 13.6, where the version with chunking shows an improvable speedup of 16.63. Although more investigation is needed at this stage, we speculate that in the simple task farm variant, there is an abundance of parallelism, where the system is saturated with many fine-grained tasks. In the chunking version, the number of tasks is reduced, but the computation size is increased, therefore reducing the communication overheads.

In future versions of this paper we aim to provide further investigation into the effectiveness of chunking for both this example and the other examples herein.

6.2 2D Haar Transform

Figure 9 shows speedup results for the 2D Haar Transform, over 24 images, each 1024x1024 in size. Here, we only consider the simple task farm without chunking. As can be seen, the application starts to scale reasonably well, tailing off with around 12 workers with a speedup of 7.2. This is due to a fact that there are 24 tasks, so with 12 workers, each worker gets exactly 2 tasks. Increasing the number of workers to 13 (and up to 23) results in imbalance in the number of tasks allocated to workers. Therefore, some workers may be idle for considerable time, waiting for workers that got more tasks to finish. This further results in only small improvements in speedups when there are between 13 and 23 workers. Balance is again restored when there are 24 workers, hence the much better speedup of 14.15.

6.3 Ant Colony Optimisation

Figure 10 shows speedup results for the Ant Colony Optimisation for 1,000 jobs, using 32 ants over 10 iterations. We see that the application scales reasonably well as workers are added, reaching a peak speedup of 11.77 when 20 workers are employed. However, as more workers are added, speedup gains drop to 8.82 or similar.

In this example, we extend the number of workers to 32. With 32 workers, equal to the population of ants in the system, each ant has its own worker. This is achievable on a 24-core machine because Erlang does not assign one process to one core, instead using scheduling to dynamically assign processes to hardware.

However, as seen in Figure 10, there is little benefit in doing so. We suggest this drop in speedup caused by there being insufficient tasks for the number of workers used. With workers less active, there is also an increase in the relative cost to maintain parallelism.

6.4 Image Merge

The speedup results for the Image Merge example are illustrated in Figure 11. As Image Merge uses two task farms, we have varied the number of workers available to both. Along the x-axis we plot the number of workers employed by the second task farm, denoted \( F_2 \), handling the merge operation. With each set of points plotting speedups for a given number workers available to the first task farm, similarly denoted \( F_1 \), handling the read operation.

In this example we see a peak speedup of 14.51 when 12 and 20 workers are available to the first and second task farms, respectively. Figure 11 also shows that the first stage dominates when allocated too few workers, such as when only 2 or 4 workers are used. The merging stage receives its input from the reading stage, and so if that input cannot be provided at the same rate as it can be processed, further speedups are not possible regardless of the number of workers made available to the merging stage.

Figure 11 also illustrates a slowdown when the number of available workers for both task farms is too little; e.g. a speedup of 0.36 when \( F_1 = F_2 = 2 \). This is the result of parallel overheads. That the system spends more time on message passing etc. than is gained by doing work in parallel.

Our manual parallelisation of the Image Merge operation produces respectable speedups of 11.8. We note that, however, and with a peak speedup of 14.51, it is possible to achieve greater speedups using skeletons. That 32 lines of code are needed to achieve this result, compared with 4 when employing Skel, is also illustrative of the simplicity of introducing parallelism using skeletons. We aim to extend similar comparisons to both Haar and ACO examples in future versions of this paper.

---

**Algorithm 12 Functions Used in the Manual Parallelisation of Image Merge**

```erlang
readImageWrapper(Image, NextPid) ->
    NextPid ! {image, readImage(Image)}.
convertMergeWrapper(Image, NextPid) ->
    NextPid ! {merged, convertMerge(Image)}.
readAllImages([], _NextPid) ->
    ok;
readAllImages([Image | Images], NextPid) ->
    spawn(util, readImageWrapper, [Image, NextPid]),
    readAllImages(Images, NextPid).
convertAllImages(0, _NextPid) ->
    ok;
convertAllImages(N, NextPid) ->
    receive
        {image, Image} ->
            spawn(util, convertMergeWrapper, [Image, NextPid]),
            convertAllImages(N-1, NextPid);
        end.
collectAllImages(O, _NextPid, MergedImages) ->
    NextPid ! {done, MergedImages};
collectAllImages(N, NextPid, Acc) ->
    receive
        {merged, Image} ->
            collectAllImages(N-1, NextPid, [Image | Acc])
        end.
```

In future versions of this paper we aim to provide further investigation into the effectiveness of chunking for both this example and the other examples herein.
7. Related Work

The Skel framework was introduced in [7], together with a methodology for parallelising Erlang programs using refactoring tools and cost-models. In this paper we attempted to follow the methodology, replacing the refactoring tool-support with a manual refactoring process instead. Since the nineties, the skeletons research community has been working on high-level languages and methods for parallel programming [4, 6, 8–11]. Skeleton programming requires the programmer to write a program using well-defined abstractions (called skeletons) derived from higher-order functions that can be parameterised to execute problem-specific code. Skeletons do not expose to the programmer the complexity of concurrent code, for example synchronization, mutual exclusion and communication. They instead specify abstractly common patterns of parallelism – typically in the form of parametric orchestration patterns – which can be used as program building blocks, and can be composed or nested like constructs of a programming language. A typical skeleton set includes the pipeline, the task farm, map and reduction.

There has been a few previous attempts at parallelising Erlang applications, such as parallelising Dialyzer [2], and a suite of Erlang benchmarks [3]. However, none of the attempts exploit structured parallelism in the form of algorithmic skeletons, as outlined in this paper. Parallelism has been exploited in other functional languages, such as Haskell, using a strategies approach for implicit parallelism in GpH [15], and an explicit structured parallelism approach, using algorithmic skeletons, for Eden [12].
8. Conclusions and Future Work

Whilst many continue to write sequential software, the shift to multi-core processors require the modern programmer to think parallel. Yet parallelism is often tedious and error-prone when manually introduced. Conversely, algorithmic skeletons allow the simple and effective introduction of parallelism.

We have presented how the Skel library, an Erlang implementation of several skeletons, might be used. In so doing we give the introduction of structured, well-defined parallelism in three Erlang applications. These exemplary applications include the Discrete Haar Transform, the Ant Colony Optimisation problem SMTWTP, and the merging of two images.

Following their implementation we evaluated the effectiveness, in terms of performance gains, discovering reasonable speedups from the use of Skel. For the 1D Discrete Haar Transform we presented two variants; the first achieving a speedup of 13.6, and the second, using chunking, achieving 16.63. The Ant Colony Optimisation example demonstrated a speedup of 9.45. Lastly, the Image Merge achieved a speedup of 14.51.

In the case of Image Merge, there is a 23% increase in performance when using Skel over the manual introduction of parallelism. This example also illustrated the simplicity of the skeleton approach, with the Skel version requiring much less additional code than the manual version. These examples hence demonstrating that parallelism with little effort, whilst achieving good performance gains.

In future work, we intend on applying the library to more complex examples and use cases – the Erlang Dialyzer, for example – and further evaluating its effectiveness. It is also our intention to expand to a range of platforms, and an OpenCL [1] variant, which would also allow us to utilise GPU architectures.

Acknowledgements

This work has been supported by EU Framework 7 grants IST-288570 “ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems” (http://www.paraphrase-ict.eu), and IST-248828 “Advance: Asynchronous and Dynamic Virtualisation through performance Analysis to support Concurrency Engineering (ADVANCE)” (http://www.project-advance.eu).

References


Parallel video denoising on heterogeneous platforms

M. Aldinucci, G. Peretti Pezzi, M. Drocco, F. Tordini
Computer Science Dept. - University of Torino, Italy
{aldinuc, peretti, tordini}@di.unito.it, maurizio.drocco@gmail.com

P. Kilpatrick
Computer Science Dept. - Queen’s University Belfast, UK
p.kilpatrick@qub.ac.uk

M. Torquati
Computer Science Dept. - University of Pisa, Italy
torquati@di.unipi.it

Abstract

In this paper, a highly-effective parallel filter for video denoising is presented. The filter is designed using a skeletal approach, and has been implemented by way of the FastFlow parallel programming library. As a result of its high-level design, it is possible to run the filter seamlessly on a multi-core machine, on GPGPU(s), or on both. The design and implementation of the filter are discussed, and an experimental evaluation is presented. Various mappings of the filtering stages are comparatively discussed.

Keywords skeletons, fastflow, parallel patterns, multi-core, OpenCL, GPGPUs, heterogeneous platforms

1. Introduction

The ever increasing computing power available from off-the-shelf processors has allowed researchers to extend the number of applications in image processing and machine vision. One important step in any machine vision system is the image restoration phase, which has attracted the attention of the image processing community, especially with the increasing importance of real-time analysis of digital images and videos for video surveillance, etc. Variational methods, which basically solve optimization problems, are well known for their effectiveness, but are rarely exploited in image restoration due to their high computational cost and complexity of tuning [12, 26].

An efficient variational image restoration template based on FastFlow technology [4, 21] has been proposed in previous work [3]. Given a traditional noise reduction filter (e.g. based on adaptive median, adaptive center-weighted median) the template generates an efficient parallel variational filter running on both multi-core and GPUs. The resulting variational filter is edge-preserving and, compared with the original filter, exhibits a better Peak-Signal-to-Noise-Ratio (PSNR) and can restore images with higher levels of noise.

The proposed image restoration schema is organised in two successive stages: an early outline of an image is first detected and then it is denoised. The schema is implemented according to a high-level pattern-based approach (a.k.a. skeleton approach [15]), and deployed as a sequence of detect and denoise stages that are defined according to the map parallel paradigm and the map-reduce parallel paradigm, respectively.

This paper extends the previous work in two directions:

1. The restoration schema is applied to video streams as opposed to a single image, and is expressed using pipeline parallelism between the two stages.

2. The implementation, realised by way of the FastFlow programming framework, is tested on a heterogeneous platform composed of a multi-core equipped with two GPGPU devices.

