

Kite: Braided Parallelism for Heterogeneous Systems

J. Garrett Morris

Department of Computer Science,
Portland State University, Portland, OR,
USA
jgmorris@cs.pdx.edu

Benedict R. Gaster

Advanced Micro Devices, 1 AMD,
Sunnyvale, CA, USA
benedict.gaster@amd.com

Lee Howes

Advanced Micro Devices, 1 AMD,
Sunnyvale, CA, USA
lee.howes@amd.com

Abstract

Modern processors are evolving into hybrid, heterogeneous processors with both CPU and GPU cores used for general purpose computation. Several languages, such as BrookGPU, CUDA, and more recently OpenCL, have been developed to harness the potential of these processors. These languages typically involve control code running on a host CPU, while performance-critical, massively data-parallel kernel code runs on the GPUs.

In this paper we present Kite, a rethinking of the GPGPU programming model for heterogeneous braided parallelism: a mix of task and data-parallelism that executes code from a single source efficiently on CPUs and/or GPUs.

The Kite research programming language demonstrates that despite the limitations of today's GPGPU architectures, it is still possible to move beyond the currently pervasive data-parallel models. We qualitatively demonstrate that opening the GPGPU programming model to braided-parallelism allows the expression of yet-unported algorithms, while simultaneously improving programmer productivity by raising the level of abstraction. We further demonstrate Kite's usefulness as a theoretical foundation for exploring alternative models for GPGPU by deriving task extensions for the C-based data-parallel programming language OpenCL.

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

General Terms Languages, tasking models, experimentation

Keywords GPGPU, compiler, work-stealing, concurrency abstractions

1. Introduction

The future is already here—it's just not very evenly distributed.

William Gibson

Following the single-core and multi-core revolutions there is now a new emerging era of computing: heterogeneous systems. Heterogeneous devices are no longer simply the realm of computer architecture classes or embedded developers, but are accessible to mainstream programmers. Hardware limitations now constrain both single-core and multi-core systems (see Figure 1) and

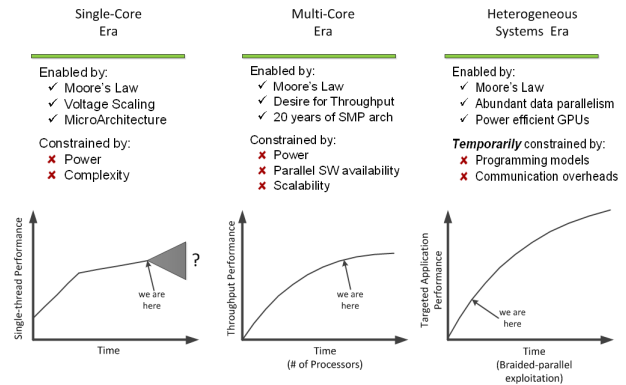


Figure 1. Three Eras of Processor Performance

it is unclear if these can be overcome. Heterogeneous systems are not without their own problems. Today, they suffer from high communication overheads (such as that imposed by the PCIe interface) and the lack of any effective established performance-portable programming model.

Many forms of parallelism are present within individual applications. For example, Figure 2 shows a single job frame from DICE's game Battlefield: Bad Company 2. This frame shows a mixture of both task- and data-parallelism, a form of parallelism Lefohn [12] calls *braided parallelism*. This is only one frame; a game like Bad Company 2 includes numerous types of parallelism, including:

- Concurrent threads for user interfaces and IO;
- Task parallelism for frame decomposition and AI;
- Different granularities of data-parallelism for the physics and particles systems; and,
- Parallel rendering of graphics.

While there is clearly a large amount of parallelism in such an application, it is by no means embarrassingly parallel. While there are embarrassingly parallel components, such as particle systems and graphics rendering, in general, work will be dynamically generated and is unlikely to be regular in nature.

There are numerous programming languages and libraries for building applications with braided parallelism in a multi-core environment, including Intel's Thread Building Blocks (TBB) [22] and Microsoft's Task Parallel Library (TPL) [13] for .NET4. However, there is a stark lack of support for braided parallelism in today's popular programming models for GPGPU computing, promoted by NVIDIA's CUDA [17] and Khronos' Open Compute Language (OpenCL) [19]. CUDA and OpenCL are designed around execution

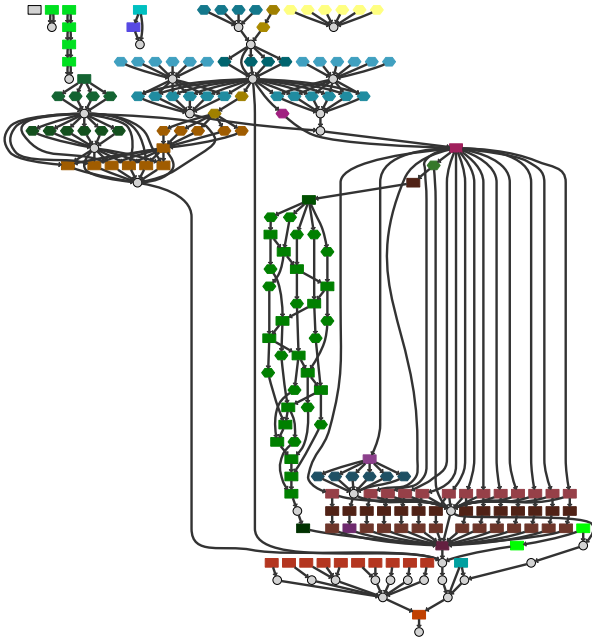


Figure 2. Job frame from DICE's Battlefield Bad Company 2 (PS3)

of multi-dimensional data-parallel grids and while each enqueued kernel instance may be viewed as a data-parallel task, the requirement to manage queued entities on the host acts as a limiting factor in scaling performance on a range of devices.

