ViperVM: a Runtime System for Parallel Functional High-Performance Computing on Heterogeneous Architectures

Sylvain Henry

ViperVM: a Runtime System for Parallel Functional High-Performance Computing on Heterogeneous Architectures

Sylvain Henry

University of Bordeaux
351 cours de la Libération
33405 Talence, France
sylvain.henry@labri.fr

Abstract

The current trend in high-performance computing is to use heterogeneous architectures (i.e. multi-core with accelerators such as GPUs or Xeon Phi) because they offer very good performance over energy consumption ratios. Programming these architectures is notoriously hard, hence their use is still somewhat restricted to parallel programming experts. The situation is improving with frameworks using high-level programming models to generate efficient computation kernels for these new accelerator architectures. However, an orthogonal issue is to efficiently manage memory and kernel scheduling especially on architectures containing multiple accelerators. Task graph based runtime systems have been a first step toward efficiently automatizing these tasks. However they introduce new challenges of their own such as task granularity adaptation that cannot be easily automatized.

In this paper, we present a programming model and a preliminary implementation of a runtime system called ViperVM that takes advantage of parallel functional programming to extend task graph based runtime systems. The main idea is to substitute dynamically created task graphs with pure functional programs that are evaluated in parallel by the runtime system. Programmers can associate kernels (written in OpenCL, CUDA, Fortran...) to identifiers that can then be used as pure functions in programs. During parallel evaluation, the runtime system automatically schedules kernels on available accelerators when it has to reduce one of these identifiers. An extension of this mechanism consists in associating both a kernel and a functional expression to the same identifier and to let the runtime system decide either to execute the kernel or to evaluate the expression. We show that this mechanism can be used to perform dynamic granularity adaptation.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors – Run-time environments

General Terms Languages, Performance

Keywords Parallel Functional Programming, High-Performance Computing, Heterogeneous Architectures

1. Introduction

Starting with the Cell Broadband Engine, nearly a decade ago, the trend in high-performance computer design has shifted from high-frequency multi-core architectures to heterogeneous architectures composed of different kinds of accelerators (GPUs, Cell BE, MIC/Xeon Phi...) alongside an *host* multi-core. Programming high-performance applications for heterogeneous architectures is notoriously difficult: notwithstanding writing codes that are executed on accelerators (*kernels*), coordinating their executions and the appropriate data transfers in the code executed on the host is arduous and burdensome. On SMP architectures, operating systems manage process scheduling on the available cores and provide a virtual memory environment to applications – using swap on disk if necessary – so that applications do not have to micro-manage memory. On heterogeneous architectures, operating systems do not provide such capabilities, hence kernel scheduling on accelerators and distributed memory management fall to applications.

Different frameworks to write host programs provide different levels of abstraction. On the low-level end of the spectrum, frameworks such as OpenCL implementations let the host program manage everything: commands (data transfer, kernel execution, etc.) can be submitted to be asynchronously executed by the runtime system. It is the responsibility of the host program to schedule kernels appropriately on accelerators to keep their occupancy as high as possible in order to fully exploit the available computing power. In particular, it must ensure that data transfers are overlapped with computations to avoid stalling accelerators waiting for some data. Writing portable applications with these low-level approaches is difficult because there is a combinatorial explosion of the number of different architectures that can be encountered: each architecture can have a different number of accelerators with different attributes (memory amount, number of units of each kind, capabilities of the link between the accelerator and the host memory...). Hence, libraries and runtime systems providing higher levels of abstraction have been introduced to ease this burden.

Some frameworks restrict the scope of the supported architectures to make device management easier. For instance, OpenACC [23] only supports architectures containing a single accelerator. A less restrictive approach that has been successfully adopted is to rely on task graph based runtime systems such as StarSS [6] or StarPU [5]. Coupled with a shared-object memory model [38], they let applications allocate objects in a virtual memory and create a graph of tasks working on these objects. Kernels are associated to each task and runtime systems are responsible for task schedul-

ing and placement on accelerators, memory management and data transfers.

Applications using task graph based runtime systems rely on them to perform load-balancing. Hence, it is natural to expect them to perform granularity adaptation too. However, as task graphs are dynamically constructed by host codes in unpredictable ways, even if the runtime system is able to substitute a task with a task graph (whose tasks are of a smaller granularity) it is very hard to know when it is wise to do it. Additionally, performing granularity adaptation has an impact on memory management as task data can be partitioned in parts that can be scattered into the different memories. Hence recomposing a data from its parts is a costly operation that should be avoided as much as possible, which is again hard to predict without knowledge of future uses of the data.

In this paper, we propose an approach where task graphs are not dynamically constructed by an host code but described with pure functional programs that are reduced in parallel by a runtime system. We also present our runtime system called ViperVM that is a preliminary implementation of this approach. Functional programs being more declarative, they can be more easily transformed. In particular, an expression corresponding to a task execution can be substituted with an equivalent expression involving tasks with smaller granularities.

2. Background

This section briefly describes programming and execution models of some high-performance frameworks and runtime systems for heterogeneous architectures.

2.1 Low-Level Frameworks

CUDA and OpenCL are the two main low-level frameworks used to program heterogeneous architectures (especially GPUs). They both share a similar programming and execution model. They define a C-like language to write kernels using an Single-Instruction Multiple-Thread (SIMT) model. In the remainder of this section we use OpenCL terminology to describe the programming interface used by host programs. We do not explain how kernels are to be programmed are our focus in this paper is on the host code.