Thus, this work incorporates the business code of the earlier work [3]: the difference from the earlier work lies in the nature of the skeleton framework in which the business code is placed. Here the sequential composition of stages is replaced by a two stage pipeline through which a stream of images is directed. In addition, the FastFlow framework itself has evolved since the earlier work: now it incorporates the facility to deploy applications onto heterogeneous platforms comprising both CPUs and GPGPUs. This has an interesting implication for the application: originally the second stage (denoising) was orders of magnitude slower than the first stage (noise detection) and so there was no point in parallelising the first stage. Now, as will be seen, with the denoise stage on a GPGPU, in some cases we expect it to become faster than the (sequential) noise detection stage and so it is worthwhile also to parallelise the noise detection stage.

For the sake of simplicity, in this paper impulsive noise (e.g. salt-and-pepper) is considered. The same approach has been shown to be effective for several types of noise (e.g. Gaussian). The analysis of restoration quality for different types of noise is beyond the scope of the present work.

The structure of the paper is as follows: in the next section the denoising filter is reviewed by presenting some related work and its mathematical foundations; also, algorithmic skeleton solutions are introduced and reviewed. Section 3 shows the performance of the sequential implementation of the described filter, while section 4 and section 5 present, respectively, the parallel implementation of the proposed algorithm over heterogeneous architecture and the experimental results. Finally, in the last section concluding remarks are provided.

2. Background

In this section the image restoration process will be introduced, by presenting its mathematical foundations and some related work.
Then an overview of state-of-the-art structured parallel programming frameworks is given, together with an introduction to the FastFlow framework. Finally, the original FastFlow-based image denoising application, which is the forerunner of the video denoiser, is described.

2.1 Variational Image Restoration: Related Work

In the past fifteen years a large number of methods have been proposed to deal with salt and pepper noise (and, more generally, impulse noise) from digital images [5]. Most of these methods employ order statistic filters that exploit the rank-order information of an appropriate set of noisy input pixels. The median filter is the most popular non-linear filter for removing impulse noise, because of its good denoising power [8] and its computational efficiency [22], but it affects image detail while removing noise. This issue has generally been addressed by filtering techniques based on median filter modifications[25, 36]. However, the performance of median filtering based approaches is unsatisfactory in suppressing signal-dependent noise [35] when the noise percentage is high (more than 50%). To achieve a good compromise between the image-detail preservation and the noise reduction, an impulse detector must be used before filtering. Several types of impulse detectors exist: the most famous is the progressive switching median (PSM) [33]. Machine learning approaches have also been widely used in recent years, e.g. approaches relying on Bayesian networks [19], fuzzy logic [34] and neuro-fuzzy [37]. The filtering is then selectively applied to the noisy regions detected by the noise detector. To the best of our knowledge, one of the most effective algorithms for edge preservation in salt and pepper denoising has been proposed by Nikolova in [26]: it applies a variational method for image detail preservation that is based on a data-fidelity term related to the impulse noise. Based on this approach Chan et al. in [12] (called for simplicity Chan’s method) proposed a powerful filter capable of removing salt and pepper noise as high as 90%. Similar approaches to Chan’s method, aimed at improving the noisy detection step and at reducing the processing times, are those proposed in [9–11, 14, 19].

2.1.1 Detection of Outliers for Salt-and-Pepper Noise

Let $\hat{y}$ be the map of noisy pixels (obtained by applying the adaptive median filter classifier to the noisy image) which has a 1 in the corresponding position for a noisy pixel, and 0 for an uncorrupted pixel. Hence the set of noisy pixels $N$ (to which the restoration algorithm has to be applied) consists of the overall pixels of the original image $y$ whose values in the $\hat{y}$ map are equal to 1, i.e.: $N = \{(i, j) \in A : \hat{y}_{i,j} = 1\}$

The set of all uncorrupted pixels is $N^c = A \setminus N$, where $A$ is the set of all pixels. Note that, since pixels with colour different from 0 and from 255 are uncorrupted, the AMF filter [23] can be modified to exclude them with significant benefits in term of performance and false positive rate.

2.1.2 Variational Denoising

The problem of image restoration for edge preserving is an inverse problem solved by using regularization, where the restored image $u$ is obtained by solving the following optimization problem restricted to the set of the noisy pixels $N$.

$$\min_{u \in N} F(u) = \alpha \int R(u) + \beta \int D(u, d)$$

where $d$ is the image corrupted by the noise; $D(u, d)$ is the data-fidelity term which is related to the kind of noise and provides a measure of the deviation between $d$ and the output image $u$; and $R(u)$ is a regularization term that uses a-priori knowledge for enforcing the solution and should be represented by a function that penalizes/removes only irregularities due to the affecting noise, thus ignoring high-level discontinuities (edges). $\beta$ and $\alpha$ are the regularization parameters that balance the effects of both mentioned terms. Among the functionals $F(u)$ (see [13]) for edge preserving proposed during the last fifteen years, we have selected the one proposed in [12] which has been shown to be very effective for Salt-and-Pepper noise. The same performance might not be guaranteed for a different type of noise. However, it is crucial to note that substituting $F(u)$ by a different functional does not require rewriting of the code, since the whole parallel denoising process is a higher-order function where $F(u)$ is a parameter. In the present case:

$$F_d|_N (u) = \sum_{(i, j) \in N} [\vert u_{i,j} - d_{i,j} \vert + \frac{\beta}{2}(S_1 + S_2)]$$

where

$$S_1 = \sum_{(m, n) \in V_{i,j} \cap N} \vert \varphi(u_{i,j} - d_{m,n})$$

$$S_2 = \sum_{(m, n) \in V_{i,j} \cap N^c} \varphi(u_{i,j} - u_{m,n})$$

where $N$ represents the noisy pixels set, $N^c$ the set of uncorrupted pixels, and $V_{i,j}$ is the set of the four closest neighbours of the pixel with coordinates $(i, j)$ and $d$ is the corrupted image. As in [12], we have used the following $\varphi$ function that provides the best trade-off between edge preserving and denoising: $\varphi(t) = \vert t \vert^\alpha$ with $1 < \alpha \leq 2$. The values of $\alpha$ and $\beta$ were, respectively, set to 1.3 and 4 in order to guarantee the trade-off between noise removal and edge preservation provided by the function $\varphi$.

A video stream is restored by independently filtering each of its frames. The restoration of a single frame follows an iterative process. At each iteration, for all outliers, its value is updated with the value $u$ which minimizes the functional (1). The iterative process is stopped in accordance with a quasi-Newton method [7] using no PSNR variation across successive iterations as fix-point. The presentation of algorithmic details, which is beyond the scope of the present work, can be found in [3].

2.2 Algorithmic Skeletons

Algorithmic skeletons have been around since the 90’s as an effective means of parallel application development. An algorithmic skeleton is a general-purpose, parametric parallelism-exploitation pattern [15]. Application programmers may instantiate skeletons (or compositions of skeletons) to encapsulate and exploit the full parallel structure of their applications. Business code may be based on the generic skeleton into a part of a parallel application.

MPI is often considered as a solution for writing efficient parallel applications [27]. The low-level approach advocated by MPI falls short in supporting performance portability, especially when hundreds or thousands of concurrent activities are involved and hybrid solutions have to be adapted (i.e. MPI+OpenMP). Applications must often be re-designed or manually tuned for each new platform by an expert parallel programmer. OpenMP [28] is a popular thread-based framework for multi-core architectures mostly targeting data parallel programming (although it is currently being extended to incorporate stream processing). OpenMP supports, by way of language pragmas, the low-effort parallelisation of sequential programs; however, these pragmas are mainly designed to
Video is processed as a stream of independent frames.

Figures 1. Two-phase denoiser.

Figures 2. Variational denoiser: architecture and refinement lattice.

exploit loop-level data parallelism (e.g. do independent). OpenMP does not natively support either farm or Divide & Conquer patterns, even though they can be simulated with lower-level features. Intel Threading Building Blocks (TBB) [32] is a C++ template library which provides easy development of concurrent programs by exposing (simple) skeletons and parallel data structures used to define tasks of computations. TBB is designed as an application-level library for shared-memory programming only; furthermore it does not provide any formal definition of its own skeletons to support global optimisations of the code.

Programming frameworks based on algorithmic skeletons have been recently introduced to alleviate the task of the application programmer when targeting data parallel computations on heterogeneous architectures. OpenCL is a parallel API provided for GPGPU programming, which allows the users to exploit GPUs for general purpose tasks that can be parallelised [24]. It is implemented by different hardware vendors such as Intel, AMD, and NVIDIA, making it highly portable and allowing the code written in OpenCL to be run on different graphical accelerators. OpenCL is quite low level, focusing on low-level feature management rather than high-level parallelism exploitation patterns. It has the capability to revert to the CPU for execution when there is no GPU in the system. Its portability makes it suitable for Hybrid (CPU/GPU) or cloud based environments. In Muesli [18] the programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons. StarPU [6] is focused on handling accelerators such as GPUs. Graph tasks are scheduled by its run-time support on both the CPU and various accelerators, provided the programmer has given a task implementation for each architecture. SkePU [17] provides programmers with GPU implementations of map and reduce skeletons.
and relies on StarPU for the execution of stream parallel skeletons (pipe and farm). MCUDA [31] is a framework to mix CPU and GPU programming. In MCUDA it is mandatory to define kernels for all available devices but the framework can not make any assumptions about the relative performance of the supported devices.

2.3 The FastFlow Programming Framework

The FastFlow parallel programming environment was originally designed to support efficient streaming on cache-coherent multi-core platforms. It is realised as a C++ pattern-based parallel programming framework aimed at simplifying the development of applications for (shared-memory) multi-core and GPGPUs platforms. The key vision of FastFlow is that ease-of-development and run-time efficiency can both be achieved by raising the abstraction level of the design phase. It provides developers with a set of parallel programming patterns (aka algorithmic skeletons) such as farm, divide&conquer, pipeline, map, reduce, and their arbitrary nesting and composition is supported [1, 4]. Map and reduce patterns can be run both on multi-cores and offloaded onto GPGPUs. In the latter case, the business code can include GPGPU-specific statements (i.e. CUDA or OpenCL statements).

The latest extensions of the FastFlow framework, aimed at supporting GPGPUs (via OpenCL) and CPU+GPGPU platforms, make it possible to easily port the application to heterogeneous platforms. OpenCL is certainly a suitable solution for programming hybrid architectures, as its specifications have been implemented by several hardware vendors, thus making the same OpenCL code runnable either on different GPGPU devices or on CPUs [30]. More details on the FastFlow OpenCL run-time can be found in [20].