We demonstrate a recursive, task-parallel programming model for GPGPU environments, implemented in a new, higher-order programming language called Kite. We then show its practicality by describing a compilation strategy from Kite to OpenCL. Kite improves on existing approaches for GPGPU programming both by addressing issues of load imbalance and by supporting nested parallelism through recursive task creation. In designing Kite, we drew from a number of existing languages, such as Cilk [5], ML [16] and NESL [4], and libraries, including Intel's TBB and Microsoft's TPL. However, unlike most previous approaches, parallelism is the default in Kite. Rather than relying on explicit introduction and elimination of parallelism, we treat all task invocations as parallel and automatically introduce synchronization when required. For example, consider the following simple example of Kite code:

```
task fib(n: int): int {
  if (n < 2) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

This example should seem unsurprising, if inefficient. However, because of Kite's parallel-by-default semantics, the recursive calls to `fib` within the body of the `else` block are parallel invocations; intuitively, `fib` "waits" for its subtasks to return before computing the sum and returning itself. While this model is unusual, there is no particular reason that it could not be adopted by other languages. However, targeting OpenCL imposes additional limitations on our implementation, as OpenCL does not support many of the features traditionally used in compiling similar languages, including:

- Function pointers;

- Dynamic allocation; and,
- Call stacks.

To account for these missing features, Kite's compilation pipeline differs from more traditional compilers. A Kite program, presumably including a number of (possibly nested) tasks and multiple recursive invocations, is compiled to a single function (the uberkernel) which includes all the code paths in the original program. We then use a memory-aware extension to a traditional work-stealing task queue to support (recursive) task invocation: invocation translates to enqueueing a new task for execution, and the program itself is executed by repeatedly dequeuing and executing tasks.

As a further simple motivating example for Kite, consider a tree-based algorithm such as that used for Barnes-Hut simulation [2]. It's easy to see that in this sort of algorithm the amount of parallelism in the system increases as we descend through the tree. On a highly parallel device operating over such a structure involves tradeoffs. One option is that we create the full amount of parallelism at the top of the tree (assuming we know how many leaves we have) and run multiple threads on the same nodes at the top, branching differently as we go down. On the way up we require synchronization primitives to recombine return values. The lack of a large, efficient stack on such highly parallel systems means that any efficient implementation requires additional information in the tree to provide links to support moving back up the tree or to show which node to move to next in a depth-first scan. In addition, where we have unnecessary replication of work we are wasting compute resources.

Kite improves this model by representing computation as a series of enqueued tasks, performing synchronization implicitly via continuations; this fills the system in a dynamic fashion that will naturally load balance. By not filling the device with replicated work at all stages we also allow other tasks to use idle resources in parallel with the narrower parts of the tree. The textual representation of the program is improved by looking almost identical to a recursion-based functional example: parallelism is naturally extracted during compilation rather than having to be carefully and confusingly designed in.

Many of the ideas that underlie the implementation of Kite are equally applicable in more traditional GPGPU programming models. To demonstrate this, we also developed an extension to OpenCL itself, called *Tasklets*. While not including Kite's parallel-by-default model or automatic synchronization, Tasklets introduce notions of task invocation and synchronization while only requiring a small number of changes to the OpenCL API and OpenCL C device language.

In summary this paper makes the following contributions:

- A general purpose model for braided parallelism, supporting light-weight task and data parallelism, implemented in the programming language Kite. Kite addresses not only existing GPGPU environments, but also emerging heterogeneous platforms that closely integrate CPU and GPU cores on the same die.
- A translation from Kite into OpenCL, demonstrating that it can be implemented on existing hardware and that a single source program can effectively target both CPUs and GPUs.
- An extension of OpenCL, called Tasklets, to support explicit task parallelism.
- An implementation of task parallelism that uses an extension of Chase-Lev's work-stealing algorithm [7] to account for reduced memory traffic and better utilization of memory locality on GPU hardware.

To our knowledge this is the first time a fine-grained general purpose tasking model has been described for the GPGPU environment.

The paper continues as follows. First (§ 2) we describe the background of parallel computing, and GPGPU programming in particular, and describe models and languages related to our work. We then (§ 3) describe Kite, our high-level programming language, and its model of parallel computation. We describe its implementation (§ 4), including both the transformations necessary to translate a Kite program into OpenCL (§ 4.1) and our runtime support for tasking (§ 4.2). Finally we describe OpenCL tasklets (§ 5) and conclude (§ 6).

2. Related Work

Numerous high-level tasking languages and libraries have been proposed [5, 13, 22]; however, for the most part these have targeted multi-core CPU environments, often making design choices that do not adapt well to the throughput style architectures of modern GPUs. While there has been some work on evolving the GPGPU programming model beyond massively data-parallel computations, it has been with a domain-specific focus (most commonly in the area of graphics) and these systems have focused on optimization and performance for the specific application domain instead of providing general purpose computing facility.

Probably the closest work we are aware of is NVIDIA's general purpose ray tracing engine, OptiX [20]. OptiX implements a programmable ray-tracer, providing a single-ray programming model supporting recursion and dynamic dispatch, and thus allowing its user to build a variety of ray tracing-based applications. OptiX and Kite share a number of implementation techniques, particularly their implementation of light-weight tasks via persistent threads and support for recursion via continuations. However, unlike Kite, OptiX targets a specific application domain on a specific platform. While this means that the OptiX compiler has access to domain-specific information to drive optimization, it also limits its applicability. Similarly, while OptiX represents a significant step in GPU-based ray tracing, it does not necessarily compare well against CPU-based ray tracers [15]. In contrast, Kite has been designed to support general-purpose computation and to provide performance portability across a variety of platforms, from CPUs and GPUs to the emerging heterogeneous platforms. While this generality may limit the number of optimizations available to the Kite compiler, we believe that it more than compensates by increasing the number of possible applications.

To date general purpose programming language design for GPUs has focused on data-parallel decomposition, with early work relying on rendering full screen quads to execute pixel shaders (traditionally intended to color pixels) written in GLSL [21]. Later, streaming and vector algebra languages were proposed [6, 14] that avoided graphics terminology but were otherwise primarily sugared pixel shaders, with applications of these languages continuing to have a graphics focus. CUDA [17] and OpenCL [19] recast this model within the C/C++ language framework, providing programmers with a more familiar environment, and added support for scattered memory writes.

There are a number of languages and libraries that provide task-parallel programming models targeting multi-core CPU environments, including Cilk [5], Thread Building Blocks (TBB) [22], and the .NET4 Task Parallel Library (TPL) [13]. Like Kite, these languages approach parallel programming from a high-level language perspective, either building parallel constructs into the language or expressing them using tools such as object-oriented classes and lambda abstractions. Cilk and TBB expose an underlying model of threaded parallelism, with explicit operations to spawn new parallel tasks and synchronize with spawned tasks; in contrast, Kite makes

no assumptions about the underlying model, and both spawning new tasks and synchronization are implicit. TPL exposes a similar model to Kite, but is built as a library rather than using features of the underlying language; while this fits their target, it limits the possibilities of their translation; in contrast, the translation from Kite to OpenCL relies on whole-program transformations that can only be performed by a compiler.

Kite models braided parallelism within a shared memory model, using conventional synchronization primitives, such as locks and critical sections, when additional communication is necessary. Kite's use of shared memory primitives reflects their ease of expression in our target language, OpenCL, rather than any particular belief that they ought to form part of a concurrent language design. We acknowledge the claim of Reppey [23], among others, that shared memory models do not ease the burden of developing modular concurrent software, but suggest that the design of more expressive concurrency operations (such as those of Concurrent ML) is orthogonal to our goals in this paper.

Hou et al. describe a container-based approach to performance portability across heterogeneous systems [10]. Their approach relies on source code containing explicit data-parallel constructions and using a limited number of provided container structures. Their run-time system selects different versions of code to implement these constructs in different environments. While their system provides performance portability, it does so at the cost of limited expressiveness: their approach is unable to express nested parallelism. However, we believe the notion of using containers to express semantic guarantees is valuable, and hope to explore its application in Kite in future work.

NESL [4] is a data-parallel language loosely based on the functional language ML. Like Kite, NESL supports nested parallelism and encourages parallel expression of all computations. However, NESL expresses parallelism primarily at the sequence level, rather than at each function invocation. As a result, tree-driven algorithms (and some other recursive algorithms) may be easier to express in Kite than in NESL. Furthermore, NESL is targeted solely at multi-core CPU environments with high communication bandwidth between processors; in contrast, Kite can target the GPGPU environment, where communication between processors can introduce expensive synchronization.

Finally, while translating Kite to OpenCL, we use an intermediate representation that strongly resembles the joins of the join calculus [9]. While it would be possible to expose the join structure in the surface Kite language, and indeed our prototype implementation does, it is never necessary to express joins directly in Kite; rather, necessary synchronization points can be automatically inserted by a primarily type-driven compilation process. Existing implementations of the join calculus [8, 11, 18] are targeted at multi-core CPU, rather than GPGPU, environments.

3. The Kite Programming Model

Kite is a recursive, higher-order, statically-typed programming language developed to explore how ideas of task and braided parallelism can be implemented and used in a GPGPU or heterogeneous environment. This section describes Kite and its programming model. We begin by briefly discussing Kite's syntactic (§ 3.1) and typing (§ 3.2) rules; we expect that practitioners familiar with conventional procedural and functional languages will find little of surprise in these sections. Then (§ 3.3), we discuss task parallelism in Kite: how it is introduced, how it is eliminated, and the (relatively few) guarantees of sequential execution in Kite. Finally (§ 3.4), we discuss expressing data parallelism in Kite, both using the normal language features and using a special data-parallel map statement.

$t ::= x$	base types
$ x(\vec{t})$	generic type applications
$p ::= x: t$	parameter/field specifications
$f ::= x = e$	labeled field values
$e ::= x \mid c \mid e(\vec{e})$	variables, constants, applications
$ e.x \mid e[e]$	selection
$ (\vec{e})$	tuple construction
$ t\{\vec{f}\}$	struct and union constructors
$ \text{new}(t)(\vec{e})$	array constructors
$ [e, e..e]$	range constructors
$ \text{task}(\vec{p}): t \{ \vec{s} \}$	anonymous tasks
$ \text{task } e$	futures
$d ::= p = e$	value declarations
$ \text{task } x(\vec{p}): t \{ \vec{s} \}$	task declarations
$ \text{struct} \{ \vec{p} \}$	struct declarations
$ \text{union} \{ \vec{p} \}$	union declarations
$b ::= x x: \vec{s}$	case statement branches
$s ::= e; \mid d;$	expressions, declarations
$ \text{return } e;$	return from task
$ x \leftarrow e;$	assignments
$ s \text{ then } s$	sequencing
$ \text{if } (e) \{ \vec{s} \} \text{ else } \{ \vec{s} \}$	conditionals
$ \text{case } (e) \{ \vec{b} \}$	union deconstruction
$ \text{for } (x = e[x]) \{ \vec{s} \}$	sequential for-each loop
$ \text{map } (x = e[x]) \{ \vec{s} \}$	parallel for-each loop

Figure 3. The Kite concrete syntax. We assume non-terminals x for identifiers and c for constants. Note that our actual parser allows some obvious variations on the syntax presented here, such as omitting semi-colons immediately preceding or following right braces, omitting else blocks from if statements, or omitting types in value declarations.

3.1 Kite Syntax

We present an (abbreviated) syntax for Kite in Figure 3. Kite’s syntax is intended to be broadly familiar to C/C++ programmers, while adding some notation to support Kite-specific parallelism notions or to discourage common C programming mistakes. We summarize the changes next.

Higher-order features. Kite supports some higher-order programming techniques. We support creating anonymous tasks (for instance, `task (x:int):int { return x + 1; }` is a task that returns the increment of its argument), including tasks in the parameter lists of other tasks (using the syntax `task<t,u,...,v>` for the type of a task that takes parameters of types t, u, \dots , and returning a value of type v), and passing tasks as arguments to other tasks (which requires no special syntax).

Extended support for structures and unions. Kite extends the C/C++ notion of structures and unions to more closely match the algebraic data types of languages like ML, while still retaining C-like guarantees of the structure of the underlying representation. Union values in Kite are automatically tagged with the branch of the union used to construct them; this makes misinterpreting union values impossible. We have added a case statement to Kite to support deconstructing unions. Kite supports construction of new structure and union values in any expression context, whereas C only permits their construction in declaration statements.

Extended support for collection types. Kite provides extended support for iterating over and operating with collection types. The looping constructs in Kite (`for` and `map`) iterate over collections, computing bounds and index values automatically (while still allowing the user access to element indices if needed). For instance,

if v is a collection of `ints`, we could initialize each element to its position in the collection as follows:

```
for (x = v[i]) { x <- i; }
```

Note that the element (but not the index) bound by the `for` loop is an alias: assignments to it are equivalent assignments to the corresponding element of the collection.

Parallelism. Finally, some syntactic features of Kite expressly support parallel programming. The `then` keyword introduces ordering dependencies between statements that might not otherwise have them, while the `map` keyword describes loops that are semantically identical to `for` loops but introduce additional parallelism. We will return to these constructs in sections 3.3 and 3.4, respectively.

3.2 Kite Typing

Kite’s typing rules are as unsurprising as its syntax; in the interest of space, we omit them here. However, we do describe one aspect of the type system as it relates to Kite’s model of parallelism.

As first described in the introduction, Kite uses a parallel-by-default model of execution. We reflect this by the introduction and elimination of `future` types in the type system. One example of the introduction of `future` types is in the typing rule for applications:

$$\frac{\vdash e : \text{task}(\vec{t}, r) \quad \vdash e_i : t_i}{\vdash e(e_i) : \text{future}(r)} [\text{APP}]$$

(Our full typing rules include environments mapping variables and structure fields to types; we have elided those environments as they do not contribute to the rules we are presenting.) In turn, `future` types can be eliminated by applications of the non-syntax-directed `FORCE` rule:

$$\frac{\vdash e : \text{future}(t)}{\vdash e : t} [\text{FORCE}]$$

The `FORCE` rule is intended to make the introduction of `future` types ultimately transparent to the user, and, in fact, we have needed no explicit futures in our work with Kite so far. However, while transparent, it also serves to indicate both the locations where parallelism could be introduced (which correspond to the introduction of `future` types) and the locations where parallelism must be eliminated (at the uses of the `FORCE` rule). It would be possible to implement Kite directly in this fashion; however, waiting for other computations to finish is expensive on the GPU, both because it requires access to shared memory, itself an expensive operation, and because it could potentially idle entire vector cores when only some of the channels need to wait. As a result our implementation of Kite takes a different approach, albeit still guided by the meanings of `future` and `FORCE` we have just described.

3.3 Task Parallelism in Kite

This section builds on the previous description of potential parallelism to describe the parallel interpretation of Kite programs. In particular, we describe the computation of a task graph from a Kite program. The task parallelism in any piece of Kite code arises from disconnected nodes in the task graph being run in parallel.

We begin with an example. In the introduction, we presented the following implementation of the Fibonacci function:

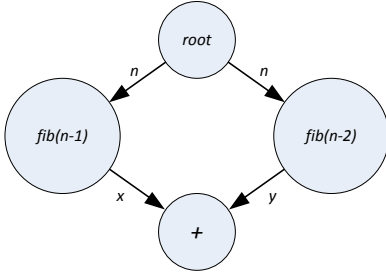
```
task fib(n:int):int {
  if (n < 2) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