OpenCL provides an API to query available accelerators and their characteristics (type of device, memory sizes, capabilities, etc.). Buffers can be allocated by the host program in accelerator memories and asynchronous data transfers can be performed between host memory and device buffers. Kernel sources can be dynamically compiled for chosen devices. To execute a kernel, the host program has to select the device that will be handle the execution and to set kernel parameters (i.e. buffers in device memory that will be used by the kernel). An important thing to keep in mind is that the only side-effect that can be performed by a kernel is to write in a buffer that is configured to be accessed in write-mode. Kernels cannot use pointers and have no concept such as global variables: the buffers they have access to are explicitly set by the host code.

To ensure a correct scheduling of the asynchronous commands (data transfers, kernel executions...), OpenCL provides two mechanisms. First, several command queues have to be created and can each be associated to a single device. Hence there is at least one command queue per device that the host program wants to use. Enqueued commands are executed by the device associated to the command queue. Command queues can be configured to execute commands in the order they are submitted so that their executions do not overlap or to execute them in arbitrary order. The second mechanism uses `event` entities: each asynchronous command has a unique associated event and may depend on several events of other already submitted commands, even if they are submitted in different command queues. Events are used to dynamically create a command dependency graph.

2.2 Task Graph Based Runtime Systems

Low-level frameworks such as OpenCL are hard to use because applications have to handle accelerator memories and kernel scheduling. To make it easier, runtime systems such as GMAC [32] or SOCL [25] extend low-level frameworks to provide a distributed-shared memory (DSM) using a shared-object memory model. Buffers are not allocated on a specific device anymore but in the DSM. When an kernel is to be executed on a device, the runtime system performs required memory allocations and data transfers so that buffers set as parameters are available in device memory. If there is not enough space left in the device memory, it releases some buffers while ensuring that a copy of them is available in another memory, transferring them beforehand if necessary. The runtime system can also use prefetching strategies to anticipate data transfers once a kernel execution has been enqueued to be executed by a given device.

When accelerator memories are automatically managed, host programs basically only have to schedule kernels on devices. However scheduling an heterogeneous task graph on an heterogeneous architecture is a strong NP-hard problem [21], not to mention the fact that the graph is dynamically built. Several scheduling heuristics exist such as Heterogeneous Earliest Finish Time (HEFT) and its variants [9, 37, 40, 46] and are provided by runtime systems [5, 25, 41] that let host programs submit kernels without specifying devices that are to execute them. That is, applications create a kernel graph (or task graph) and the runtime system handles task scheduling in addition to accelerator memory management. For this to work, kernels have to be portable: either the kernel source can be compiled for several accelerators or several codes are provided for the different kinds of devices. Runtime systems based on OpenCL or that generates kernels (e.g. HMPP [17], Par4All [2]...) usually use the former approach while others such as StarSS [6] or StarPU [5] use the latter.

2.3 New Challenges

Host programs using task graph based runtime systems basically perform the following tasks: (1) they allocate and release data in the DSM; (2) they use IOs to initialize input data and to store output data; (3) they create the task graph (i.e. they submit kernels to be executed with their dependencies). Most of these actions are performed asynchronously with the execution of the kernels. As such, synchronization primitives are provided by the runtime system so that parts of the host code are executed appropriately after some tasks and before some others, for instance by using callback methods.

Challenge 1: how to do away with opaque host code? Host programs are hard to write because of the programming model involving asynchronous callbacks and mutable shared memory that is tricky and error-prone. In addition, we say that the host code is opaque to the runtime system because the latter cannot predict what will happen in host code sections: some data may be allocated or released and some new tasks may be submitted. Hence lookahead scheduling algorithms cannot be efficiently used, for instance those which execute in priority tasks that reduce the amount of required memory or tasks which have the highest number of dependencies. In addition, as control is performed in host code (loops, conditionals...), speculative task execution strategies are dismissed. Finally, a paradox is that the host code that creates the task graph is statically optimized by a compiler but the task graph itself is not. Although it is important to submit commands as fast as possible, it is even more important to create an optimal task graph.

Challenge 2: how to automatize data management? Currently data must be explicitly allocated by the host code in the DSM before tasks using them can be submitted. If we want to get rid of the opaque host code, we need to find a way to allocate them automatically when they are required. Data allocation may depend on previously computed results and as such we should find a way to integrate them into the task graph. Several frameworks of which OpenCL use explicit reference counting for buffers, thus have drawbacks of both explicit memory management and garbage collected approaches: explicit data release function calls in host code that decrement reference counters and no precise control on the time the memory release actually occurs. It gets even harder with frameworks that support sub-data (e.g. tiles of a matrix can be considered as independent data). For instance, StarPU requires that every task using a data have completed before tasks using its sub-data can be submitted (and vice-versa) implying additional arduous synchronizations in host codes.

Challenge 3: how to automatize task granularity adaptation? With automatic task scheduling, the host program cannot adapt task granularities to the targeted accelerators anymore. Nevertheless, on some architectures it would be better to have several "smaller" tasks than to have a single "big" task (for instance on the Cell BE where memory is a scarce resource on SPUs). Additionally, it may be desirable to be able to adapt task granularities depending on the runtime state (number of tasks being executed, number of idle devices, etc.). To support automatic scheduling of divisible load, runtime systems must provide a way for programmers to declare how each task can be replaced by an equivalent task graph. Scheduling strategies that benefit from these alternative task graphs (e.g. [7]) must be implemented.

3. ViperVM

To take up challenges stated in the previous section, we suggest to use pure functional programs to describe task graphs and to use a runtime system that performs parallel reduction of the programs. This proposition is based on the following observations: (1) kernels are almost side-effect free except for the buffers which are accessed in write mode; (2) a kernel graph is similar to a graph of super-combinators as kernels only depend on their parameters; (3) task dependencies are often used to set data-flow dependencies (read after write); (4) task graph based runtime systems often use a kind of garbage collecting strategy with explicit reference counting; (5) deciding which task to execute next is similar to choosing the next reducible expression in parallel functional rewriting systems.