FastFlow CPU implementation of patterns are realized via non-blocking graphs of threads connected by way of lock-free channels [4], while the GPU implementation is realized by way of the OpenCL bindings and offloading techniques [2]. Also, different patterns can be mapped onto different sets of cores or accelerators, thus, in principle, using the full available power of the heterogeneous platform. The business code running on a GPU can be further hand-tuned by exploiting OpenCL specific features. This kind of fine-tuning might bring significant performance improvements but, in general, is difficult to automate at the FastFlow level, since it is very related to the business code and requires extensive low-level programming effort.

Note that FastFlow does not provide any automatic facility to convert C++ code into OpenCL code. Thus any parallel activity (e.g. a map body) that can be possibly mapped onto a GPGPU should be coded with OpenCL. FastFlow, however, helps this task with a number of features including:

- Integration of the same pattern-based parallel programming model for both CPUs and GPGPUs. Parallel activities running on CPUs can be either coded in C++ or OpenCL.
- Setup of the OpenCL environment.
- Simplified data feeding to both software accelerators and hardware accelerators (with asynchronous Host-to-Device and Device-to-Host data movements) [2].

In this work, we consider the extension of the two-phase image denoiser to a two-phase video denoiser.

2.4 A High-Level Pattern-Based Approach for Image Restoration

As we already stated above, the image restoration problem can be tackled by applying the variational filter independently to each frame of the image. In order to optimally parallelise this task, we propose a structured approach using high-level algorithmic skeletons.

2.4.1 Two-phase Edge-Preserving Image Restoration

Variational filtering is performed using a two-stage approach:

1. **Detect.** Accurate detection of the location of noise (the outlier candidates) using a parallel implementation of the original filter.
2. **Denoise.** Edge-preserving restoration of outlier pixels by way of an iterated application of a variational filter up to a global convergence criterion (e.g. no PSNR increment in the last step). The full knowledge of the outlier mask at this stage helps in weighting the optimization function on neighbour pixels.

The detect stage can be implemented by plugging the code of a standard sequential filter into a FastFlow map template (i.e. a higher-order parallel pattern implemented as a header-only C++ template), which provides all the additional code to partition the image; dispatch partitions to a (dynamically) malleable set of workers; and join partial results. The outlier mask is produced as the difference between the restored and the noisy images. FastFlow lock-less run-time support ensures minimal coherence traffic and a close-to-optimal speedup on cache-coherent multi-core.

The denoise stage, computationally more demanding, is implemented as a map-reduce pattern. As we shall see, this stage is implemented in OpenCL in such a way that it can be run on both CPUs, GPGPUs, and CPUs+GPGPUs. A C++ version of the same code is used as a baseline for comparison purposes. Also, due to the fact that the restoring code is almost independent of noise type, the kernel does not need to be recoded for different filters (possibly exhibiting different or dynamic convolution kernel stencils).

The two stages can be pipelined to guarantee high or real-time throughput on video applications and the full utilization of the coupled multi-core+GPU platform. Typically, the detect stage is fast, orders of magnitude faster than the variational denoise stage. The GPGPU’s higher computational power naturally balances higher computational requirements on the second stage. If needed, the FastFlow run-time is able to dispatch different frames to different GPU devices dynamically (autonomically) acquiring new devices during the application execution.

The design of the proposed schema is summarised in Fig. 1.

2.4.2 Effectiveness of Two-Phase Denoiser on Images

The denoising quality of the two-phase image restoration was tested on standard images of different size. To compare results with the literature, the results reported in this section are in relation to two 512x512 8-bit grayscale images: Lena (Fig. 3) featuring homogeneous regions and Baboon (Fig. 4) characterised by high activity. These images have been corrupted by 70% and 90% salt-and-pepper noise.

In [3] has already been demonstrated the good performance of the restoration algorithm for one single image (for both the Lena and Baboon images of size 512x512 with 90% of noise a performance improvement of more than 25x on a Intel 32-core platform has been achieved using FastFlow).

With respect to the quality of restoration, the low rate of false positives in the detection of noisy pixels achieved by the adaptive median filter brings an overall improvement of the quality of restoration as non-noisy pixels are not changed by the denoise phase. We noticed an improvement of both PSNR (Peak Signal-to-Noise Ratio) and MAE (Mean Absolute Error) with respect to state-of-the-art results reported in the literature, e.g. Chan’s method.
3. Video Denoising: a Skeleton-based Approach

We advocate the simple extension of the application working for images presented in [3] to video, and to heterogeneous platforms. We emphasise the word “simple”. In the long term, writing parallel programs that are efficient, portable, and correct must be no more onerous than writing sequential programs. To date, however, few parallel programming models have embraced much more than low-level libraries, which often require the architectural re-design of the application. This approach is unable to effectively scale to support the mainstream of software development where human productivity, total cost and time to solution are equally, if not more, important aspects.

3.1 Two-Phase Video Denoiser

The two-phase filter methodology naturally induces a high-level structure for the algorithm, which can be described as the successive application of two filters as described in Fig. 1. The two phases can operate in a pipeline in the case where the two-phase denoiser is used on a stream of images (e.g. in a video application). In addition, both filters can be parallelised in a data-parallel fashion. Let $\text{map} \ [a_0, a_1, \ldots] = [f(a_0), f(a_1), \ldots]$ and $\text{reduce} \ [a_0, a_1, \ldots] = a_0 \oplus a_1 \oplus \ldots$, where $\oplus$ is a binary associative operator, and $[a_0, a_1, \ldots]$ an array of pixels (e.g. the pixels of an image).

Detect is the first phase of the algorithm, where the Adaptive Median Filter is applied to the noisy image. Each pixel of the image is processed independently, provided the processing element can access a read-only halo of the pixel. The process is expressed sequentially over the elements of an array: each pixel of the image is analysed. A set of (candidate) noisy pixels is produced. The parallel processing here is exploited by way of the FastFlow map template, which can be easily added to the code using a ParallelFor high-level pattern provided by the FastFlow framework to accelerate for loops [16]. The ParallelFor pattern is used on the outer loop traversing the pixel matrix, which partitions the work on matrix strips. However, for small frame sizes, as those considered in the experimental section of this work, the detection phase has a negligible execution time w.r.t. the denoising phase, so the parallelisation of the Detect phase is not really necessary.

Denoise is the second phase of the algorithm. The variational method can be computed independently for each pixel (with the previous iteration halo). This step is iterated until a convergence criterion is reached. The convergence criterion is a global propriety of the image, and so it should be computed via an associative operator by reducing the difference between the last two iterations according to a given convergence criterion. The parallel processing is described as a loop of a sequence (i.e. functional composition) of map and reduce patterns. The map works with a fixed stencil of eight neighbours for each outlier pixel (except for borders). The reduce pattern computes the global convergence criterion, to be checked in the loop exit condition. Using the OpenCL back-end, it is possible to specify the way in which the underlying heterogeneous architecture has to be exploited for the computation.

In the sequential version, the detect stage exhibits linear execution time with respect to the total number of pixels. However, the detect phase, typically a simple convolution, is two–three orders of magnitude faster than a single iteration of the second step (denoise). The second phase shows a computational complexity of $O(n_{\text{noisy\,pixels}} \cdot n_{\text{iterations}})$, where $n_{\text{noisy\,pixels}}$ is the number of noisy pixels identified in the first step, and $n_{\text{iterations}}$ is the number of iterations required to reach one of the convergence criteria. On a fully CPU deployment, a sequential detect stage is able to sustain the throughput of the parallel denoise stage even

Figure 3. Lena: 70% & 90% of noise and restored version.

Figure 4. Baboon: 70% & 90% of noise and their restored versions.
in a large multicore platform. This is no longer true if the denoise stage is run on one or more GPGPUs.

4. Experimental Evaluation

All experiments reported in this section were conducted on an Intel workstation with 2 eight-core double-context (2-way hyperthreading) Xeon E5-2660 @2.2GHz, 20MB L3 shared cache, 256K L2, and 64 GBytes of main memory (also equipped with two NVidia Tesla M2090 GPGPU) with Linux x86_64.

Three main families of experiments are reported:

1. Performance for the denoise stage in the OpenCL version on both CPUs (Fig. 5) and GPU (Fig. 6) on different work group sizes (i.e. different parallelism degrees).

2. Elapsed execution time for the C++ version on CPUs varying the number of threads used for the entire application (Fig. 7).

3. Maximum performance on the given platform for different versions of the code (C++ against OpenCL) and different noise levels.

A brief description of the different cases tested using only CPUs and both CPUs and GPGPUs is reported in Table 1.

In Fig. 5 and Fig. 6 the performance of the denoise stage is analysed for different sizes of OpenCL working groups. Here, the farm implementing the denoise stage is set to a single worker, i.e. no parallelism across different frames is exploited. In this case, the FastFlow Emitter and Collector (see Fig. 2) have been removed. It can be noticed that the working group size affects actual performance only for low levels of noise, i.e. for small problems. In the case of deployment on CPUs this leads to very fine grained parallelism and thus a significant synchronisation overhead.

Figure 7 reports the execution time obtained running the FastFlow C++ version deployed onto CPUs again in the case where the farm implementing the denoise stage is set to a single worker, i.e. no parallelism across different frames is employed. The exploited parallelism in the denoise stage comes entirely from the map pattern. The maximum speedup obtained with respect to the sequential CPU time ranges from 10.8 for 10% noise to 11.4 for 90% noise.
It can be noticed that the executions reported in Fig. 5 and Fig. 7 exploit exactly the same parallelism patterns on the same platform. They are actually the OpenCL and C++ versions of the same FastFlow application (using OpenCL and C++ as guest languages). They can therefore be safely compared in terms of absolute performance in the same graph (the CPUs C++ and CPUs OCL bars in Fig. 8) where the minimum overall execution time for the entire video denoising is reported. However, the maximum performance reached by the OpenCL version with sufficient working groups to saturate all cores is very low (it is comparable with the plain C++ version with parallelism degree 2 for the denoise stage). Such low performance results were in part expected since the OpenCL code was optimized for GPGPUs and not for CPUs. As noticed in some recent research work [29], performance of OpenCL code on CPUs may be improved tuning several aspects such as: host-to-device and device-to-host memory copies, increasing the computation granularity and avoid explicit caching of data through local memory which introduces unnecessary overhead on multicore. This tuning is partly due to currently available OpenCL compiler technology, which is not yet fully mature.