$g ::= (\overrightarrow{(id, n)}, \vec{e})$	graphs
$e ::= (id, id)$	edges
$n ::= \text{stmt } s$	non-compound statements
$\text{if } (e) \ g \ g$	conditionals
$\text{case } (e) \ (\overrightarrow{id, id, g})$	
$\text{for } (x = e[x]) \ g$	iteration
$\text{map } (e = x[e]) \ g$	
$\text{task } (\vec{p}) \ t \ g$	task declarations

Figure 4. The syntax for our intermediate language of nested task graphs. We assume a syntactic class of unique identifiers, indicated by *id*. Non-terminals *e*, *s*, *x* are used as in Figure 3. We will maintain the invariant that nodes in a graph are listed in topological sorted order.

```
}
}
```

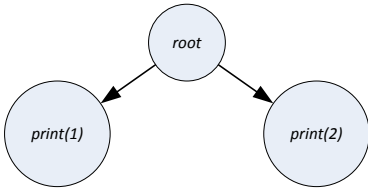
The opportunity for parallelism arises from the two recursive calls to *fib* in the *else* block. We could represent this with the following task graph:



Note that, in general, we do not assume any sequencing between expressions without data dependencies, regardless of their relative placement in the source code. For example, the following code:

```
task example() {
  print(1); print(2);
}
```

would give rise to the task graph



As suggested by the task graph, the program could equally validly generate the output 12 or 21.

In the remainder of this section, we describe the current algorithm we use to compute task graphs from Kite programs. We do not consider this algorithm a specification of the maximum parallelism possible from a given Kite program; however, we do not believe any present shortcomings are fundamental to the approach.

Our goal is to describe a transformation from Kite statements, as described in Figure 3 to a language of *nested task graphs* (NTGs), described in Figure 4. Intuitively, each node in a task graph corresponds to a single statement in the Kite source; we nest complete task graphs inside some nodes corresponding to compound Kite statements.

We separate the construction of these graphs into three stages, described in the next sections. First (§ 3.3.1), we rewrite Kite source

code to separate potentially parallel expressions into separate statements; second (§ 3.3.2), we construct task-graph nodes for each statement, while simultaneously computing dependencies for each node; finally (§ 3.3.3), we use the dependency information to connect the nodes into a graph. The latter two steps are necessarily mutually recursive—for example, to compute the dependency for an *if* statement, we have to have finished the graph construction for its alternatives. We also (§ 3.3.4) discuss a further graph transformation that would expose additional parallelism, but potentially at the cost of additional complexity in the graph structure.

3.3.1 Rewriting

We begin by introducing separate statements for each potentially parallel expression; this amounts to rewriting to a type of A-normal form that allows function arguments to be any non-parallel expression. For example, we would rewrite the Fibonacci function as:

```
task fib(n:int):int {
  if (n < 2) {
    return n;
  } else {
    x, y: int;
    x <- fib(n - 1);
    y <- fib(n - 2);
    return x + y;
  }
}
```

where *x* and *y* are fresh variables. We perform several additional transformations at the same time, such as renaming variables to avoid shadowing, moving type declarations to the top level, and lifting anonymous tasks to named task declarations in the same block. These translations are fairly mechanical, and we will not describe them further.

3.3.2 Dependency and Node Construction

After rewriting, we construct one NTG node per statement. Additionally, we compute data dependencies for each node. We define data dependencies using the following grammar, where *X* and *Y* are sets of variables:

$$\Delta ::= X \rightarrow Y \quad \text{non-parallel computations} \\ \quad \quad \quad | X \rightsquigarrow Y \quad \quad \text{parallel computations}$$

Let α, β range over $\{\rightarrow, \rightsquigarrow\}$, and let \rightarrow be ordered before \rightsquigarrow . We can now define the sum and composition of dependencies as follows:

$$(X\alpha Y) + (X'\beta Y') = (X \cup X') (\max\{\alpha, \beta\}) (Y \cup Y') \\ (X\alpha Y) ; (X'\beta Y') = (X \cup (X' \setminus Y)) (\max\{\alpha, \beta\}) (Y \cup Y')$$

The mapping of statements to nodes is straight-forward. We map non-compound statements (such as value declarations, expression statements, or return statements) to *stmt* nodes. Compound statements are mapped to the corresponding node types; for instance, *if* statements are mapped to *if* nodes. As *if* nodes represent the alternatives as graphs, construction of *if* nodes is necessarily recursive with the procedure for linking nodes into subgraphs (§ 3.3.3). The mapping is described in Figure 5.

We also compute data dependencies for nodes as we create them. We describe the computation of data dependencies by the set of functions δ in Figure 6, mapping from expressions, nodes, and graphs to dependencies. We make use of a function *fv* mapping from expressions or statements to their free variables; it is defined in the usual way, and we omit its definition here.