By using this approach, we can conjecture solutions to the different challenges:

- **Challenge 1:** thanks to functional purity, programs can be quite easily transformed. Hence we can perform optimizations on the task graph (i.e. the pure functional program) both at compilation time and at execution time during the reduction of the program. For instance, optimizations such as inlining or common sub-expression elimination would become possible. In addition, it is easy to include control (function calls, conditionals...) into functional programs so that it is not hidden to the runtime system anymore.
- **Challenge 2:** we can use state-of-the-art garbage collection algorithms for pure functional programs. In addition, there is a clear separation between data that are manipulated by the kernel and which cannot contain references to other data, and data handles manipulated by pure functional programs.
- **Challenge 3:** by writing pure functions, programmers describe parametric task graphs that can be associated to kernels. During the reduction of the functional program, instead of executing a kernel (associated to a function call), the runtime system can substitute that function call with the alternative task graph parameterized with kernel parameters.

The foreseen issues of this approach are that kernel performing in-place data modifications do not fall easily into place in the functional model. A compiler or the runtime system would have to detect when it is safe to perform destructive updates otherwise it would have to duplicate data before kernel execution which can be very costly. This is a well studied problem [26, 39] that could be revisited in the context of heterogeneous architectures where some data are already duplicated in different memories and where destructive updates cannot add or remove references to other data (because there cannot be any in the first place), simplifying the garbage collection algorithm.

Another issue concerns non deterministic task graphs which cannot be expressed using pure functional programming without introducing special operators. For instance, we could have a set of tasks that perform write accesses into a given buffer without being dependent one upon the other. In this case the runtime system would sequentialize task executions in an unspecified order. We cannot express this kind of task graph because of the data immutability. However we could add a special unordered `fold` operator that would provide the same functionality if necessary.

Last but not least, performance of the whole runtime system could be an issue. Implicit parallel function languages often face too fine task granularities [24] that make thread management overhead too high compared to the benefits of the parallel execution. In our case, functional programs are used to coordinate the execution of computational kernels that are supposed to work on coarse-grained data so that the runtime overhead is compensated by a good exploitation of the heterogeneous architecture. This will have to be observed in practice though.

3.1 Implementation

ViperVM is our proof of concept implementation of a runtime system using parallel functional programming on heterogeneous architectures. It is implemented in Haskell and can be used like any other library. Listing 1 presents a simple Haskell program using ViperVM. An host code is still needed with our model to perform the following actions: (1) Configure the runtime system; (2) Register kernels that will be used; (3) Register input data; (4) Load the function program to evaluate; (5) Decide what to do with computed (output) data. In the remainder of this Subsection, we describe the different modules composing ViperVM.

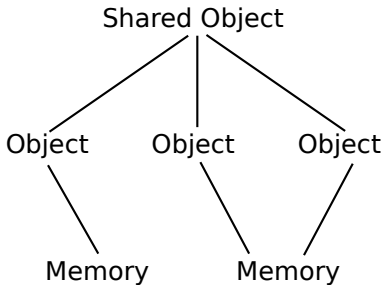
Generic Platform Internally, ViperVM uses the foreign function interface (FFI) to have access to low-level framework interfaces in order to control accelerators. A generic platform module is provided to the other modules: each enabled driver (Host, OpenCL, CUDA...) returns a set of *memories*, a set of *links* between memories and a set of *processors* attached to some memories. A generic API is provided by this module to allocate buffers in memories, to transfer data through links and to execute kernels on processors. The host driver is mandatory because it is used by other drivers as it returns a single memory entity corresponding to the host memory and a single loop-back link to perform data duplications in host memory.

Users have to configure the runtime system to choose device drivers to use – OpenCL, CUDA, NUMA... – and to set driver specific options. The current implementation only supports OpenCL and options are limited to the OpenCL library path as shown in Listing 1. Note that we decided to use dynamic linking for drivers because it is much harder to distribute software, especially in binary form (e.g. as a Linux distribution package) when it can be statically linked to several optional libraries chosen at compilation time and that may not be available in other environments.

Listing 1. Basic usage of the runtime system

```
import ViperVM.Platform.Platform
import ViperVM.Platform.Runtime
import ViperVM.Library.FloatMatrixAdd
import ViperVM.Library.FloatMatrixMul
import ViperVM.UserInterface
import ViperVM.Scheduling.Eager

let config = Configuration {
      libraryOpenCL = "libOpenCL.so"
    }

pf <- initPlatform config
rt <- initRuntime pf eagerScheduler

-- Initialize some data
a <- initFloatMatrix rt [[1.0, 2.0, 3.0],
                         [4.0, 5.0, 6.0],
                         [7.0, 8.0, 9.0]]

b <- initFloatMatrix rt [[1.0, 4.0, 7.0],
                         [2.0, 5.0, 8.0],
                         [3.0, 6.0, 9.0]]

-- Register kernels and input data
builtins <- loadBuiltins rt [
   ("+", floatMatrixAddBuiltin),
   ("*", floatMatrixMulBuiltin),
   ("a", dataBuiltin a),
   ("b", dataBuiltin b)]

-- Evaluate an expression
r <- evalLisp builtins "(+ (* a b) (* b a))"

-- Display the result
printFloatMatrix rt r
```

Figure 1. Shared-Object Memory Model



Data Management Our runtime system uses a shared-object memory model: data manipulated by functional programs are only handles associated to a set of effective objects that can be replicated in any memory. On top of the generic platform module, a `SharedObjectManager` module can be used to allocate a shared object, to attach an instance to a shared object or to detach an instance. Figure 1 represents a shared object that is associated to three objects. These objects contain the same data and are distributed in two memories, in particular the object is duplicated in the second memory. Duplicated objects can be stored differently (e.g. different count of padding bytes for matrices...).