Figures 8 and 9 compare all versions for maximum performance in terms of overall execution time of the whole restoration pipeline. As expected, mapping the denoise stage onto GPGPUs greatly accelerates the application. Also, multiple GPGPUs can be used with almost linear speedup. It is worth noticing that in the OpenCL case it is possible to deploy the denoise stage on both CPUs and GPGPUs, actually mapping different workers of the farm on different devices, thus fully using the aggregate computational power of CPUs and GPGPUs for the denoise stage. In order to illustrate this case, two different scheduling policies are presented in Fig. 8: an order-preserving strict round-robin (RR) policy and the auto-scheduling policy implemented using the on-demand (OD) FastFlow scheduling pattern. While the order-preserving policy can perform well on homogeneous environments, in this case it bounds the performance to the slowest device. On-demand policy manages to overcome this problem but at the price of some additional overhead due to frame buffering and reordering. The speed edge between CPUs and GPGPUs forces either suspension of the dispatching of frames to GPUs or storage of out-of-order frames in a buffer, inducing a high dynamic allocation pressure on the system and a jerky output stream.

It can be argued that, in the case of an ordered stream as the one produced by streaming frames of a video, the extra tuning effort needed for a proper deployment of a mixed CPUs and GPGPUs architecture for the denoising phase is not worth the performance gain (if any).

### 5. Concluding Remarks

In this work we have built upon earlier work on single images to produce a parallel video denoiser for heterogeneous platforms. The key result here is the ease with which the earlier system could be extended: in essence we were able simply to lift noise detection and denoising code from the earlier system and insert it in a new skeleton framework to turn image processing into vision processing. The ease with which this could be achieved is testament to the abstraction powers of skeleton programming.

Moreover, the extension of the skeleton framework – FastFlow – to accommodate CPUs and GPGPUs has allowed us seamlessly to migrate the vision processing system to a heterogeneous setting and easily experiment with various assignments of application components to CPU/GPU devices.

From a machine vision perspective, the resulting application has succeeded to demonstrate the usability of variational denoising, considered very effective but slow, also for real-time video streams. The experiments conducted have also highlighted several issues in development of a unified parallel programming framework for heterogeneous platforms that are worth further investigation, and thus can steer further research in the area of parallel programming models for heterogeneous platforms.

### Acknowledgement

This work has been supported by the EU FP7 grant IST-2011-288570 “ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems” and Compagnia di San Paolo project id. ORTO11TPXX “IMPACT: Innovative Methods for Particle Colliders at the Terascale”.

---

**Table 1.** Description of the deployments tested for the two pipeline stages of the video denoiser application on the considered platform. In this particular case, being the Detect stage much faster than Denoise stage, not all deployments reported in Fig. 2 have been tested (i.e. ones including 2, 3, 4, and 5).

<table>
<thead>
<tr>
<th>Experiment name</th>
<th>Deployment (Fig. 2)</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs C++</td>
<td>1</td>
<td>Detect C++/1CPU, Denoise C++/14CPUs</td>
</tr>
<tr>
<td>CPUs OCL</td>
<td>2</td>
<td>Detect C++/1CPU, Denoise OpenCL/14CPUs</td>
</tr>
<tr>
<td>OCL CPUs+GPGPU(RR)</td>
<td>3</td>
<td>Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, Round-Robin scheduling</td>
</tr>
<tr>
<td>OCL CPUs+GPGPU(OD)</td>
<td>4</td>
<td>Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, On-Demand scheduling</td>
</tr>
<tr>
<td>1 GPGPU</td>
<td>5</td>
<td>Detect C++/1CPU, Denoise OpenCL/1GPGPU</td>
</tr>
<tr>
<td>2 GPGPUs</td>
<td>6</td>
<td>Detect C++/1CPU, Denoise OpenCL/2GPGPUs</td>
</tr>
</tbody>
</table>

**Figure 9.** Maximum speedup (compared to sequential CPU version) for different application deployments and different data sets.
References


Pricing Multiple Option Contracts with Monte-Carlo Methods on GPU

Guoqing Zhang  
Tampere University of Technology  
Tampere, Finland  
guoqing.zhang@tut.fi

Juho Kanniainen  
Tampere University of Technology  
Tampere, Finland  
juho.kanniainen@tut.fi

Tapani Ahonen  
Tampere University of Technology  
Tampere, Finland  
tapani.ahonen@tut.fi

ABSTRACT
The purpose of this paper is to study how to use GPU to accelerate the valuation of multiple option contracts and give an example of software design with an acceptable speed. For demonstration purposes we use Heston model though the framework is model-free, and we use computational-intensive Monte-Carlo simulations to explore the computational power of GPU. The ArrayFun feature provided in Matlab is the primary interface we use in this paper to program on GPU. One contribution of the paper is the method of analysing the distribution of maturity times in option contracts to understand the regularity of option data. Regular data is more efficient to be parallelized on GPU. The paper also points out that the time of data-copying between GPU memory and CPU memory and the time of sequential execution on CPU should be the factors targeted for optimization. We also realize that two factors are correlated and could be optimized together. For the irregular distribution of maturity times, we recommend to have the calculation of option payoffs on CPU and use GPU for Monte-Carlo simulation only. The task division can bring some amount of performance improvement but it is restricted due to the decreasing proportion of parallel part in the whole program as suggested in the Amdahl’s law. The case study at the end of the paper supports the findings by running the designed program on a machine which has 2 Intel Xeon X5650 CPUs and 2 Nvidia Tesla M2090 GPUs.

Keywords
GPU, Option Pricing, Parallel computing, Monte-Carlo Simulation, Matlab

1. INTRODUCTION
Graphical Processing Unit (GPU) has gained its popularity in High Performance Computing (HPC) area due to the notable computational power of the hardware. However, there has been many cases where the amount of performance improvement brought by GPU implementations couldn’t reach expected levels. The reasons for that could be various but are mainly related to unsuitable designs of software or not familiar with features of the hardware. Comparing with the hardware design of a traditional CPU core, the core of GPU is light weight by skipping some of the hardware features that are normally available in CPU. The core of GPU focuses on arithmetic operations including floating point operations to originally support massive computations in image processing and nowadays also support general purpose computing. A high end GPU in market has more than hundreds of cores available, which make it suitable for parallel computing.

In financial engineering, option pricing is a popular topic where Heston model [9] is a well-known stochastic volatility model used for option pricing. Using Monte-Carlo method [8] to simulate the model has been proven to be beneficial for the cases where option payoffs not only depend on the final value of the underlying asset but also, for instance, depend on the average value of the underlying asset. However, the simulation of the method is very computational intensive [11]. Due to the fact that the simulation paths of Monte-Carlo method is independent from each other, it is natural to apply a parallel programming model when implementing Monte-Carlo method for performance improvement. Since GPU has been widely deployed in computers, it becomes an ideal choice. There have been some published researches concerning using parallel computing techniques to solve option pricing problems (see, [19], [7], [18], [17]), especially [19] which uses GPU.

As indicated in the title, the paper is to address multiple option contracts pricing on GPU. The multiple option contracts can be based on the same underlying asset with different maturity times or based on different underlying assets. The challenge of price multiple option contracts parallelly on GPU with Monte-Carlo methods is caused by irregularity of maturity times across option contracts, especially in Over-The-Counter (OTC) markets where maturity times can be arbitrary. The collection of sampling payoffs for option contracts on maturity is a time-consuming operation and halt the other simulation paths to wait. If the operation is required to be done many times, the communication overhead will be increased which undermines the performance.

The paper is to address these challenges by introducing a CPU-and-GPU hybrid implementation for multiple option contracts pricing. The paper suggests to analyse the data
of option contracts to understand the regularity of maturity across option contracts. The analysing result can be used to assist task division between CPU and GPU in a hybrid design. The major contributions are highlighted as below:

- The method to analyze the distribution of maturity time $T$ in option contracts is proposed to understand the regularity of option data.
- The time of data-copying between GPU memory and CPU memory $T_{data\_copy}$ and the time of sequential executions on CPU $T_{cpu}$ should be the factors targeted for optimization. They are correlated and could be optimized together.
- Calculating option payoffs on CPU and using GPU for Monte-Carlo simulation brings a good performance for irregular distribution of maturity time $T$.
- Using multiple CPU threads to control corresponding number of GPUs can scale up performance.

Matlab has been a handy tool for financial engineers because of its built-in support for finance modelling. In the recent Matlab release, the Parallel Toolbox in Matlab adds a new version of ArrayFun which is designed to be the programming interface for GPU. The usage of the GPU ArrayFun is otherwise similar to its CPU counterpart except that the GPU ArrayFun must be applied to data located in GPU memory. In this paper, the term ArrayFun is used to refer to GPU version of it unless it is explicitly mentioned to be the CPU version.

A function declared as ArrayFun applies the function to each element of input matrices or vectors in a streaming manner. Hence, all of the input matrices or vectors to an ArrayFun must share the same dimensions. The streaming feature of ArrayFun matches the hardware design of GPU nicely. A GPU is composed by a group of stream processors [14], each of which has tens of cores. In a stream processor, a series of operations to be applied to incoming data, normally a set of hardware instructions, are stored in the cache memory. The processor fetches a set of data from GPU memory. The data will be sent to all of the cores of the processor and each of them has been set to perform the same instruction fetched from the cache memory. In this way, GPU achieves data parallelism by processing multiple data at the same time. This kind of operation is common known as SIMD, namely Single Instruction Multiple Data. In an ArrayFun, the implementation of the function is the set of instructions to be executed while the input matrices or vectors are the data to be fetched from memory. A huge performance improvement could be achieved with this kind of design when the instruction set is fixed and the amount of data to be processed is big. We will use ArrayFun to implement Monte-Carlo simulation for Heston model on GPU.

The paper is organised as follows: Section II starts by introducing Heston Model and Monte-Carlo simulation in general and then discuss the possibility of solving the problem in parallel and how the distribution of maturity time $T$ can affect. Section III illustrates the design of GPU program by explaining the task division between GPU and CPU and the reason behind it, then introduces how to extend the design to use multiple GPUs when available. Section IV applies the GPU program designed in previous section to experimental data and compares the result with CPU implementation at the end. Seciton V draws the conclusions and summarises the future work.