In (1), we define a dependency for expressions. This definition only serves to simplify later definitions; there are no NTG nodes corresponding to expressions. In particular, the dependency for an

$$\begin{aligned}
\text{node}(\text{if } (e) \{ \vec{s} \} \text{ else } \{ \vec{s}' \}) &= \text{if } (e) \text{ graph}(\vec{s}) \text{ graph}(\vec{s}') \\
\text{node}(\text{case } (e) \{ \overrightarrow{b \ x : \vec{s}} \}) &= \text{case } (e) \{ (b_i, x_i, \text{graph}(\vec{s}_i)) \} \\
\text{node}(\text{for } (x = e[x']) \{ \vec{s} \}) &= \text{for } (x = e[x']) \text{ graph}(\vec{s}) \\
\text{node}(\text{map } (x = e[x']) \{ \vec{s} \}) &= \text{map } (x = e[x']) \text{ graph}(\vec{s}) \\
\text{node}(s) &= \text{stmt } s
\end{aligned}$$

Figure 5. The mapping from Kite statements to NTG nodes, defined by cases.

$$\delta(e) = fvs(e) \rightarrow \emptyset \quad (1)$$

$$\delta(\text{stmt } (f(\vec{e}))) = (\bigcup fvs(e_i)) \rightsquigarrow \emptyset \quad (2)$$

$$\delta(\text{stmt } (l \leftarrow f(\vec{e}))) = (l_{fvs} \cup (\bigcup fvs(e_i))) \rightsquigarrow l_{fvs} \quad (3)$$

$$\text{where } (l_{fvs}, l_{fvs}) = fvs(l)$$

$$\delta(\text{stmt } (l \leftarrow e)) = (l_{fvs} \cup fvs(e)) \rightarrow l_{fvs} \quad (4)$$

$$\text{where } (l_{fvs}, l_{fvs}) = fvs(l)$$

$$\delta(\text{stmt } (x : t = e)) = fvs(e) \rightarrow \{x\} \quad (5)$$

$$\delta(\text{stmt } s) = fvs(s) \rightarrow \emptyset \quad (6)$$

$$\delta(\text{if } (e) \ g \ g') = \delta(e) + \delta(g) + \delta(g') \quad (7)$$

$$\delta(\text{case } (e) \ \overrightarrow{(b, x, g)}) = \delta(e) + \sum ((X_i \setminus \{x_i\}) \alpha_i Y_i) \quad (8)$$

$$\text{where } X_i \alpha_i Y_i = \delta(g_i)$$

$$\delta(\text{for } (x = e[x']) \ g) = \delta(e) + ((X \setminus \{x, x'\}) \alpha Y) \quad (9)$$

$$\text{where } X \alpha Y = \delta(g)$$

$$\delta(\text{map } (x = e[x']) \ g) = \delta(e) + ((X \setminus \{x, x'\}) \rightsquigarrow Y) \quad (10)$$

$$\text{where } X \alpha Y = \delta(g)$$

$$\delta(\overrightarrow{((id, n), e)}) = \delta(n_0) ; \delta(n_1) ; \dots ; \delta(n_m) \quad (11)$$

Figure 6. The mappings from expressions, nodes, and graphs to dependencies, defined by cases.

expression statement is not (necessarily) the same as the dependency for the expression itself.

Next, we define the dependencies for statements. There are three factors that complicate this definition: first, we must determine when statements can introduce parallelism; second, there is some complexity in the dependencies of assignments; and third, the dependency of a nested node must take the dependency of its subgraphs into consideration. We address each of these concerns in order.

First, we must take account of which statements can introduce parallelism. This information is encoded in the types of expressions; however, during the rewriting stage (§ 3.3.1) we split each potentially parallel expression into its own statement. As a result, in this phase we can determine potential parallelism with simple, structural checks. Parallelism appears in NTG statements as top-level applications or as assignments from top-level applications. This is reflected by the \rightsquigarrow dependencies in (2) and (3).

Next, we address the dependencies computed for assignment statements. For assignments with simple variables on the left-hand side (LHS), this is quite simple: the free variables of the right-hand side (RHS) determine the free variables of the LHS. However, it is less simple when the LHS includes array or field selectors. Consider the statement:

$$\begin{aligned}
fvs(x) &= (\{x\}, \emptyset) \\
fvs(e.f) &= (e_{fvs}, fvs(e)) \\
&\text{where } (e_{fvs}, e_{fvs}) = fvs(e) \\
fvs(e[e']) &= (e_{fvs}, e_{fvs} \cup fvs(e')) \\
&\text{where } (e_{fvs}, e_{fvs}) = fvs(e)
\end{aligned}$$

Figure 7. The fvs function, defined by cases, returns two sets of free variables for an LHS expression, corresponding to the variables determined by and determining an assignment to that LHS.

$$x[i] \leftarrow e;$$

The evaluation of this statement depends on the free variables of e , the free variables of i , and the free variables of x (as the values after the assignment of all components of x not indexed by i are the same as they were before the statement). The statement determines the free variables of x . To express this dependency, we introduce an auxiliary function fvs , defined in Figure 7. This function computes two sets of free variables from an LHS expression, corresponding to the variables determined by and determining an assignment to that LHS. This function is used in computing the dependencies for assignment in (3) and (4).

Finally, we address nested nodes. Compound statements, such as conditionals or loops, give rise to nested task graph nodes; to compute the dependency of such a nested node, we need to know the dependency of their subgraphs. (11) gives one way to compute the dependency of a graph; composing the dependencies of all the nodes is valid because we maintain the invariant that the nodes of a graph are stored in topological sorted order. (7)–(10) use the dependencies of subgraphs when computing the dependencies of nested nodes; there is additional work in (8)–(10) to handle bound variables introduced by the compound statement.

3.3.3 Node Linking and Graph Construction

The prior section described transforming statements into graph nodes; it remains to link those nodes together to form a complete task graph. Our approach here is relatively simple, and is based solely on the dependencies of nodes without having to make further reference to the structure of statements. If a node has dependency $X \alpha Y$, we construct edges to that node from:

- The nearest earlier node that determines each variable in X , and
- All earlier nodes that depend on variables in Y .

This approach works because Kite only supports structured flow of control, and all control flow is captured in compound statements that give rise to nested graph nodes. Because we can rely on structured flow of control, a node constructed from a particular statement need never depend on the nodes constructed from statements that followed it in the source; similarly, because branches are enclosed within nested nodes, a node never needs to depend on more than one node for the value of a particular variable.

3.3.4 Task Graph Optimization

This section demonstrates the flexibility of the task-graph-based intermediate language by describing several optimizations that could be performed on task graph. While this list is hardly complete, and we hope to return to further graph transformations in future work, we believe this shows some of the potential of our present approach.