An `ObjectManager` module provides primitives to allocate and release objects and to transfer the content of an object into another over a link between two memories (or using the loop-back link). Shared objects are considered *immutable* from a user perspective in the sense that they reference objects containing the same immutable content. To modify an object, it must first be detached from its associated shared object (if any). When it has been modified, it can be attached to another shared object (usually a freshly allocated one). A locking mechanism is used to ensure that objects are not modified concurrently. However, simultaneous read accesses are permitted.

In high-performance applications, it is common to work on sub-data. For instance, matrices may be partitioned in tiles that are processed in parallel by different processors. Our runtime system provides support for sub-data by allowing a shared object to indicate how it relates to another. When an operation uses a shared object specifying this information, it can use objects of the other shared object instead of its own objects. In fact, it may not have any object of its own. This mechanism lets applications work with a data and its sub-data at the same time.

Currently, the supported data types are vectors and matrices of single-precision floating-point values. Once the implementation is mature enough, we will add support for other data types. Regular matrix partitioning in tiles is supported. Each tile is then a sub-data with the same matrix type as its parent but with different dimensions.

Kernel Scheduling Schedulers are implemented as modules using both the generic platform and data manager modules. We provide two basic schedulers: round-robin and eager. The first one distributes kernels as they are submitted in a cyclic fashion to the available processors. The second one lets idle processors peek into a queue of kernels ready to be executed and that is shared by all of them. We have not yet implemented the whole machinery required for advanced scheduling heuristics such as HEFT. It would basically consist in selecting for each task the device that minimizes its predicted termination date (taking into account predicted data transfer durations, etc.), thus it would be necessary to store previous kernel execution times and to predict future kernel execution and data transfer durations.

In the current implementation, kernels are compiled lazily the first time they are scheduled on a processor. Other scheduling strategies could choose to compile kernels for all devices at once and/or to store binaries for future executions. In addition, some accelerators such as the Cell BE would require the kernel binary to be explicitly loaded on the accelerator before being executed. Several strategies to efficiently handle kernels that are already loaded on an accelerator could be envisioned.

Parallel Graph Reducer ViperVM provides a `Graph` module based on software transactional memory (STM) so that graph nodes can be modified concurrently. For this preliminary release we implemented a parallel reducer for graphs of supercombinators based on the template instantiation approach presented in Peyton-Jones and Lester book [34]. Thus we use a lazy evaluation order where the outermost reducible expression is reduced first. Currently, the main difference is that when we need to evaluate parameters of a built-in (e.g. kernels) operation, we evaluate them in parallel, spawning a light thread per parameter.

Once kernel parameters have been evaluated, the kernel and its parameters (data handles and immediate data) are submitted to the selected scheduler. Upon kernel completion, the function call associated to the kernel is substituted with the resulting expression, generally a data handle or an immediate data. It is up to the kernel scheduler to return a functional expression that is not in weak head normal form (WHNF) such as a function application. In this case the parallel graph reducer will reduce the returned expression. This

mechanism could be used by schedulers to perform task granularity adaptation.

Parser ViperVM is language neutral as long as the input language is purely functional and can be converted into a graph of supercombinators. Parsers transform functional expression that are given to them as strings into graphs that can be involved in the program execution. In Listing 1, we use a parser for a Lisp-like syntax to parse the expression `(+ (* a b) (* b a))`. We plan to write other parsers to support different syntaxes (e.g. Haskell) and different typing systems. Currently only the parser for a untyped Lisp-like language is implemented as it was the easiest to implement and because it sufficed to test and demonstrate the basic properties of our approach.

To make things clear at this point, there are three programming levels involved in our approach:

- Host code to configure the runtime system and to perform IOs: our runtime system is written in Haskell and presents itself as a library, hence host code examples are written using Haskell in this paper.
- Coordination of the computational kernels: programs written using pure functional languages are interpreted by our runtime system. In this paper, the pure functional language is always using a Lisp-like syntax, unless otherwise stated.
- Computational kernels executed by accelerators and CPUs: they can be written using low-level languages such as C, Fortran, OpenCL, CUDA... Additionally it should be possible to use kernels generated from high-level approaches such as Accelerate [13], Nikola [28] or Obsidian [42] in the future.

The following constructions are supported both by the parser and by the graph reducer. Top-level functions can be constructed with the `defun` operator. For instance, the following code define two functions `f` and `main`.

```
(defun f (x y)
   (+ (* x y) (* y x)))

(defun main ()
   (f a b))
```

If the source of this program is contained in the `src` variable, it can be evaluated with the following host code:

```
r <- evalLispModule builtins src
```

Anonymous functions are also supported. For instance, in the following code we define an anonymous function equivalent to the previous `f` that is applied without giving it a name:

```
(defun main ()
   ((lambda (x y) (+ (* x y) (* y x))) a b))
```

It is possible to introduce new variables in the functional expression by using `let` and `let*` operators. The difference between the two is that the second one permits mutually recursive definitions. The following code uses `let` to introduce two variables `c` and `d`.

```
(defun g (a b)
   (let ((c (* a b))
         (d (* b a)))
      (* (- c d) (+ c d))))
```

`let` and `let*` operators are important because they allow to define graphs instead of trees: variables introduced with these operators can be used more than once in an expression but the expressions they reference are only evaluated once.

Conditional execution is supported with the `if` operator. It takes three arguments: a condition expression and two expressions for the different branches. The condition is evaluated and depending on its result one of the two branches is evaluated. In the following code we use an hypothetical `isSymmetric` function to check if a matrix is symmetric in order to use a more efficient code to compute its square.

```
(defun square (m)
   (if (isSymmetric m)
      (ssyrk m)
      (sgemm m m)))
```

Kernel Library and Built-ins ViperVM provides a library of simple dense linear algebra kernels that can be easily used. Matrix multiplication and addition operations used in Listing 1 come from this library and only have to be associated to appropriate symbols to be used in a functional program.

