Parakeet: GPU Acceleration of Dynamic Array Languages

Alex Rubinsteyn
alexr@cs.nyu.edu
New York University

Eric Hielscher
hielscher@cs.nyu.edu
New York University

Dennis Shasha
shasha@cs.nyu.edu
New York University

Abstract

Contemporary GPUs offer the potential for substantial performance improvements on general purpose computation – from an average 2.5X speedup over CPUs on 14 standard "throughput computing" benchmarks [16] to 80X on data parallel workloads [10]. However, the commonly used frameworks for general purpose GPU (GPGPU) programming, OpenCL [20] and NVIDIA's CUDA [21], require highly specialized low level programming and performance tuning knowledge to tap into this potential.

There has been much recent work on enabling GPU acceleration of higher level languages to ease GPGPU programming and to bring this performance potential to a wider audience. However, these projects typically (1) are intimately tied to a specific source language; (2) expose a constrained or non-idiomatic programming model; and (3) suffer from long GPU compile times, inhibiting rapid prototyping.

We present Parakeet, an intelligent runtime library and JIT compiler for array-oriented subsets of existing high level languages. Parakeet dynamically translates array-oriented code into a typed intermediate language and performs optimizations that eliminate costly abstractions and increase computational density. Parakeet then executes its intermediate representation by quickly synthesizing and executing data parallel GPU programs. Parakeet's runtime transparently handles data movement, GPU memory space selection, data layout, as well as garbage collection. Parakeet is attached as a library to the source language's interpreter, allowing all of the source language's standard features and tools to be used.

We evaluate Parakeet on two standard benchmarks: Black-Scholes option pricing, and K-Means clustering. We compare high level array-oriented implementations to hand-written, tuned GPU versions from the CUDA SDK [22] and the Rodinia GPU benchmark suite [10]. Despite having orders of magnitude shorter source code, the high level versions perform very favorably when executed by Parakeet, in some cases even faster than hand-written CUDA code.

1. Introduction

The ubiquity and performance potential of modern GPUs has led to much interest in enabling their use for the execution of general-purpose programs. Unfortunately, the two widely used GPGPU frameworks – OpenCL [20] and NVIDIA's CUDA [21] – require the programmer to have extensive knowledge of low level architectural details in order to fully harness this performance potential. Hence, many recent projects attempt to lower the barrier of entry to programming GPUs by allowing the use of high level languages [7–9, 18, 26, 27].

GPUs are able to achieve impressive performance because they have been highly specialized for data parallel workloads. Array languages are thus well suited for high level GPU programming due to their idiomatic preference for data parallel array operations instead of explicit loops ("collection-oriented" [25] programming). We observe that many of the above projects are structured around the use of data parallel array operators such as map and reduce.

In this paper, we present Parakeet, an intelligent runtime for executing high level array programs on GPUs. Parakeet is neither a new programming language, nor is it tied to any specific language. The Parakeet library is designed to accelerate the array oriented constructs of existing dynamic languages. A key objective is to harness the elegance and parallelizability of functional array operators while sacrificing as little programmer convenience as possible. Specifically, Parakeet supports the use of array language semantics such as scalar promotion as well as a restricted use of mutable state.

We have implemented our first Parakeet front end for Q [5], a descendant of APL that is widely used in financial computing. Q is particularly amenable to GPU acceleration by Parakeet as its use of array operators is very extensive – its idiomatic style makes sparser use of loops than even Matlab [19] or Python's NumPy extensions [23]. We are also developing a front end for NumPy.

The main contributions of this paper are the following:

- A detailed analysis of the constraints imposed by graphics hardware and their implications for the implementation of high level languages.
- A demonstration that fully dynamic GPU compilation can be realized with minimal overhead, and a discussion of the opportunities that opens up.
- An intermediate language that is both sufficiently expressive to capture significant subsets of high level languages, while being restricted to constructs which allow for compilation to efficient GPU programs.
- A working system in which programmers can write complex algorithms in existing high level languages that is automatically parallelized into efficient GPU programs.

2. Overview

Parakeet is designed as an accelerator library for dynamic languages which possess either intrinsic array-oriented semantics or expressive array libraries. Parakeet acts as an interpreter nested within its source language's execution context, which allows the programmer to continue using all of the source language's normal tools and support libraries.