Tail-call optimization. Consider the following sequence of task invocations:

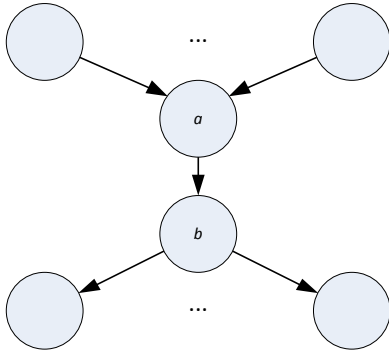
$$x \leftarrow f();$$


```

y <- g(x);
return h(y);

```

We might hope to avoid much of the overhead of the task calls in this block of code. As we invoke each task in turn (there is no opportunity for parallelism), and the input to each task is determined by the previous task, we could reuse the same stack frame (or corresponding abstraction the GPU, which lacks a traditional call stack) for each of the tasks *f*, *g* and *h*. The opportunities for this kind of optimization are evident in the task graph. In general, any time the following pattern is present, the same task frame can be used to evaluate graph nodes *a* and *b*.



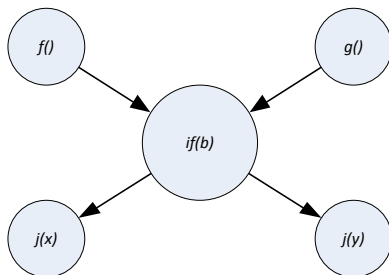
Graph flattening. While encapsulating conditionals in nested task graph nodes has advantages, as discussed in the previous section, it can also hide parallelism. Consider the following contrived example:

```

x <- f(); y <- g();
if (b) {
  x <- h(x);
} else {
  y <- h(y);
}
j(x); j(y);

```

where we assume that *b* is computed by some prior expression. We would generate the following task graph for this block:



However, note that we have lost some parallelism in this construction. Should *b* be true, the computation of *h*(*x*) has no reason to wait for the computation of *y*, and the computation of *j*(*y*) has no need to wait for the computation of *h*(*x*). A similar argument holds if *b* is false. In this case, it would be appealing to “flatten” the nested node, exposing the alternatives and increasing the parallelism. We hope to investigate approaches to this kind of flattening, and other possible applications of graph flattening, as future work.

3.4 Data Parallelism in Kite

Kite’s support for task parallelism is sufficient to encode some patterns of data-parallel computation. For instance, recursively expressed algorithms, such as many algorithms over tree-like data

structures or divide-and-conquer style algorithms, are naturally data-parallel in Kite. This section describes an additional, explicitly data-parallel construct in Kite and discusses how it differs from other features of the language and why it might be preferable in some use cases.

We begin with a motivating example. Suppose that we have four instances of the same task running in parallel, which we will refer to as tasks A through D. Each of these tasks will issue two recursive calls, creating two more tasks—A1 and A2, B1 and B2, etc. As the tasks are executing in parallel, we would expect to encounter the recursive calls at the same time in each task, so the sequence of recursive calls will be A1, B1, ..., C2, D2. We refer to this pattern as *column-major* dispatch. It might be desirable instead to treat all the recursive calls from a single task together—that is, to generate the sequence of calls A1, A2, ..., D1, D2—a pattern we call *row-major* dispatch.

Both dispatch methods can be preferable, depending on memory access patterns and SIMD-fashion execution. Continuing the earlier example, suppose that the recursive calls made by each task access adjacent areas of memory: for example each task is generating work to process a further level of a tree data structure. In that case, executing the tasks A1 and A2 simultaneously could lead to fewer memory accesses, better cache locality, or more efficient mapping of control flow to a SIMD unit. On the other hand, if each task is generating a single subtask or a set of different subtasks that would cause divergent control flow then combining the work generated by a set of tasks in column-major form would be preferable. This approach aims to support performant mappings of the language across architectures using declarative information rather than hand tuning.

Kite provides two otherwise-identical looping constructs to support row-major and column-major dispatch: the `for` and `map` statements, respectively. Both have for-each-like syntax, allowing access to the elements of a collection and their indices while excluding the possibility of off-by-one errors. For example, if *v* is a collection of `ints`, the following code initializes each element of the collection to its index:

```
for (x = v[i]) { x <- i; }
```

Note that *x* is an alias to an element of the collection, so *x* does not require new allocation and assignments to *x* update the collection itself. As indicated by (9, Figure 6), the `for` loop does not of itself introduce new parallelism, so this loop will run in sequence within the enclosing task. On the other hand, as indicated by (10, Figure 6), the `map` loop does introduce potential parallelism regardless of the parallelism of its body. So, the loop

```
map (x = v[i]) { x <- i; }
```

could execute in parallel, even though the loop body contains no task applications or other parallelism-introducing constructs.

Kite’s looping constructs can be used to introduce row- and column-major dispatch over collections. For example, consider the following loops:

```
for (x = v[i]) { f(x); }
map (x = v[i]) { f(x); }
```

The `for` loop invokes *f* on each element of *v*, in sequence. If multiple instances of the loop were running on different collections in parallel, all the work for the first elements of the respective collections would be invoked first, then the work for the second elements, and so on. This results in column-major dispatch. On the other hand, the `map` loop begins by introducing one task for each element of the collection. These tasks are dispatched in parallel. Each of those tasks, in turn, invokes the work for that element, leading to row-major dispatch. While it might seem that we have

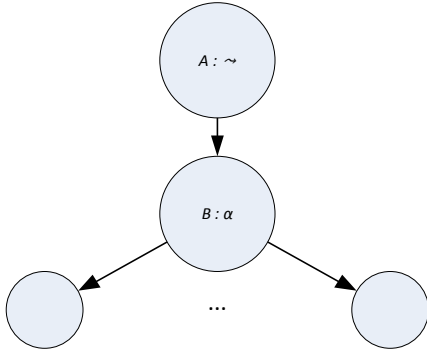
also introduced extra work, in the form of the extra task per collection element, we would expect that the optimizations discussed in § 3.3.4 would eliminate much of the overhead.

4. Kite Implementation

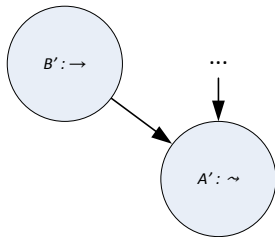
4.1 Kite-to-OpenCL Translation

The previous section described a translation from Kite programs to nested task graphs. Unfortunately, this translation is not enough to describe the execution of Kite programs in a GPGPU environment. In particular, we cannot be sure that all the inputs to a particular graph node will be available at the same time. To avoid the need for a blocking wait, we use a translation inspired by continuation-passing style to collect the arguments to a particular graph node. Once all the arguments are available, the node can be evaluated.