A built-in list data type is supported alongside basic list operations (`head`, `tail`, `cons`...). This data type can be used to represent matrices as list of list of tiles and to write algorithm that use tiling. In particular, `split` built-in can be used to partition a matrix into tiles of the specified dimensions. The result is given as a list of list of matrices. Another important list operation is `reduce` that can be used to reduce a non-empty list in parallel by using a balanced binary tree pattern. It is not a built-in and its code is the following one:

```
(defun reduce (f xs)
   (if (null xs)
      error
      (head (reduce' f xs))))

(defun reduce' (f xs)
   (if (or (null xs)
           (null (tail xs)))
      xs
      (reduce' f (cons
         (f (head xs)
            (head (tail xs)))
         (reduce' f
            (tail (tail xs)))))))
```

Listing 2 shows how a tiled matrix multiplication can be written using list operations. `sgemm` takes two matrices (`x` and `y`) and three tile dimensions (`w`, `h` and `k`) as parameters. `x` is partitioned using `split` in blocks of dimensions $k \times h$ and `y` in blocks of dimensions $w \times k$. `sgemm'` is called with the partitioned matrices as parameters and the result is recomposed into a single dense matrix with `unsplit`. `outerWith f` performs an operation similar to an outer product of two vectors but where the multiplication is replaced with `f`, its first parameter. `sgemm'` uses it to combine every row of `xs` with every columns of `ys` using `dotProduct` function. The latter is a dot product on matrices using `add` and `mul` matrix addition and multiplication kernels, respectively. For completeness, note that `'()` is the Lisp syntax to create an empty list, `null` returns true if the given parameter is an empty list and `map` applies its first parameter to every element of the list given in second parameter.

As the parallel graph reducer performs lazy evaluation and stops on weak head normal forms, results may not be what users would expect. For instance, if we want to compute a list of values such as `'(1 (+ 1 1) 3)`, the parallel graph reducer will stop on the list constructor without evaluating anything. To force the evaluation, the built-in function `deepseq` can be used so that `(deepseq '(1 (+ 1 1) 3))` is correctly reduced to the normal form `'(1 2 3)`. Note that list elements are then evaluated in parallel.

3.2 Static Optimizations

In a preliminary work of ours, we used the rewrite rules of Glasgow Haskell Compiler (GHC) to statically increase parallelism in

Listing 2. Tiled Matrix Multiplication

```
(defun sgemm (x y w h k)
   (unsplit (sgemm' (split k h x)
                    (split w k y))))

(defun sgemm' (xs ys)
   (outerWith dotProduct xs (transpose' ys)))

(defun dotProduct (xs ys)
   (reduce add (zipWith mul xs ys)))

(defun outerWith (f xs ys)
   (map (lambda (y)
      (map (lambda (x) (f x y))
         xs))
         ys))

(defun transpose' (xs)
   (if (null (head xs))
      '()
      (cons (map head xs)
            (transpose' (map tail xs)))))

(defun zipWith (f xs ys)
   (if (or (null xs) (null ys))
      '()
      (cons
         (f (head xs) (head ys))
         (zipWith f (tail xs) (tail ys)))))
```

a functional program. This mechanism could be used in our runtime system to perform similar transformations. For instance, the following purely functional code written with Haskell syntax:

```
a + b + c + d + e
```

corresponds to the following code using Lisp syntax, where we clearly see that is will be reduced in sequence:

```
(+ (+ (+ (+ a b) c) d) e)
```

However with some rewrite rules, we can transform this expression into the following one which is more parallel:

```
(reduce (+) '(a b c d e))
```

In the previous subsection, we mentioned that during the reduction of the graph it is possible to substitute an expression with another equivalent one and that this mechanism can be used to perform granularity adaptation. For instance, when a matrix multiplication kernel is called with very large matrices as parameters, the function call could be replaced with a function call to `sgemm` as defined in Listing 2 with automatically chosen parameters. Now suppose that we want to compute `(* (* a b) c)` where `a`, `b` and `c` are very large matrices. The steps of the reduction are shown in Listing 3.

The issue with this derivation is that the result `ab` of the multiplication of `a` with `b` is recomposed (with `unsplit`) on line 5 and then partitioned again on line 8. Yet recomposition is a very costly operation that may involve many data transfers. We want to avoid these kinds of superfluous data transfers so the runtime system should do two things: (1) it should avoid doing do the recomposition if it is followed by a partition with the same parameters; (2) it should try to ensure that data partitions will be compatible with the following ones applied to the same data.

Listing 3. Substitutions with tiled matrix mutliplication

```
   (* (* a b) c)                                  1
=> (* (sgemm a b w h k) c)                        2
=> (* (sgemm' (split k h a) (split w k b)) c)     3
=> ...                                            4
=> (* (unsplit ab) c)                             5
=> (* ab' c)                                      6
=> (sgemm ab' c w' h' k')                         7
=> (sgemm' (split k' h' ab') (split w' k' c))     8
=> ...                                            9
```

This last issue is the most tricky to deal with. We plan to algebraically pre-compute some derivations at compilation time so that it would become possible to replace compositions of a recomposition and a partition with the identity function in some cases. For instance, consider a module that exports the following `mul3` function:

```
(defun mul3 (x y z)
   (* (* x y) z))
```

The derivation in Listing 3 could be performed algebraically at compilation time and it could be useful if the compiler could infer that in order to avoid a superfluous data recomposition, `k'` must be equal to `w` and `h'` must be equal to `h`. This way, we may be able to automatically derive an alternative task graph for `mul3` as well as constraints on the partitioning factors so that the search space domain is reduced.