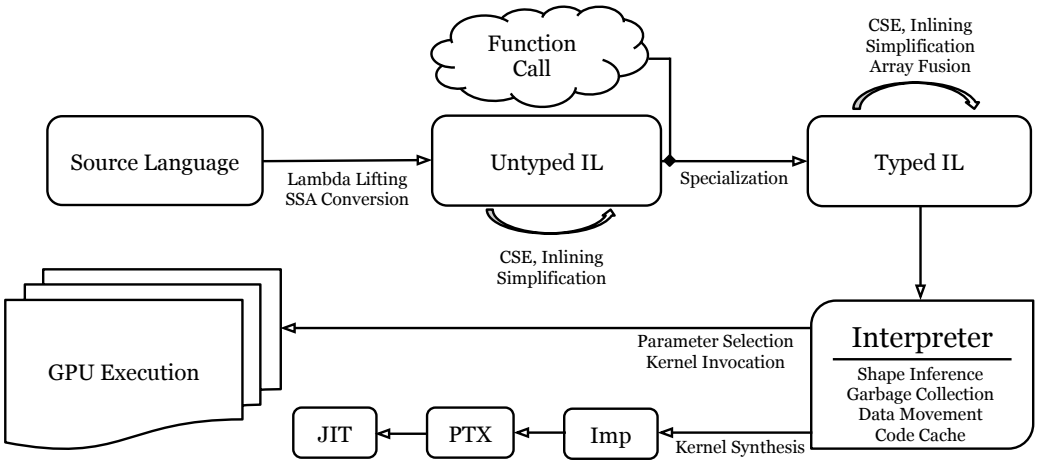

Figure 1. Overview of Parakeet

The program execution pipeline for our system (Figure 1) begins in the standard interpreter of the source array language. At program start time, a subset of the source language's functions (either annotated manually by the programmer or detected automatically as parallelizable by Parakeet) are registered with Parakeet. The body of a registered function is then translated into an untyped intermediate representation using the Parakeet front end interface and the source language's introspection facilities.

When a call is made to a function which has been registered with Parakeet, the untyped function is specialized by propagating type information from the arguments to all values in the body. Type specialization translates the function into a typed intermediate language. Further standard optimizations are performed at this stage, the most impactful of which is *array operator fusion*, wherein array operators are combined according to rewriting rules. This fusion step can be extremely beneficial to the final GPU program's performance, since it can potentially drastically improve the computational density of GPU programs and eliminate many wasteful array temporaries.

Execution of the optimized typed IL is initiated by Parakeet's interpreter, which is then responsible for offloading certain operations onto the GPU. When the interpreter encounters an array operator it employs a simple cost-based heuristic (which considers nested array operators, data sizes, and memory transfer costs) to decide whether to execute that array operator on the GPU or CPU (for more details see Section 6.1). Parakeet's interpreter invests great effort into analyzing the both the program and information about the data to make dynamic execution decisions. The cost of these analyses are mitigated by the tremendous computational density of the intermediate language's array operators.

If an array operator's computation is deemed a good candidate for GPU execution, Parakeet flattens all nested array computations within that operator into sequential loops. This payload is then inlined into a GPU program skeleton that implements that operator. For example, in the case of a **map** operation, Parakeet provides a skeletal algorithm that implements the pattern of applying the same function to each element of an array on the GPU. The flattened payload function argument is inlined into this skeleton, and a complete GPU program is synthesized and JIT compiled.

To execute the GPU program, Parakeet first copies any of its inputs that aren't already present on the graphics card to the GPU's memory, which Parakeet treats as a managed cache of data present in the CPU's RAM. The GPU program is then executed, with its output lazily brought back to the CPU either when it is needed or when Parakeet's GPU garbage collector reclaims its space.

3. GPU Hardware

To set the stage and motivate the design choices of Parakeet, we discuss here some important features of modern GPU hardware. A GPU consists of an array of tens of multiprocessors, which contain various resources such as local memories and instruction issue units shared among simple cores. Since the issue units are shared, threads running on a single multiprocessor execute instructions in lockstep – hence the execution model's name: Single Instruction Multiple Thread (SIMT). A typical programming pattern is to write short programs which are executed in parallel by thousands of lightweight threads running on these hundreds of simple cores, each operating on different subsets of input data. When all these cores are well utilized, a graphics card may attain a peak throughput of many hundreds of GFLOP/s – an order of magnitude more than typical high end CPUs.

SIMT differs from the more common Single Instruction Multiple Data (SIMD) model in that branching instructions are allowed whose branch conditions aren't uniformly met among threads within a single multiprocessor, allowing co-located threads to execute divergent code paths. This improves the ease of programming a GPU, but divergent branching incurs a very expensive performance penalty as each thread effectively execute no-ops along the irrelevant code paths. This illustrates a common point in GPGPU programming: the hardware allows for somewhat expressive programming styles, but failure to match what the hardware actually does well results in very inefficient execution.

Graphics cards have high peak memory bandwidth – well over 100GB/sec is common. However, in order to achieve this high bandwidth (which is essential to achieving peak performance), nearby threads must access memory in particular, regular patterns. Random memory access can be over an order of magnitude slower than linear stride access. To alleviate some of this performance bottleneck, the GPU also provides several other memory spaces with varying performance characteristics and preferred access patterns. These memory spaces include some read-only cached space (called *texture* and *constant* memory) and some programmer-managed multiprocessor-local fast memory called *shared memory*. Efficient manual use of these memory spaces can be cumbersome, but is also essential to good performance for most workloads.