2. PARALLELIZING THE VALUATION OF OPTION PRICES WITH HESTON MODEL

2.1 Heston Model

In this paper, we use a standard Heston model for option pricing [9], where the stochastic process of asset price under the risk-neutral measure follows

$$dS_t = r S_t dt + \sqrt{v_t} S_t dW_t^S,$$

where $r$ is risk-free interest rate and $dW_t^S$ is a Wiener process. The $v_t$ is also a stochastic process which is described by

$$dv_t = k(\theta - v_t)dt + \eta \sqrt{v_t} dW_t^v,$$

where the $dW_t^v$ is a Wiener process that is correlated with $dW_t^S$ in Eq. (1). Importantly, we could use here any other model to price options, but use here Heston model as an illustrative example. Especially non-affine models are computationally interesting because, unlike Heston model, they do not necessarily have a semi-closed form solution to price options and Monte Carlo methods are needed.

2.2 Monte-Carlo Simulation

Monte-Carlo simulation of a stochastic process is a procedure for sampling random outcomes for the process. When used to value an option, Monte-Carlo simulation uses the risk-neutral valuation. Paths for the underlying stock price are simulated in a risk neutral world to obtain the expected payoff. In risk-neutral world, the expected return on stocks is risk-free interest rate $r$.

There are certain advantages concerning using Monte-Carlo simulation for option pricing. One of them is that the method is feasible to adapt to any stochastic process. However, the major drawback of the method is computational intensity. A very large number of paths is required to estimate the option price with accuracy. There are variance reduction procedures intending to save computational time. In this paper, empirical martingale methods, antithetic variables, and control variates are used. Moreover, with Heston model, it is possible to use accurate discretization schemes.

The stochastic process defined in Heston model follows a Markov process where only the present value of stock price is relevant for predicting the future. This is aligned perfectly with the market efficiency theory [15] but introduces data dependency inside the process. The design of parallel program must not violate the dependency in order to ensure the correctness of the result. The simulation paths are independent from each other so there is no dependency between the processes.

Assuming an European option contract that depends on a stock $S_t$ with yearly dividend yield $d$ and initial stock price $S_0$ at time zero, the option has maturity time $T$ and strike
price \( K \). The risk-free interest rate is \( r \). The stochastic process of \( S_t \) is described by Eq. (1) and the volatilities are described by Eq. (2). The steps of pricing the derivative with Monte-Carlo simulation are as follows:

1. Reduce the present stock price by the present value of dividend payment during the life of the option to get \( S_0 \).
2. Simulate a path of \( S_t \) by using \( r \) as expected rate of return
3. Terminate the simulation path at maturity time \( T \) and calculate the payoff of the derivative using strike price \( K \).
4. Repeat steps 1 and 2 to get enough sample value of the payoffs.
5. Calculate the expected payoff in a risk-neutral world
6. Discount the expected payoff at risk-free rate \( r \) to get an estimate of the value of the option.

### 2.3 Parallelizing Monte-Carlo simulation

As discussed in previous section, Monte-Carlo simulation is computational intensive. In this section, we investigate how to do the simulation in parallel to improve the speed. The following three scenarios are studied:

- **Scenario A**: Pricing multiple European option contracts that are based on the same underlying asset and are quoted on the same time.
- **Scenario B**: Pricing multiple European option contracts that are based on the same underlying asset and are quoted on different times with different spot volatilities.
- **Scenario C**: Pricing multiple European option contracts that are based on different underlying stocks and therefore quoted with different spot volatilities.

Computationally, scenarios B and C are identical as all options can not be priced in the same run; in case B, initial spot volatilities are different due to different time of option quotations and in case C, every underlying asset has not only different spot volatilities but also different volatility dynamics with different parameter values. In scenarios B and C, we need to categorize option contracts into \( n \) subgroups if option contracts are based on \( n \) different assets or are quoted on \( n \) different times.

#### 2.3.1 Scenario A

We begin with investigating parallel programming for a scenario where multiple option contracts to be priced are based on the same stock asset and offered on the same day. This scenario is used, for example, in [19] but without Monte Carlo methods. The result of the section will be used as a basis to continue the discussion for pricing option contracts for different stocks in the next section.

The preconditions we set in the scenario are *same day* and *same stock*. When these conditions are true, Monte-Carlo simulation of multiple option contracts is almost the same as the steps we described in previous section for one option contract except for the differences listed below:

- Simulation needs to be terminated when reaching the maximum maturity time among the option contracts
- Samples of payoffs should be collected during the simulation process when there are options on maturity

The preconditions *same day* and *same stock* guarantee that the \( N \) simulation processes are identical to each other in terms of workload. Hence, the workload balancing is kept among simulation paths. Keeping workload balancing as much as possible is a crucial requirement for the design of parallel programs in order to achieve maximum amount of performance improvement [12]. Without workload balancing in place the fast path would have to wait for the slow one at each of the synchronization points, which undermines the performance. Graphic representation of the parallel processing with workload balance is shown in Figure 1, and the work flow of option pricing for Scenario A is shown in Figure 2. Synchronization points are inserted due to data dependencies in a Markov process as discussed before. Eq. (3) gives the estimate of total running time to value prices of options on GPU hardware.

\[
T_{\text{total}} = \text{MAX} \left( \begin{array}{c} T_1 \\ T_2 \\ T_3 \\ \vdots \\ T_N \end{array} \right) \cdot T_{\text{gpu}} + T_{\text{data_copy}} + T_{\text{cpu}} 
\]

where \( T_{\text{data_copy}} \) is sum of the time used for data copying between CPU memory and GPU memory during the simulation, \( T_{\text{gpu}} \) is the time used by GPU for getting \( N \) simulation paths one step forward, \( T_{\text{cpu}} \) is the time used by CPU for controlling and computing and \( T_x \) is the maturity time of option contract \( x \).

The \( T_{\text{data_copy}} \) in Eq. (3) is sum of the time spent in synchronization points as shown in Figure 1. The amount of \( T_{\text{data_copy}} \) depends on the design of parallel program and also on the hardware. In Scenario A, the time consumption for synchronization points without options on maturity is equal to zero while the other synchronization points need certain amount of time for moving data between CPU memory and GPU memory. The \( T_{\text{data_copy}} \) is one of the key factors to minimize in order to achieve high performance. In practice, the minimization of \( T_{\text{data_copy}} \) means to avoid data copying as much as possible.

#### 2.3.2 Scenario B and C
When pricing multiple option contracts based on different underlying stocks or quoted on different time in parallel, we need to categorize option contracts into subgroups of Scenario A. The option contracts based on the same stock but quoted on different time are treated as different subgroups.

As discussed before, the work load balance is a decisive factor of performance. However, in Scenarios B and C, the work load balance can’t be always guaranteed as in Scenario A but depends on the feature of option contracts in question. Figure 3 is the graphic representation of valuing multiple options in Scenario B and C. Options with different quotation times (Scenario B) used in several research papers in Finance (see, for example, [2–4, 6]), though typically without superior computing. Recently, some papers with Scenario B type data with distributed computing are published [16].

In Figure 3, we assume that the option contracts in question are categorized into $n$ subgroups from $S_1$ to $S_n$. The subgroups are valued in parallel and each of them will carry on the Monte-Carlo simulation for Scenario A. The maturity times in a subgroup, marked as $T$ in Figure 3, is the factor that affects the work load balance of the parallel program.

By plotting distribution graph of $T$ of all subgroups, we could examine the regularity of the option data. A regular $T$ plot will achieve good performance on GPU. The Figure 4 shows an example of a regular distribution of $T$. In reality, the distribution plot could be irregular as shown in Figure 5. In this case, the performance of computing on GPU will be undermined. As discussed before, the GPU is a SIMD hardware where same set of operations are applied to multiple sets of data to achieve data parallelism. When simulation reaching a value of $T$, the samples of payoffs should be collected to calculate the expected payoffs for the options on maturity. A irregular $T$ plot means that the operations to calculate estimated payoffs could only apply to part of data, which may not be able to gain wanted amount of performance improvement on GPU when the amount of applicable data is not big enough. Moreover, the maximum of $T$ for a subgroup, marked as $Max(T_{Sx})$ in Figure 3, defines how many steps are needed in a path of Monte-Carlo simulation, namely the number of loop iterations in Matlab code. So a irregular $T$ plot will lead to uneven length of simulation paths.

For irregular $T$, a compromise such like adjusting the work load between GPUs and CPUs is needed to achieve an acceptable result. We will have more discussion on this topic in Section III when addressing the design of option pricing program for GPU.

3. DESIGN OF GPU PROGRAM FOR OPTION PRICING

To design a program for GPU, it is beneficial to understand features of the hardware. GPU hardware was originally designed for massive arithmetic operations in image processing and has evolved to general purpose GPU (GPGPU) nowadays. So unlike general purpose processor CPU, it is more suitable for data-flow computation rather than control-flow logic. In control-flow code, a program selects the next piece of code to execute based on the result of condition checking affected by input data.

Due to the SIMD feature, the scheduler of GPU hardware
will schedule $N$ threads at the same time where $N$ is equal to the number of cores that a stream processor has. Each of the threads will be assigned to the same instruction but different input data. When executing control-flow code, the result of condition checking is likely to be different due to different input data. The scheduler of GPU has to separate the $N$ threads for the following execution until the execution merges back. The separation is necessary even though only one from the $N$ threads has a different result of condition checking, which makes the rest $N-1$ threads do nothing. So control-flow code doesn’t fit well on GPU hardware from performance perspective. Because of this, there is almost no exception that a GPU program will be co-designed together with a CPU program which is responsible for the control-flow logic. We follow the same routine in our GPU program for option pricing. The design will focus on Scenario B and C that are more generic cases. Scenario A is a just a special case of Scenario B so the GPU program design for Scenario B will work on Scenario A as well.