Work on these transformations is still in progress. It shares many similarities with derivations in the Bird-Meertens Formalism [8] and several open questions remain such as: (1) how to decide to use a functional expression instead of another if several are provided? (2) how to select computing factors? We hope ViperVM could be used to develop and to experiment with heuristics for these issues.

4. Evaluation

In this section we present some code examples and some preliminary performance results. We use some basic linear algebra kernels that we have implemented but that have not been yet optimized for the targeted CPUs and GPUs. For these tests, we used an Intel Sandy Bridge E5-2650 dual-processor architecture with a total amount of 16 hyper-threaded cores and 64 gigabytes of memory. Three NVidia Tesla M2075 GPUs are used as accelerators. CPU and GPUs are used through the OpenCL implementation provided by each hardware vendors. In the case of the CPU, it means that commands such as `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` are used by ViperVM, hence superfluous data copies are performed in host memory, incurring additional overhead compared to a native C or Fortran implementation of the kernels.

In the first matrix addition example, we evaluate the overhead of our runtime system by comparing it with the same example implemented with StarPU. With the matrix multiplication example, we observe the scalability of the current implementation and we point out a drawback of the current eager scheduling strategy. In both cases, input matrices are filled with random single-precision floating-point values.

Note that absolute performance is under what we might have obtained if we had used highly tuned kernels (BLAS, etc.). We can observe from the comparison with StarPU, which can be very fast with this kind of kernels (see Magma project [1] for instance), that our prototype still has a some room for improvement before being on par with such mature runtime systems.

Table 1. Matrix Addition (tile dimensions: 8192x8192)

Input dimensions	ViperVM 3 GPUs + CPU	ViperVM 3 GPUs	StarPU 3 GPUs
16K x 16K	1.9s	2.1s	1.4s
24K x 24K	4.0s	4.4s	2.9s

4.1 Matrix Addition

Matrix addition is an operation that is performed element-wise. As such, it is very easy to write an equivalent algorithm that works on tiles of the input matrices. In the following code, input matrices `a` and `b` are partitioned in tiles of dimensions `(w,h)`. Then `add` is applied to every element-wise couple of tiles of `a` and `b` with the `zipWith2D` function. Finally, `unsplit` recomposes the resulting tiles to form a dense matrix.

```
(defun main ()
  (unsplit (zipWith2D add (split w h a)
                          (split w h b))))

(defun zipWith2D (f xs ys)
  (zipWith (zipWith f) xs ys))
```

For comparison, Listing 4 presents the host code using StarPU that creates the same task graph. Compared to the functional program, the code is much more verbose, in particular the data `c` used to store the result of the matrix addition has to be explicitly allocated and partitioned before kernels are submitted. Moreover, tasks working on matrix tiles have to be completed before `c` can be unpartitioned, hence the `task_wait_for_all` synchronization. A solution to avoid this barrier is to use callbacks associated to tasks but the code becomes even more cumbersome.

In order to test the substitution mechanism during the evaluation of the functional program, we have integrated a naive granularity adaptation strategy to the `add` operation: if one of the dimensions of the input matrices is greater than 8192, we substitute the `add` application with the previous equivalent code working on tiles where tile dimension is arbitrarily set to 8192. Hence the user code in the ViperVM case is just `(add a b)`.

Performance results are presented in Table 1 in comparison with performance obtained with StarPU. As StarPU does not support data transfers with a stride introduced in OpenCL 1.1, we have used an equivalent CUDA implementation of our OpenCL kernel, hence results with StarPU only use the GPUs (we checked that both kernel execution times are the same). These results show that our runtime system still has an additional overhead of 50% compared to StarPU on this example.

4.2 Matrix Multiplication

Matrix multiplication algorithm working on matrix tiles has already been presented in Listing 2 and the main part is reproduced here:

```
(defun main ()
  (unsplit (sgemm' (split w h a)
                   (split h w b))))

(defun sgemm' (xs ys)
  (outerWith dotProduct xs (transpose ys)))
```

With this second example, we show that ViperVM benefits from the simple eager scheduling strategy it uses when performing a tiled matrix multiplication. Performance results are reported in Table 2 for different values of `w` and `h` and we can observe that performance increases with the number of GPUs in almost all cases. The matrix multiplication kernel we implemented is not optimized and is particularly slow on the CPU so that using it actually decreases the overall performance. Hence adding the CPU shows a shortcoming of the eager scheduling strategy. We would need to use performance models and to implement a better scheduling strategy (such as HEFT) to avoid giving a task to the CPU if there are not enough tasks for the GPUs.

Table 2. Matrix Multiplication (4096x4096)

w x h	1024x1024	4096x1024	1024x4096
GPU (1x)	4.5s	4.4s	4.3s
GPU (2x)	3.6s	2.9s	3.2s
GPU (3x)	3.1s	2.5s	3.3s
CPU	31s	36s	35s
GPU (3x) + CPU	3.3s	3.7s	10s

5. Related Works

The current trend in the high-performance computing community is to rely on runtime systems to manage heterogeneous architectures. Indeed, actions usually performed by operating systems such as scheduling and low-level memory management have to be done explicitly in user code. Runtime systems stand in for operating systems on heterogeneous architectures.

Implementations of the OpenCL specification are not considered as runtime systems because they let users micro-manage everything explicitly. However, some OpenCL extensions such as the ones proposed by Boyer *et al.* [11], by Grewe *et al.* [22] or SOCL [25] add support for automatic management of several heterogeneous devices. Some other runtime systems provide their own interfaces, either using compiler annotations (pragmas) or an API. Despite its limited scope (support for architectures with a single accelerator only), several implementations of the OpenACC specification [23] are available. HMPP [17] is a more complete predecessor of OpenACC that is also based on compiler annotations and that generates kernels from C codes. Par4All [2] is another framework that generates kernels from sequential C codes. Finally, some runtime systems do not generate kernels for accelerators but rely on kernels written in low-level languages such as C, Fortran, OpenCL and CUDA that they schedule on available accelerators. Among them, StarSS [35] and XKaapi [19] provide compiler annotations while StarPU [4] mainly provides its own API (compiler annotations are also provided as a GCC plugin [16]). XKaapi is more targeted at using work-stealing scheduling policies while StarPU and StarSS provide several scheduling strategies, especially those based on performance models.