3.1 Limitations Imposed by GPUs

GPUs are able to achieve their specialized high performance because they have been optimized for data parallel workloads. Data parallelism is widely found in typical graphics applications that perform simple operations on large amounts of pixel or triangle data, and so is a natural choice for execution on graphics acceler-

ators. However, architectural optimization for data parallelism carries with it various restrictions on the types of code and programming models that naturally fit the GPU architecture:

- **Flat, unboxed data representation**. GPU hardware is optimized to utilize memory in highly structured access patterns (so-called "coalescing"). The use of boxed or indirectly accessed data leads to unstructured memory accesses and results in severe performance degradation.
- **No polymorphism**. GPUs generally share instruction dispatch units among many concurrently executing threads. When a group of threads "diverge", meaning that they take different branches through a program, their execution must be serialized. Thus it is important to eliminate as many sources of runtime uncertainty as possible. In particular, type-tag dispatch commonly used to implement polymorphic operations would incur unacceptable costs if translated naively to the GPU.
- **No function pointers**. Most GPUs (excluding the recently released NVIDIA Fermi architecture) do not support the use of indirect jumps or function pointers. In fact, a common implementation strategy for GPU function calls is extensive inlining. Even in the case where function pointers are theoretically supported, they require every possible function to be transferred to the GPU and incur overhead due to potentially unpredictable branching.
- **No global communication**. Synchronization in a GPU computation is limited to local neighborhoods of threads. Various schemes have been devised for achieving global synchronization between all executing threads [12], but these schemes are all either slow or unsafe. This implies that the use of shared mutable state is a large hindrance to effective utilization of GPU resources.
- **All data must be preallocated**. The current generation of GPUs lack any mechanism for dynamically allocating memory. Any heap space required by a GPU computation (for output or temporary values) must be allocated beforehand.

With these constraints in mind, we turn to efficient high level abstractions chosen to fit them.

4. Array Language Programming with Q

Array programming languages, as mentioned in Section 1, include native support for array creation and manipulation via bulk array operators. These array operators, such as **map**, typically have natural data parallel implementations which plays well to the strengths of GPUs. A key contribution of Parakeet is to provide a compiler framework that capitalizes on this strength while respecting the constraints necessary to maintain good performance.

While the Parakeet framework is built to be agnostic to the source array language, we chose to implement its first front end for Q, a high-level, sequential array programming language from the APL family [5]. Q is dynamically typed and uses native array types and a rich set of array operators and higher-order data parallel function modifiers that map well onto the Parakeet array operators. Q is also a fully-featured language, with a large library of built-in functions, and is fast enough to support intraday trading in the financial computing domain. Since the focus of this paper is the Parakeet runtime and compiler, we omit many details of the Q language. We present only the salient features for illustrating the programming style and Q's support for array operators.

4.1 K-Means Clustering Example

We illustrate the relevant features of array programming in Q with an example: an implementation of K-Means clustering, a widely used unsupervised clustering algorithm. In Figure 2, we see an implementation of K-Means in Q. Five functions are defined using Q's brace notation for surrounding function bodies and the colon operator for assignment. For example, the `calc_centroid` function on line 1 takes three parameters `X`, `a`, and `i` (used as the input data matrix, the current assignment vector, and the scalar index of the centroid to calculate, respectively), and calculates that cluster's current centroid.

```
1  calc_centroid:{[X;a;i] avg X[where a = i]}
2  dist:{[x;y] sqrt sum (x-y) * (x-y)}
3  minidx:{[C;x] ds: dist[x] each C;
4    ds ? min ds }
5
6  kmeans:{[X;k;a]
7    C: calc_centroid[X;a] each til k;
8    converged: 0;
9    while[not converged;
10     lastAssignment: a;
11     a: minidx[C] each X;
12     C: calc_centroid[X;a] each til k;
13     converged: all lastAssignment = a];
14   C}
```

Figure 2. K-Means Clustering implemented in Q

This function illustrates some of the array-oriented features of Q. First, Q allows implicit mappings of functions element-wise across vectors – for example, to test element-wise equality between the assignment vector `a` and the scalar index `i`, we simply write `a = i`. In addition, this showcases Q's support for *scalar promotion*. Semantically, Q implicitly promotes `i` to be a vector of the value of `i` repeated a number of times equal to the length of `a` and computes the element-wise equality of these two vectors.

We then generate the list of indices we want by applying the built-in `where` operator to the result of this test, and index into the data matrix `X` to get the list of data points in the centroid. The result of this indexing is itself a 2-D matrix which contains the list of data points belonging to the `i`-th cluster. Finally, the built-in `avg` function – a reduction operator – gets applied to this 2-D list. Thus we see that reductions and other built-in array operators can be applied to arrays of any arity. Further array built-ins used in K-Means include `sum`, `min`, and the find operator (denoted by '`?`').