3.1 Task division between CPU and GPU

As shown in Figure 2, the Monte-Carlo simulation for option pricing has condition checking at the end of each simulation step for option contracts on maturity. This part is assigned to CPU for the reason discussed above. Besides the condition checking, CPU is also responsible for moving option data and payoff samples between CPU memory and GPU memory and is responsible for calculating expected payoffs for option contracts on maturity as well due to the irregular maturity times in the experimental data we have. The irregularity of maturity times causes inadequate data for GPU to gain enough performance improvement to compensate the time used by data moving and GPU functions invoking. Hence, paralleling the computation of expected payoffs on GPU wouldn’t gain performance improvement. The details of experimental data will be discussed in Section IV.

GPU is responsible for the Monte-Carlo simulations of all subgroups. The ArrayFun for GPU is implemented according to the SDE of Heston model as shown in Eq. (1) and Eq. (2). Figure 6 shows the details of the task division between CPU and GPU. The estimate of total running time is shown in Eq. (4).

$$T_{total} = \sum_{x=1}^{n} \text{MAX}(T_{S_x}) + T_{gpu} + T_{data\_copy} + T_{cpu}$$  \hspace{1cm} (4)$$

where $T_{data\_copy}$ is sum of the time used for data copying between CPU memory and GPU memory during the simulation, $T_{gpu}$ is the time used by GPU for getting $N$ simulation paths one step forward, $T_{cpu}$ is the time used by CPU for controlling and computing and $T_{S_x}$ is the maximum maturity time of option contracts in Subgroup $x$.

Having the calculation of payoffs on CPU increases the proportion of the sequential part in the whole program so does its execution time $T_{cpu}$ in Eq. (4). According to Amdahl’s law [10], the amount of performance improvement brought by a parallel program is decided by Eq. (5).

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \hspace{1cm} (5)$$

where $P$ is the proportion of a program that runs in parallel and $N$ is the number of cores used.
If the proportion of the sequential part in a parallel program increases, the amount of performance improvement is limited to $\frac{1}{1-P}$ no matter how many cores are used. Section IV will give some experimental results to prove this.

The issue of inadequate data in the payoff calculation could be solved by accumulating samples of payoffs for some steps during simulation process until the amount of options on maturity is big enough for GPU to gain performance improvement. This approach can increase the proportion of the parallel part in the whole program, and at the same time reduce $T_{\text{data}}$ consumed by data moving between CPU memory and GPU memory. However, there are two issues to be considered before using this method. The first issue is the limitation of memory capacity on GPU hardware as holding the samples of payoffs for some steps would consume huge memory space when the number of simulation paths is big. The second issue is related to the system requirements. If the speed of valuing individual option is more important than the total throughput in system requirements, this method can not be used since holding the data for some steps means delay for an individual option. We don’t use the approach due to limited memory capacity in the testing environment, which allows maximum 200 megabytes GPU memory per user according to the policy.

3.2 Use multiple GPUs for option pricing

When there are multiple GPUs available in a system, the architecture design in Figure 5 could be replicated to use multiple GPUs for acceleration. Assuming there are $N$ GPUs available, the replication means creating $N$ threads running concurrently in CPU side and each thread is associated with one GPU. The $N$ threads execute the same program shown in Figure 5 and option data will be partitioned into $N$ blocks to handle in parallel. Another advantage is that the sequential CPU code could also be run in parallel by the CPU threads associated to GPUs. By using multiple CPU threads to control corresponding number of GPUs, the scalability of performance can be achieved with the expense of having more GPUs.

It is important to know that when the number of running threads is not bigger than the number of cores in CPU, the threads will be distributed across the cores automatically by operating system and there is no contention for CPU time among threads as each of the threads is running on a dedicated physical core separately. When the number of running threads is bigger than the number of cores in CPU, there will be threads sharing the CPU time of the same core and the other resources associated with the core. Performance wise, it is not so good as the case of dedicated physical core.

In Matlab, the threads are created via matlabbpool utilities from Parallel Computing Toolbox and are named as worker. The Parallel Computing Toolbox has a built-in local cluster feature to schedule local workers on a single machine. Matlab also provides a separate package called Distributed Computing Server to schedule workers in a network cluster but it is out of the scope of this paper. The maximum number of local workers in Matlab is version dependent as shown in Table 1.

### Table 1: Version dependant maximum local workers in matlab

<table>
<thead>
<tr>
<th>Matlab Version</th>
<th>Maximum Local Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before R2009a</td>
<td>4</td>
</tr>
<tr>
<td>R2009a</td>
<td>8</td>
</tr>
<tr>
<td>R2011b and afterwards</td>
<td>12</td>
</tr>
</tbody>
</table>

4.1 Experimental data

We used 15 years of S&P 500 option data, obtained from OptionMetrics, which is a widely used high-quality source. The experimental data contains daily option trading information collected from option market from the beginning of 1996 to the end of 2010. Each record includes a number of option contracts that was traded on a day during that period based on the same stock. There are 3710 records (days) in total having at least one option contract. Each option contract has information such like option identity, maturity time, strike price, close price of the underlying asset, yearly interest rate and yearly dividend yield. It also includes the best bid and best offer of the option, which can be used to calculate the real market price for the option and used to justify the accuracy of the modelling result. The spot volatility is estimated from VIX index using the relation (see, for example, [1,5])

$$VIX_t^2 = \theta \left( 1 - \frac{1 - e^{-k\tau}}{k\tau} \right) + \frac{1 - e^{-k\tau}}{k\tau} v_t,$$

where $\tau$ denotes the number of days in a month. In addition, as specified in Eq. (2), $k$ is the volatility mean revision, $\theta$ the long-term squared volatility and $v_t$ the spot volatility at time $t$.

By applying the previous terminology, we are dealing with scenario B and have 3710 subgroups to be processed in parallel from the experimental data. The distribution plot of $T$ of the 3710 subgroups is shown in Figure 7 and obviously it is not regular. Since we don’t have ideal work load balance preferred by a parallel program in the experiential data, the performance improvement on GPU will be restricted by that. The following section will give detail figures on performance improvement comparing with existing multi-core CPU implementation.

![Figure 7: Distribution plot of $T$ in experimental data](image)
The machine we use for testing has 2 6-core Intel Xeon X5650 CPU-chips and 2 Nvidia Tesla M2090 GPU boards [13] attached. The Table 2 has details information on figures of the hardware. The operating system is Scientific Linux 6.4 which Matlab release 2012b installed. Even though the total size of memory on GPU is 6 GB but the available memory allocated for this experiment is restricted to 200 megabytes for the reason we mentioned before.

4.2 Performance Evaluation

The performance evaluation is the section will be carried out by comparing the computational speed of option pricing on the following 4 kinds of hardware configurations:

- 2 Intel Xeon X5650 cores plus 2 Nvidia Tesla M2090 GPUs
- 1 Intel Xeon X5650 core plus 1 Nvidia Tesla M2090 GPU
- 4 Intel Xeon X5650 cores
- 12 Intel Xeon X5650 cores

The 12 Intel Xeon X5650 cores provided by 2 Intel Xeon X5650 CPU-chips in the testing machine are the maximum number of CPU cores we can use in the evaluation due to the limitation of both local cluster in Matlab Release 2012b and hardware resources we have.

4.2.1 Case 1

In this case, we value the option contracts offered in the first 70 days from experimental data. The distribution plot of maturity time $T$ is shown in Figure 8 and the speed comparison plot is shown in Figure 9.

The result shows that using 2 Nvidia Tesla M2090 GPUs to value the 70 options gives the best performance. The number of simulation in this case was chosen intentionally high to increase the proportion of executing time of the parallel program on GPU. The percentage of parallel program on GPU in the total executing time of the program is shown in Figure 10.

4.2.2 Case 2

In this case, we value all of option contracts from the experimental data, namely all option contracts offered across the 3710 days. The distribution plot of maturity time $T$ was shown in Figure 7 in the beginning of the section. The speed comparison plot is shown in Figure 11.

<table>
<thead>
<tr>
<th>Hardware configuration of testing machine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
</tr>
<tr>
<td>Number</td>
</tr>
<tr>
<td>Model</td>
</tr>
<tr>
<td>Frequency</td>
</tr>
<tr>
<td>Memory</td>
</tr>
</tbody>
</table>

Figure 8: Distribution plot of $T$ of options data offered during 70 days

Figure 9: Speed comparison for Case 1

Figure 10: Percentage of parallel program execution time on GPU in Case 1

The result shows that 12 Intel Xeon X5650 cores beats 2 Nvidia Tesla M2090 GPUs in speed when valuing much more option contracts than in Case 1. It is due to the fact that the proportion of executing time of the parallel program on GPU in the total execution time of the program has decreased dramatically as shown in Figure 12 when comparing with...
5. CONCLUSION

Through this paper, we have discussed the design of Monte-Carlo simulation of Heston model on GPUs for option pricing. It is a challenging task to design a parallel program on GPU for option pricing due to the internal dependency of the Heston stochastic process and the irregular distribution of maturity times in the experimental option data we have. The proposed method of checking the distribution of maturity times gives information on the regularity of the option data, which indicates how efficient the option data can be processed in parallel in GPU. For irregular option data, calculating the option payoffs on GPU would undermine the performance due to the inadequate amount of options on maturity after each simulation step. A reasonable work division between CPU and GPU as suggested in the example design can bring certain amount of performance improvement for the case. However, when the proportion of the parallel part in the whole program decreased, the amount of performance improvement given by the GPU program will decrease as described by the Amdahl’s law. The case study at the end of the paper has proven the findings with figures showing that 2 Nvidia Tesla M2090 GPUs couldn’t beat 12 Intel Xeon X5650 cores when the proportion of the parallel program running on GPU is decreased to less than 20% of the whole program.

As pointed out in the paper, the factors $T_{data\_copy}$ and $T_{cpu}$ should be focused for optimization of a GPU program. For option pricing case, we have already mentioned that accumulating the sample data for a couple of steps could minimize both of the factors at once with the expense of using more memory space. The future work would be to continue the investigation on possible ways of optimization relevant to decreasing $T_{data\_copy}$ and $T_{cpu}$.

6. ACKNOWLEDGMENT

The research leading to these results has received funding from the ARTEMIS JU under grant agreement number 295371.