Automatic code generation for accelerators can also be achieved by compiling high-level functional data-parallel codes into low-level kernel codes. Accelerator [45] and Vertigo [18] were among the first frameworks to generate GPU pixel shaders from higher-order data-parallel operations. Nikola [28] and Obsidian [44] came later and generated CUDA codes. Similarly, Sh [30] used metaprogramming to generate shaders from C++ codes. It was later integrated into RapidMind development platform [14], that was itself later bought by Intel in 2009 and combined with Intel Ct [20] to give Intel Array Building Blocks (ArBB) [33], dropping GPU support during the process and discontinued by Intel in 2012. EmbArBB [43], the successor of Harbb [42], is an interface to ArBB for Haskell programs that avoided most C++ boilerplate code while still providing comparable performance. Accelerate [13] is another embedded Haskell language that generates CUDA kernels. It performs advanced optimizations such as sharing recovery and array fusion, in particular to ensure that kernel granularity is not too fine [31].

Other declarative approaches to represent task graphs and thus to avoid excessive unfolding have been used, for instance in para-

Listing 4. Tiled matrix addition example using StarPU

```
struct starpu_data_filter f = {
    .filter_func = starpu_matrix_filter_vertical_block,
    .nchildren = nw
};

struct starpu_data_filter f2 = {
    .filter_func = starpu_matrix_filter_block,
    .nchildren = nh
};

starpu_data_map_filters(a, 2, &f, &f2);
starpu_data_map_filters(b, 2, &f, &f2);
starpu_data_map_filters(c, 2, &f, &f2);

for (i=0; i<w; i++) {
    for (j=0; j<h; j++) {
        starpu_data_handle_t sa = starpu_data_get_sub_data(a, 2, i, j);
        starpu_data_handle_t sb = starpu_data_get_sub_data(b, 2, i, j);
        starpu_data_handle_t sc = starpu_data_get_sub_data(c, 2, i, j);

        starpu_insert_task(&add, STARPU_R, sa, STARPU_R, sb, STARPU_W, sc, 0);
    }
}

starpu_task_wait_for_all();
starpu_data_unpartition(c, 0);
```

metric task graphs (PTG) [15] and DAGuE [10]. Parallel reduction or rewriting of a functional program (i.e. a graph of supercombinators) is a relatively old idea: Hammond [24] mentions that one of the first suggestion of this technique appeared in 1975. However as it is hard to decide when to spawn a new thread because granularity can quickly become too fine and thread management overhead too high, explicit approaches are still predominant in functional languages such as Haskell [29] or Eden [27]. A good resource on parallel graph rewriting as found in Clean is Plasmeijer *et al.* book [36].

Some integrated approaches such as Delite [12] compile codes written using domain specific languages (DSLs) to target heterogeneous architectures. An execution graph is generated from the DSL in addition to kernels for each kind of device, then a runtime system schedules the generated kernels on the given architecture using the execution graph. Compared to our approach, Delite is more high-level and inclined toward automatic kernel generation, hence it does not separate coordination and computation codes as clearly as we do. Our approach is designed to be more conservative because even if we plan to integrate kernel generation in the future, a first milestone is to demonstrate that the switch from dynamic task graph creation to an approach involving parallel evaluation of pure functional programs is beneficial because it only focuses on improving the coordination code (regarding productivity, safety, performance with some transformations and automatic granularity adaptation...), leaving the computational code that is executed in fine on the accelerators as it currently is.

Transforming a functional expression to increase the performance of its execution, especially by increasing parallelism, is at the core of the Bird-Meertens formalism (BMF) [8]. Windows [47] proposes a set of transformation rules to optimize programs when a partition operator is inserted into them. PetaBricks [3] is a framework that takes several algorithms performing the same computation, of which at least one is recursive so that at each recursion level it selects the most appropriate algorithm by using auto-tuning. It is also able to select algorithms depending on their accuracy.

6. Conclusion

Heterogeneous architectures have become ubiquitous, especially in high-performance supercomputers. Task graph based runtime systems have been used successfully to provide automatic memory management and task scheduling to applications. However, they introduce new issues that need to be dealt with. We identify three challenges for the upcoming frameworks: (1) reducing the amount of host code that is opaque to the runtime system in order to enhance its ability to make intelligent decisions; (2) improving data management so that users do not have to explicitly allocate and release intermediate data (produced by a task and consumed by another); (3) performing task granularity adaptation automatically.

In this paper, we presented ViperVM, a runtime system prototype that uses an approach combining parallel functional programming and task graph based runtime systems for heterogeneous architectures. It leverages pure functional programming to deal with issues such as automatic granularity adaptation and more generally to provide a new framework allowing a new class of optimizations (at compilation time and at execution time) and of scheduling strategies to be used.

7. Future Work

The current implementation of ViperVM is closer to a proof of concept than to a mature runtime system as evaluations show. Some engineering time will be devoted to implement missing functionalities (garbage collection, other drivers, etc.). In addition, there are several further directions we would like to explore.

Granularity Adaptation We would like to develop heuristics and static optimizations so that the runtime system could automatically perform granularity adaptation. Ideally, we would like the runtime system to automatically choose partitioning factors. Thus,

we need to work on algebraic transformations of the functional programs so that constraints on factors can be inferred to reduce the search space domain.