Intuitively, our transformation works as follows. First, we identify locations at which CPS translation is necessary, by looking for the following pattern in the generated task graph (where we annotate the nodes by the arrow in their dependencies):



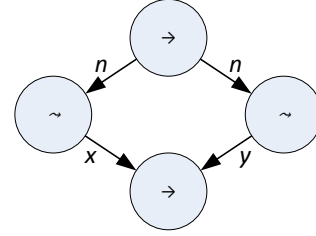
This represents a case where the remainder of the computation (the tree below node B) has to wait for the result of the computation in node A . Here, we generate a new node, B' , which contains the old node B and the subgraph below it. We call this node the *continuation*. We then perform a standard continuation-passing style transformation on A giving A' : instead of returning, it takes a continuation parameter and invokes that continuation with its result. The resulting graph resembles the following:



Regardless of the original annotations in the subgraph below B , the annotation of B' is \rightarrow : building a continuation does not introduce any parallelism. As a result, we have eliminated the need to wait for the result of A . By iteratively applying this transformation from the bottom of the task graph up, we can eliminate all points where one computation might have to wait for another to complete. At that point, the program can be translated into OpenCL. While intuitively simple, there are a number of issues that arise in performing this translation. The remainder of this section discusses some of those issues.

Multiple inputs. The previous section assumed that there was a single value flowing into B ; the more common case in actual Kite

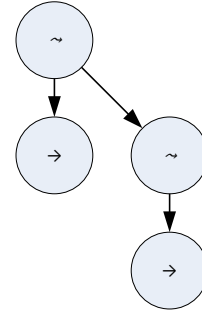
programs is that there are multiple parallel computations that “join” at later nodes. For instance, the Fibonacci example discussed earlier (§ 3.3) would give rise to the following graph:



The bottom node of the graph has to wait for two parallel computations to finish; we encode this by creating a two-parameter continuation. This introduces an additional twist: each parallel computation has to write its result into the appropriate slot of the continuation; however, the parallel computations are both instances of the same task. We capture this by introducing a new type, called a *result*, that pairs a continuation with an index into its argument vector. The CPS transformation of tasks transforms them to take result arguments rather than continuation arguments. When a task completes, it writes its return value into the argument slot indicated by its result parameter and runs the continuation only if all its argument slots are filled.

Multiple outputs. It is also likely that parallel nodes have multiple outputs. Consider the following code fragment and associated task graph:

```
y <- f(x);
print(y);
z <- f(y);
print(z);
```



There are two nodes waiting for the result of the first f computation, while only one waits for the result of the second f computation. We handle this by counting the number of outputs from each node that calls a given task before doing the CPS transformation of that task; we then add enough result parameters to the task to handle the highest number of outputs. When transforming calls to the task, we then insert a dummy result argument if there are fewer outputs at a particular call than the task has result parameters.

4.2 OpenCL Runtime Implementation

In developing this paper we have focused on the Kite programming model and its translation to OpenCL. However, to avoid omitting all runtime details, the remainder of this section provides a short synopsis of our current implementation.

OpenCL limits task scheduling of tasks to command queues maintained on the host; once kernel code is executing on the device no additional scheduling or load balancing can be performed. Further, encoding a scheduler in OpenCL is complicated by the lack of function pointers. We address these limitations with the technique of persistent threads [1, 25] and uber-kernels (a form of defunctionalization [24]), respectively. The remainder of this section outlines these approaches.

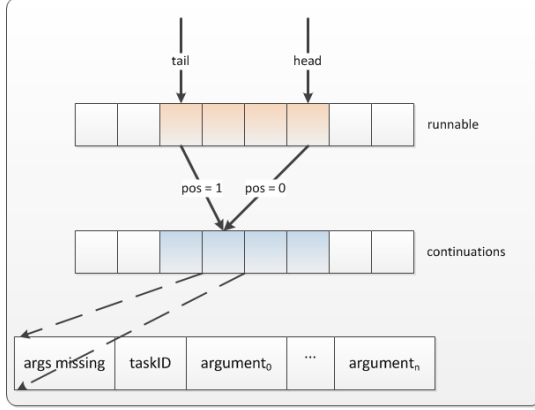


Figure 8. CPU/GPU queue memory layout

Persistent threads. While OpenCL supports fine-grained parallelism, it does not support pre-emption and relies on hardware scheduling of execution grids far larger than the actual parallelism of the underlying device; this means that, if not carefully coded, a software scheduler may never get to run. Note that, as this is part of the OpenCL execution model, these observations are true for both the CPU and GPU devices, as the CPU device is exposed as a single aggregate device. The persistent thread model addresses this by launching just enough work to ‘fill’ the machine, i.e. an implementation can assume the invariant that all launched threads will execute concurrently and thus it is possible to perform global communication while maintaining forward progress. OpenCL does not directly expose support for ‘filling’ the machine and instead we calculate semi-automatically the appropriate number of task-scheduler ‘uber-kernel’ instances (described below) using either the Device Fission extension, that allows a CPU device to be subdivided to individual cores [3], or using GPU vendor tools including AMD’s profiler and NVIDIA’s occupancy calculator.

The runtime implementation itself is based on a standard work-stealing scheduler and uses variants of the Chase-Lev algorithm [7]. The CPU implementation is a direct implementation of this algorithm, maintaining a single work-queue for each core in the system and the scheduler supports stealing from both the top and bottom of the task graph, accounting for differences in cache configuration. As described previous (§ 4.1) our Kite-to-OpenCL translation is based on continuations and the runtime maintains runnable and continuation queues, laid out in memory as per Figure 4.1. The GPU implementation is complicated by a desire to increase the amount of vector efficiency (i.e. WAVEFRONT utilization) and our implementation extends the original Chase-Lev algorithm in two ways: firstly we have extended it to allow a vector width’s chunk of work to be consumed, by adding a consuming edge to the lock-free implementation to guarantee atomic access; secondly we maintain two versions of the queues, one in global memory and one in local memory. The local memory queue is accessible to a single compute device only and thus work enqueued there cannot be stolen by other compute units. A consequence of this choice is that if this queue is too big then load-imbalance can become an issue. To date we have found that a size of 32 wavefront-width tasks seems to be work well for a number of applications, however, we have found others that behave badly with this limit and we do not currently have an automatic way to derive a sensible value.

It should be noted that it is possible in this implementation for work to be stolen by the device from the host and vice versa using shared, mapped memory regions to contain queues that are accessed by persistent threads on any OpenCL device. In this fash-

ion an entire heterogeneous system can load balance by moving tasks between a set of queues spread over its constituent devices. Optimal behaviour for this heterogeneous task stealing remains to be analysed as future work; in particular, the trend towards high-performance desktop Fusion architectures starting aggressively in the next year will decrease overhead from shared queues and allow more fine-grained task interaction than is currently practical.