7. REFERENCES


GPGPU implementation of a tridiagonal matrix solver using Thomas method

Akiyoshi Wakatani
Faculty of Intelligence and Informatics Konan University
wakatani@konan-u.ac.jp

Abstract

GPGPU (General Purpose computing on GPU), which applies GPU (Graphics Processing Unit) to general purpose processing as well as graphics processing, has been recently paid to attention very much, because it produces the enormous computing power. In this paper, a tridiagonal matrix solver based on Thomas method, which consists of a non-linear recurrence equation and linear recurrence equations, is implemented on GPUs by using CUDA and the effectiveness of the rearrangement of array configurations is evaluated for the GPGPU implementation with Kepler GPU architecture.

Although the recurrence equations are hard to parallelize by nature, the P-scheme algorithm, which has been originally developed for distributed memory multicomputers, is applied to the non-linear recurrence equation and the tridiagonal matrix solver based on Thomas method. Moreover the rearrangement of array configurations is utilized as to use the coalesced communication for the data transfer between the global memory and the processing cores.

Our experimental results show that our method achieves a speedup of 16.84 in the case of the array size of $2^{25}$, whereas the conventional method (parallel cyclic reduction) achieves at most a speedup of 1.49.

Categories and Subject Descriptors CR-number [subcategory]: third-level

Keywords multithreading, tridiagonal solver, GPU, multicore, CUDA

1. Introduction

Recently, the peak performance of GPU (Graphics Processing Unit) has increased very much and outperforms the performance of general-purpose processors. For example, while the theoretical peak performance of Intel Xeon is about 500 GFLOPS, the performance of GeForce GTX 680 GPU of NVIDIA reaches over 3 TFLOPS (=3,000 GFLOPS). Since past GPUs consisted of special-purpose hardware, they were used only for image processing and graphics processing. However, recent GPUs are composed of general-purpose unified shaders, so by using CUDA (Compute Unified Device Architecture) [1], they are used for general-purpose processing like numerical calculations as well as graphics processing. This kind of computation is called GPGPU (General-Purpose computing on Graphics Processing Units).

A tridiagonal matrix can be solved by using recurrence equations and several tridiagonal matrix solvers have been implemented on GPUs. Kass et al. used an ADI method for an approximate depth-of-view computation and solved the tridiagonal matrix by using a CR (Cyclic Reduction) method on a GPU [4]. Zhang et al. applied four methods (CR, parallel CR, RD (Recursive Doubling) and hybrid) to the tridiagonal matrix solver on the GPU and found that the hybrid method achieved the best performance [5]. Goddeke and Strzodka proposed a mixed precision iterative solver using the CR method and implemented it on the GPU. They found that the resulting mixed precision schemes are always faster than double precision alone schemes, and outperform best-tuned CPU solvers [6].

Although Thomas method [7] is a basic algorithm for solving a tridiagonal matrix, it is a sequential method by nature that consists of two forward substitutions and a backward substitution, and thus it is not easy to parallelize it straightforwardly. One of the forward substitution and the backward substitution are linear recurrence equations and another forward substitution is a non-linear recurrence equation. Since a prior art [8] deals with linear recurrence equations, we focus on the non-linear recurrence equation of Thomas method in this paper and implement it on GPGPU systems. The performance of the tridiagonal solver using our methods is also described.

The rest of this paper is organized as follows: Section 2 presents the tridiagonal matrix solver and the non-linear recurrence equation. We also mention our method called P-scheme in Section 2. Section 3 presents the experimental method and discusses the results and Section 4 describes the related works. Finally Section 5 concludes this paper with a summary.

2. Non-linear recurrence equation

2.1 Tridiagonal matrix solver

Let’s consider the following tridiagonal system of equations of $A \times x = c$ where $A$ is a tridiagonal matrix with $N \times N$ elements and $x$ and $c$ are vectors with $N$ elements. The system is given by

\begin{align}
x_0 & = c_0 \\
-b_i \cdot x_{i-1} + a_i \cdot x_i - b_{i+1} \cdot x_{i+1} & = c_i \quad (1 \leq i \leq N-2) \\
x_{N-1} & = c_{N-1}
\end{align}

where arrays $a$ and $b$ are non-zero elements of matrix $A$ and vector $c$ is given in advance, array $x$ is an unknown array and $N$ is the number of elements of the arrays.
It is known that the system given by Equations (1), (2) and (3) can be deterministically solved by Thomas method, which utilizes two auxiliary arrays \( p \) and \( q \).

\[
p_0 = 0, \quad q_0 = c_0
\]

\[
p_i = \frac{b_i}{a_i - b_i \cdot p_{i-1}} \quad (1 \leq i \leq N-2)
\]

\[
q_i = \frac{c_i + b_i \cdot q_{i-1}}{a_i - b_i \cdot p_{i-1}} \quad (1 \leq i \leq N-2)
\]

\[
x_i = x_{i+1} \cdot p_i + q_i \quad (1 \leq i \leq N-2)
\]

The above procedure consists of a forward substitution (Equation (5)) and a backward substitution (Equation (7)). On parallel computers, the procedure cannot be straightforwardly parallelized due to the data dependency that resides on both the forward and backward substitutions. So, the Thomas method is not suitable for parallel processing. Suppose that the number of processor is \( P \), \( N = P \cdot M + 2 \) and arrays are block-distributed. Processor \( k \) \((k = 0 \sim P - 1)\) is in charge of \( M \) elements of arrays from \( k \cdot M + 1 \) to \( k \cdot M + M \), thus can be calculated on processor \( k \) only after \( p_{(k-1) \cdot M} \) is calculated on processor \( k - 1 \). Meanwhile, \( x_{(k+1) \cdot M - 1} \) is calculated on processor \( k + 1 \). These data-dependencies completely diminish the possibility of parallel computing.

### 2.2 P-scheme

We proposed a parallel and scalable algorithm, called P-scheme, which was originally developed for distributed-memory multicomputers [9]. We focus on array \( p \) on Equation (5) and explain how the P-scheme works for it. Note that the equation for array \( p \) is a non-linear recurrence equation and the equations for arrays \( q \) and \( x \) are linear recurrence equations. The P-scheme for the linear recurrence equation has been studied in [8], so we focus on the non-linear recurrence equation. Then we assume that \( p_{j-1} \) and \( p_j \) can be expressed by the following equation:

\[
\frac{\beta_j + \gamma_j \cdot p_j}{\delta_j + p_j} = (-1)^j \cdot p_{j-1} \quad (j > 0)
\]

where \( \beta_j, \gamma_j \) and \( \delta_j \) are auxiliary arrays that are defined below. Thus, \( \beta_j, \gamma_j \) and \( \delta_j \) can be determined by the following system of equations:

\[
\beta_1 = 1, \quad \gamma_1 = \frac{a_1}{b_1}, \quad \delta_1 = 0
\]

\[
\beta_{j+1} = \frac{1}{\gamma_j} \left( \delta_j + (-1)^{j+1} \cdot \frac{a_{j-1}}{b_{j-1}} \cdot \beta_j \right)
\]

\[
\gamma_{j+1} = \frac{1}{\gamma_j} \left( 1 + (-1)^j \cdot \frac{a_{j-1}}{b_{j-1}} \cdot \gamma_j \right)
\]

\[
\delta_{j+1} = \frac{\beta_j}{\gamma_j}
\]

It should be noted that \( \beta_j, \gamma_j \) and \( \delta_j \) are independent of \( p_j \); Hence \( p_i \) can be determined by using only \( p_0 \) as follows:

\[
p_i = \frac{-\beta_i + (-1)^i \cdot p_0 \cdot \delta_i}{\gamma_i - (-1)^i \cdot \beta_i}
\]

Therefore, if \( \beta_i, \gamma_i \) and \( \delta_i \) are calculated in advance, \( p_i \) can be directly determined just after \( p_0 \) is determined. By using the above relation, we propose a scheme called P-scheme (Pre-Propagation scheme), which consists of three phases. First of all, every processor simultaneously starts its calculation of \( \beta_i, \gamma_i \) and \( \delta_i \). This is called pre-computation phase. After that, processor 0 can directly determine \( p_M \) from \( p_0, \beta_M, \gamma_M \) and \( \delta_M \) and sends \( p_M \) to processor 1. After receiving it, processor 1 can directly determine \( p_{2 \cdot M} \) from \( p_M, \beta_M, \gamma_M \) and \( \delta_M \) and sends \( p_{2 \cdot M} \) to processor 2 and then processor 2 can directly determine \( p_{3 \cdot M} \) from received \( p_{2 \cdot M} \) and its auxiliary arrays and sends \( p_{3 \cdot M} \) to processor 3 and so on. This is called propagation phase. It should be noted that processor \( k \) can determine \( p_{k \cdot M + 1} \) without calculating \( p_{k \cdot M + 1} \). Finally, all processors can determine \( p_{k \cdot M + 1} \) \((i = 1, 2, ..., M - 1)\) by using received data \( p_{k \cdot M} \). This is called determination phase.

The pre-computation and determination phases can be completely parallelized. Meanwhile the propagation phase is still sequential but the data to be exchanged is very slight, like just one data, so the communication cost is also expected to be very slight. The cost of the propagation phase is in proportion to the number of processors. Thus the total execution time is estimated by \( O(\frac{N}{P}) + O(P) \) (as shown in Figure 1). It is general that \( O(P) \) is absolutely less than \( O(N) \), because \( P \) is supposed to be much less than \( N \).

![Figure 1. Timing chart of P-scheme](image)

Although the pre-computation and determination phases can be parallelized, the computational complexity is larger than the original substitution given by Equation (5). Since the original contains 1 multiplication, 1 division, 1 addition, 3 loads and 1 store and the P-scheme contains 4 multiplications, 3 divisions, 3 additions, 6 loads and 4 stores, P-scheme must carry out over twice more computation than the original substitution. Execution times of the original substitution and P-scheme on PC (Pentium III (1 GHz), 1GB memory, GCC4.1.1 with O3) are shown in Figure 2. The graph shows that the execution time of P-scheme is about twice slower than that of the original substitution. However, since the P-scheme algorithm can be parallelized, the execution time of the P-scheme on parallel systems is much smaller than the original algorithm.