Scheduling Policies We would like to experiment different scheduling policies. In particular, we could use lookahead now offered by the more declarative model to execute in priority tasks that reduce the memory pressure or whose results are required by many other tasks. Speculative execution is another mechanism that we would like to evaluate in this context.

Destructive Updates Kernels performing destructive updates are commonly found especially in codes using dense linear algebra. We want to allow their use while minimizing the number of superfluous data duplications. The memory model we use may offer new optimization opportunities as data may already be duplicated in different memories. Interaction with the garbage collection algorithm could be interesting too: if there is only one remaining task using a data then it can perform a destructive update.

Automatic Benchmarking QuickCheck is an Haskell library used for testing purpose. Given a function prototype, it automatically generates test cases (i.e. sets of input data) and function results are checked against constraints. We would like to use a similar mechanism to automatically generate micro-benchmarks for the computational kernels. Additionally, we could automatically check that different kernels performing the same operation on different architectures return correct results.

Check-Pointing With our model, the runtime system controls the parallel reduction of the program. We would like to provide a way to stop the reduction and to store the current state of the program on disk alongside the data required to continue the execution. Thanks to functional purity, it may be possible to automatically perform check-pointing without interrupting the reduction.

Acknowledgments

We are grateful to the anonymous reviewers for their useful comments and suggestions.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. *Faster, Cheaper, Better a Hybridization Methodology to Develop Linear Algebra Software for*. 2010.

[2] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. McMahon, F. Pasquier, G. Péan, and P. Villalon. Par4all: From convex array regions to heterogeneous computing. In *2nd International Workshop on Polyhedral Compilation Techniques, Impact (Jan 2012)*, 2012.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, Aug. 2009. Springer.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 2010.

[6] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. *Euro-Par 2009 Parallel Processing*, pages 851–862, 2009.

[7] O. Beaumont, L. Marchal, Y. Robert, and L. de l'informatique du parallisme. Scheduling divisible loads with return messages on heterogeneous master-worker platforms. 2005.

[8] R. Bird et al. *Lectures on constructive functional programming*. Oxford University Computing Laboratory, Programming Research Group, 1988.

[9] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 27–34. IEEE, 2010.

[10] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2011.

[11] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load balancing in a changing world: dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 21. ACM, 2013.

[12] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100. IEEE, 2011.

[13] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[14] I. Christadler and V. Weinberg. Rapidmind: Portability across architectures and its limitations. In R. Keller, D. Kramer, and J.-P. Weiss, editors, *Facing the Multicore-Challenge*, volume 6310 of *Lecture Notes in Computer Science*, pages 4–15. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-16232-9.

[15] M. Cosnard and M. Loi. Automatic task graph generation techniques. In *System Sciences, 1995. Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122. IEEE, 1995.

[16] L. Courtès. C language extensions for hybrid cpu/gpu programming with starpu. *arXiv preprint arXiv:1304.0878*, 2013. URL `http://arxiv.org/abs/1304.0878`.

[17] R. Dolbeau, S. Bihan, and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. 2007.

[18] C. Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 45–56. ACM, 2004.

[19] T. Gautier, J. V. F. Lima, N. Maillard, B. Raffin, et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013. URL `http://hal.inria.fr/hal-00799904/`.

[20] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A flexible parallel programming model for tera-scale architectures. Intel, 2007.

[21] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics. v5*, pages 287–326, 1977.

[22] D. Grewe, Z. Wand, and M. F. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *ACM/IEEE International Symposium on Code Generation and Optimization*, Shenzen, China, Feb. 2013.

[23] O. Group. The openacc application programming interface, 2011.

[24] K. Hammond. Parallel functional programming: An introduction. In *International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, 09 1994. World Scientific.

[25] S. Henry, A. Denis, and D. Barthou. Programmation unifie multi-acclrateur opencl. *Techniques et Sciences Informatiques*, 31(8-9-10): 1233–1249, 2012. . URL `http://hal.inria.fr/hal-00772742`.

[26] J. Launchbury and S. L. P. Jones. State in haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995. URL `http://link.springer.com/article/10.1007/BF01018827`.

[27] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in eden. *Journal of Functional Programming*, 15(3): 431–476, 2005.

[28] G. Mainland and G. Morrisett. Nikola: embedding compiled gpu functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. .

[29] S. Marlow. Parallel and concurrent programming in Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *CEFP 2011*, volume 7241 of *LNCS*, pages 339–401. 2012.

[30] M. D. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Wellesley, 2004.

[31] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional gpu programs. In *ICFP*, 2013.

[32] I. Multicoreware. Gmac: Global memory for accelerator, tm: Task manager, 2011. http://www.multicorewareinc.com.

[33] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 224–235, 04 2011. .

[34] S. L. Peyton Jones and D. R. Lester. *Implementing functional languages: a tutorial*. Prentice-Hall, Inc., 1992.

[35] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[36] R. Plasmeijer, M. Van Eekelen, and M. Plasmeijer. *Functional programming and parallel graph rewriting*, volume 857. Addison-wesley, 1993.

[37] S. Ranaweera and D. P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 445–450. IEEE, 2000.

[38] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, May 1998. ISSN 0164-0925. .

[39] P. Roe and A. Wendelborn. Implicit array copying: Prevention is better than cure, 1992.

[40] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111. IEEE, 2004.

[41] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 84–93, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1233-2. .

[42] B. J. Svensson and R. Newton. Programming future parallel architectures with haskell and intel arbb. 2011.

[43] B. J. Svensson and M. Sheeran. Parallel programming in haskell almost for free: an embedding of intel's array building blocks. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1577-7. .

[44] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. *Implementation and Application of Functional Languages*, pages 156–173, 2011.

[45] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. .

[46] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260 –274, mar 2002. ISSN 1045-9219. .

[47] J. Windows. Automated parallelisation of code written in the bird-meertens formalism, 2003.