Uber-kernels. Given a set of Kite tasks t_1, \dots, t_n , compiled to OpenCL, using the translation described in § 4.1, as tcl_1, \dots, tcl_n , an uber-kernel encapsulating these tasks is defined by:

```
void runTask(
    GlobalScope globalScope,
    struct Task task)
{
    switch (task.id) {
        case t_1_ID:
            tcl_0;
            ...
        case t_n_ID:
            tcl_n;
    }
}
```

To run a task the scheduler dequeues a task and passes it along with some encapsulated state to the runTask, and takes the form:

```
Task t = popBottom(lwsdeque);
while (!isTaskEmpty(t)) {
    runTask(globalScope, t);
    t = popBottom(lwsdeque);
}
```

The persistent threading model is quite general, and work-stealing is only one possible task queueing approach. We intend to develop alternative approaches, such as maintaining one queue per task type, and study their impacts when used on a GPGPU architecture.

5. OpenCL Tasklets

While we believe Kite is interesting as a programming language in its own right, it can be difficult to get developers to adopt new languages. To address this concern we use Kite to formulate an extension to core OpenCL supporting light-weight tasks, called Tasklets.

The `_task` (or `task`) qualifier declares an asynchronous function to be a future that can be executed by an OpenCL device(s).

A task returning a result, i.e. a future, is spawned with the following syntax:

```
identifier <- identifier (expr_1, ..., expr_n);
```

For a task that does not return a value or the user does not intend to use the return value, the following syntax is provided:

```
() <- identifier (expr_1, ..., expr_n);
```

We tried a number of approaches to avoid exposing any notion of continuations in the OpenCL C language but again and again came across the issue of escaping variables. At first, this might not seem like a problem as variables local to a work-item (i.e. private memory) can be stored in a continuation task by value, global memory is shared by all work-items across the system and so storing references to this memory introduces no additional burden. The problem is local memory usage, which is shared only by collection of work-items running on the same compute unit. Continuation tasks, due to stealing, may not run on the the same compute unit, and furthermore, local memory is shared by all tasks running on a particular compute unit.

We address the issue of escaping values with the following syntactic construct:

```
force(var_1, ..., var_n) {
  statement-list
}
```

where `var_1, ..., var_n` are variables defined within the surrounding context and the only free variables in the block are defined globally. A `force` block is assumed to always return from the enclosing task that it is defined in. Thus the `force` construct is an explicit representation of Kite's implicit continuation-passing implementation. Like Kite's implementation, continuations are only used when necessary and the cost of building closures is only incurred when used explicitly by the programmer.

The `force` block is closely related to use of delegate functions in TPL, which are used to represent tasks. Delegates in TPL capture free variables of tasks, as per our `force` construct, allowing parallelism to be introduced in a compositional manner. We considered not requiring `force` blocks to always return, but this breaks with the generation of a compositional semantics, making the resulting code hard to reason about.

6. Conclusion

This paper has explored an alternative programming model, based on braided parallelism and targeting the GPGPU architecture. Existing GPGPU programming models are data-parallel, generalized versions of OpenGL and DirectX, finding application in domains which contain large amounts of regular flat data-parallelism. We have motivated Kite's development from a desire to widen the set of potential applications for the emerging heterogeneous computing era. Using OpenCL as our target language, we have described what we believe to be the first general purpose programming language for braided parallelism that targets the GPGPU programming model.

Differences in core design and latency of communication between cores both become important considerations in managing queues. As we saw with the `for` and `map` constructs in § 3.4, there is reason to consider how the structure of an execution maps to the underlying hardware. When we have a system with heterogeneous cores we must also consider where a series of sparked tasks will be placed to make efficient use of the device, and the relative cost of task stealing across different communication latencies. A narrow execution with little data parallelism might be better served by running on the high speed scalar CPU core, while a wider dispatch generated by a `map` might be a good trigger to move work over to a throughput oriented GPU core.

Finally, many existing higher-level languages rely on garbage collection to ensure memory safety and simplify programming models. If GPGPU computing is to be successful in the mainstream, then addressing complexities in existing GPGPU memory models and providing simplified, garbage-collection-based memory models must will be increasingly important.

References

- [1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, New York, NY, USA, 2009. ACM.
- [2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.
- [3] G. Bellows, B. R. Gaster, A. Munshi, I. Ollmann, O. Rosenberg, and B. Watt. Device fission. OpenCL Working Group, Khronos, 2010.
- [4] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.
- [5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.
- [6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.
- [7] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM.
- [8] S. Conchon and F. Le Fessant. Jocaml: mobile agents for Objective-Caml. pages 22–29. IEEE Computer Society, 1999.
- [9] C. Fournet and G. Gonthier. The Join Calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 268–332, London, UK, 2002. Springer-Verlag. ISBN 3-540-44044-5.
- [10] Q. Hou, K. Zhou, and B. Guo. SPAP: A programming language for heterogeneous many-core systems, 2010.
- [11] S. G. Itzstein and D. Kearney. Applications of Join Java. *Aust. Comput. Sci. Commun.*, 24(3):37–46, 2002.
- [12] A. Lefohn, M. Houston, C. Boyd, K. Fatahalian, T. Forsyth, D. Luebke, and J. Owens. Beyond programmable shading: fundamentals. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–21, New York, NY, USA, 2008. ACM.
- [13] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 227–242, New York, NY, USA, 2009. ACM.
- [14] M. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004.
- [15] M. McGuire and D. Luebke. Hardware-accelerated global illumination by image space photon mapping. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 77–89, New York, NY, USA, 2009. ACM.
- [16] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.
- [17] NVIDIA Corporation. NVIDIA CUDA programming guide, version 3.1, 2010.
- [18] M. Odersky. Functional nets. *Lecture Notes in Computer Science*, 1782, 2000.
- [19] OpenCL Working Group. The OpenCL specification, version 1.2, revision 15. Khronos, 2011.
- [20] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: a general purpose ray tracing engine. In *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, pages 1–13, New York, NY, USA, 2010. ACM.
- [21] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM.
- [22] J. Reinders. *Intel Thread Building Blocks*. O'Reilly & Associates, Inc., 2007.
- [23] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.
- [24] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference—Volume 2*, pages 717–740, Boston, Massachusetts, United States, 1972. ACM.
- [25] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the gpu. In M. Doggett, S. Laine, and W. Hunt, editors, *High Performance Graphics*, pages 29–37. Eurographics Association, 2010.