![Figure 2. Comparison of execution times](image)

For linear recurrence equations, the P-scheme can be easily applied as shown in [8]. Consider the equation \( w_i = s_i \times w_{i-1} + t_i \) \((1 \leq
In the precomputation phase, each processing core independently calculates \( w_i = x_i + w_{i-1} + t_i \) and \( a_i = \prod_j s_j \) for sub-vectors. Then, in the propagation phase, \( w_i \) is corrected as \( w_i = w_i + a_i \times w_{i-M} (M = N/P) \). Finally, in the determination phase, the rest of \( w_i \) is corrected independently on processing cores. Therefore, linear recurrence equations can be also parallelized by using the P-schema method although the computational cost increases at some degree.

3. Experiments and discussion

3.1 Experimental environment

The specification of the GPGPU system is summarized in Table 1.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel i7-3770 (3.4GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>8.0 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.6.32 (CentOS)</td>
</tr>
<tr>
<td>GPU</td>
<td>GTX 680</td>
</tr>
<tr>
<td>Compute capability</td>
<td>3.0</td>
</tr>
<tr>
<td>CUDA</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Table 1. Experimental system

<table>
<thead>
<tr>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
<th>32K</th>
<th>64K</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.76</td>
<td>1.03</td>
<td>7.62</td>
<td>12.36</td>
<td>18.52</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Effectiveness of the rearrangement of array configurations

As mentioned before, the pre-computation and determination phases can be completely parallelized between threads, but the global memory accesses are done in either the coalesced communication or the non-coalesced communication, which depends on the array assignment. On the P-schema algorithm for distributed memory computers, the array assignment. On the P-scheme algorithm for distributed memory computers, the array should be rearranged (configuration 2) in order to utilize the coalesced communication, that is, thread 0 is in charge of array elements 0, 1 and 2, thread 1 is in charge of array elements 3, 4 and 5, and thread 2 is in charge of array elements 6, 7 and 8, because this configuration (configuration 1) can enhance the locality of memory accesses and the efficiency of the cache memory.

In order to cope with this difficulty, array elements that are accessed simultaneously should be rearranged so that they are adjacent to each other.

\[
\begin{align*}
\text{Rearrangement, double} & : \quad w_{iP:s+j} = w_{iP:s+j} (0 \leq i \leq s-1, 0 \leq j \leq P - 1) \\
\text{Rearrangement, float} & : \quad w_{iP:s+j} = w_{iP:s+j} (0 \leq i \leq s-1, 0 \leq j \leq P - 1)
\end{align*}
\]

where \( s = \frac{N}{P} \). Namely, this rearrangement is equal to the transposition of a \( P \times s \) two-dimensional array into a \( s \times P \) two-dimensional array.

In order to cope with this difficulty, array elements that are accessed simultaneously should be rearranged so that they are adjacent to each other.
arrangement of array configurations is 8.52 when \( P \) is 8192 and the speedup without the rearrangement of array configurations is just 0.76. Therefore, the effectiveness of the rearrangement of array configurations is very clear and it is indispensable in order to achieve the maximum speedup. As expected, the maximum speedup for single precision floating point numbers outperforms the maximum speedup for double precision floating point numbers. So, the implementation using single precision floating point numbers is preferable.

3.4 Experiment 2

![Figure 5: Results on GPGPU systems](image)

Figure 5 shows the difference of the speedups of the non-linear recurrence equation solver for different array sizes. As shown in [8], three phases of the P-scheme method are in proportion to \( N/P \), \( P \) and \( N/P \), respectively. According to our results, the parameters, \( \alpha \), \( \beta \) and \( \gamma \) are approximately measured as follows:

\[
\alpha : \beta : \gamma = 1.66 : 0.46 : 0.65 \quad (\text{float}) \quad (14)
\]

\[
\alpha : \beta : \gamma = 2.3 : 0.69 : 1.07 \quad (\text{double}). \quad (15)
\]

The optimal parallelism \( P_{opt} \) is determined as follows:

\[
P_{opt} = \sqrt{\frac{(\alpha + \gamma)N}{\beta}}. \quad (16)
\]

So, when single precision floating point numbers are used, \( P_{opt} \) is about 6300, 8900 and 12700 for \( N \) of \( 2^{23} \), \( 2^{24} \) and \( 2^{25} \), respectively. When double precision floating point numbers are used, \( P_{opt} \) is about 5000, 7700 and 11000, respectively.

On the other hand, our experiments show that the parallelism where the maximum speedup is achieved is measured as 4096, 8192 and 8192 for \( N \) of \( 2^{23} \), \( 2^{24} \) and \( 2^{25} \), respectively, when single precision floating point numbers are used. When double precision floating point numbers are used, the parallelism is measured as 4096, 4096 and 8192. Since these results are almost identical to the estimation, the validity of our policy to determine the optimal parallelism is empirically confirmed.

3.5 Experiment 3

We implement a tridiagonal matrix solver based on Thomas method on GPU by integrating CUDA programs of non-linear recurrence equations mentioned above and CUDA programs of linear recurrence equations [8].

Figure 6 shows the elapsed time of the tridiagonal matrix solver with different parallelisms \( P \) for the cases of \( N = 2^{23}, 2^{24}, 2^{25} \). It also shows speedups of the P-scheme and PCR (Parallel Cyclic Reduction) method [5], which is one of conventional parallel methods for solving tridiagonal matrices and recurrence equations. The computational complexity of the PCR method is \( O(N \log N/P) \) and is larger than the P-scheme method. In the figures, the elapsed times of the non-linear recurrence equation (Eq. (5)), the linear recurrence equation (Eq. (6)) and the linear recurrence equation (Eq. (7)) are represented as non-linear, linear/non-const. and linear/const., respectively. Note that single precision floating point number data is used and the rearrangement of array configurations is applied in this experiment.

As mentioned in Experiment 2, the elapsed time of the non-linear recurrence equation is minimized with the optimal parallelism and the elapsed time of the linear recurrence equation is also minimized as mentioned in [8]. Thus, the elapsed time of the tridiagonal matrix solver that consists of the recurrence equation solvers can be also minimized with the optimal parallelism. Namely, when \( N \) is \( 2^{23} \), the optimal \( P \) is 4096. When \( N \) is \( 2^{24} \) or \( 2^{25} \), the optimal \( P \) is 8192. The elapsed time of the linear recurrence equation (Eq. (6)) is slightly larger than the linear recurrence equation (Eq. (7)), because Eq. (6) has more computation than Eq. (7).

Next, we compare the elapsed times at the point of the optimal parallelism. For example, the elapsed times of non-linear of \( N = 2^{23} \), \( 2^{24} \) and \( 2^{25} \) are 7.39, 9.63 and 15.15, respectively. The difference between elapsed times of different sizes is not necessarily twice, because the total elapsed time includes the cost of the propagation phase that is irrelevant to \( N \). However, the difference between \( N = 2^{23} \) and \( 2^{25} \) is closer to twice than the difference between \( N = 2^{23} \) and \( 2^{24} \), so the overhead of the propagation phase relatively decreases as \( N \) increases.

Furthermore, the maximum speedup increases as \( N \) increases and it reaches 16.84 when \( N = 2^{23} \). The reason is that the overhead of the propagation phase relatively decreases and then the effectiveness of the parallelism is enhanced, as \( N \) increases. Meanwhile, the superiority of the P-scheme method to the PCR method is very clear as shown in the results of the comparison. Namely, when the array size is \( 2^{25} \), the speedup of the PCR method is just 1.49. Although our implementation of the PCR method is straightforward and it may be able to be improved with some optimizations, the main reason is that the computational complexity of the PCR method is larger than that of the P-scheme.

4. Related works

Generally, the first-order recurrence equation cannot be parallelized easily since the \( i \)-th element can be determined by using the \( (i-1) \)-th element. Several parallel algorithms, like CR (Cyclic Reduction) and RD (Recursive Doubling), are known as equation solvers which
can be directly applied on parallel computers [10], [11], [12], [13], [14].

As mentioned, a tridiagonal matrix can be solved by using recurrence equations, and several tridiagonal matrix solvers have been implemented on GPUs. Kass et al. used ADI method for an approximate depth-of-view computation and solved the tridiagonal matrix by using CR method on a GPU [4]. Zhang et al. applied four methods (CR, parallel CR, RD and hybrid) to the tridiagonal matrix solver on the GPU, and evaluated the performances to find that the hybrid method achieved the best performance among several implementations [5]. Goddeke and Strzodka proposed mixed precision iterative solvers using CR method, and implemented it on the GPU. They found that the resulting mixed precision schemes are always faster than double precision alone schemes, and outperform tuned CPU solvers [6].

When the size of the matrix is $N \times N$, the computational complexity of CR is $O(N)$ but it requires $2 \log_2 N$ synchronizations between processors. Meanwhile, parallel CR requires only $\log_2 N$ synchronizations but its total computational complexity is $O(N \times \log_2 N)$. On the other hand, since the sequential algorithm (Thomas method) consists of two recurrence equations (a forward substitution and a backward substitution) and both recurrence equations can be parallelized by using our method, those computational complexities are $O(N/P)$, $O(P)$ and $O(N/P)$, respectively. Our method also requires only two synchronizations for each recurrence equation. Then, we implement our method for recurrence equations on GPUs and evaluate the parallelism and the effectiveness of the rearrangement of array configurations in order to utilize the coalesced communication. In addition, we implement a tridiagonal solver that consists of the recurrence equation solvers for GPGPU systems in this paper.

5. Conclusion

GPGPU has been applied to a lot of applications because it produces the enormous computing power. In this paper, the non-linear recurrence equation, which is used to Thomas method for solving a tridiagonal matrix, is implemented on GPUs by using CUDA and the effectiveness of the rearrangement of array configurations is evaluated for the GPGPU implementation. Although the recurrence equations are hard to parallelize by nature, the P-scheme algorithm, which has been originally developed for distributed memory multicomputers, is applied to the non-linear recurrence equation and the tridiagonal matrix solver based on Thomas method. Moreover the rearrangement of array configurations is utilized as to use the coalesced communication for the data transfer between the global memory and the processing cores.

Our experimental results show that our method achieves a speedup of 16.84 in the case of the array size of $2^{25}$, whereas the conventional method (parallel cyclic reduction) achieves at most a speedup of 1.49.

In the near future, we will implement the solver by OpenCL on other GPU architectures than Nvidia and evaluate it. In addition, we will also construct an ADI solver based on our tridiagonal matrix solver on GPGPU systems.

Acknowledgments

This work was supported in part by MEXT, Japan.

References