In this algorithm we also see a number of higher-order data-parallel function modifier keywords, which in Q are called *adverbs*. For example, we calculate a cluster's new centroid by using the `each` keyword (which applies a function elementwise to its argument) to apply the `calc_centroid` function to an array of integers from `0` to `k-1`.

In this example it is evident that array programming is both expressive and compact. While the CUDA implementation of K-Means in the Rodinia benchmark suite is hundreds of lines of code long, our Q version is only 14.

5. Translation, Specialization, and Optimization

Now we turn to the problem of compiling an array language program into an efficient GPU program. At first glance, there seems to be a significant mismatch between the highly dynamic expressiveness of an array language like Q and the limitations imposed by GPU hardware discussed in Section 3.1. Indeed, the Parakeet intermediate language must serve as a compromise between two competing tensions. In order to translate array programs into efficient GPU code it is necessary for the compiler to eliminate as much abstraction as possible. On the other hand, we must be careful not to make our program representation overly concrete with re-

gard to evaluation order (which would eliminate opportunities for parallelism provided by the array operators).

In deference to the above-mentioned GPU hardware restrictions, any program which is compiled to run on the GPU cannot have any of the following:

- Polymorphism (over types, array ranks, etc..)
- Recursion
- User-specified higher-order functions
- Compound data types other than arrays

Some of these restrictions (such as recursion, use of compound data types) limit the set of programs which can be executed by Parakeets. Certain types of polymorphism and some uses of higher order functions, however, can be effectively eliminated by program transformations. We enumerate some of these transformations (such as lambda lifting and specialization) which help bridge the abstraction gap between an expressive dynamically typed language and the GPU hardware. We also demonstrate several optimizations which, while beneficial in any compiler, are particularly important when targeting a graphics processor. Seemingly small residual inefficiencies in our intermediate form can later manifest themselves as the creation of large arrays, needless memory transfers, or wasteful GPU computations.

To help elucidate the different program transformations performed by Parakeet, we will show the effect of each stage on a distance function defined in Q, shown in Figure 3.

```
dist: {[x;y] sqrt sum (x-y) * (x-y)}
```

Figure 3. Distance Function in Q

5.1 Lambda Lifting and SSA Conversion

After a function call has been intercepted by the Parakeet runtime, Parakeet performs a syntax-directed translation from a language-specific abstract syntax tree (AST) into Parakeet's IL. Since type information is not yet available to specialize user functions, the functions must be translated into an untyped form (by setting all type assignments to $\perp$). The translation into Parakeet's IL maintains a closure environment and a name environment so that simultaneous lambda lifting and SSA conversion can be performed.

Since we would like to interpret our intermediate language we use a gated SSA form based on the GSA sub-language of the Program Dependence Web [24]. Classical SSA cannot be directly executed since the ϕ-nodes lack deterministic semantics. Gated SSA overcomes this limitation by using "gates" which not only merge data flow but also associate predicates with each data flow branch. Aside from simplifying certain optimizations, these gates also enable us to execute our code without converting out of SSA. For the sake of clarity, we elide SSA gates from the specifications of our untyped and typed intermediate languages.

Figure 4 shows the `dist` function after it has been translated to Untyped SSA.

5.2 Untyped Optimizations

Parakeet performs optimizations both before and after type specialization. We subject the untyped representation to inlining, common subexpression elimination and simplification (which consists of simultaneous constant propagation and dead code elimination). This step occurs once for each function, upon its first interception by Parakeet. It is preferable to eliminate as much code as possible at this early stage since an untyped function body serves as a template for a potentially large number of future specializations. The

$$
\begin{aligned}
&\text{dist (x, y)} \rightarrow \text{(z)} = \\
&\quad \mathrm{t}_1 = \mathrm{x} - \mathrm{y} \\
&\quad \mathrm{t}_2 = \mathrm{x} - \mathrm{y} \\
&\quad \mathrm{t}_3 = \mathrm{t}_1 * \mathrm{t}_2 \\
&\quad \mathrm{t}_4 = \mathrm{sum}(\mathrm{t}_2) \\
&\quad \mathrm{z} = \ \mathrm{sqrt}(\mathrm{t}_4)
\end{aligned}
$$

Figure 4. Untyped Distance Function in SSA form

only optimizations we do not perform on the untyped representation are array fusion rewrites, since these rely on type annotations to ensure correctness.

In our distance example, untyped optimizations will both remove a redundant subtraction and inline the definition of the sum function, which expands to a **reduce** of addition.

$$
\begin{aligned}
&\text{dist (x, y)} \rightarrow \text{(z)} = \\
&\quad \mathrm{t}_1 = \mathrm{x} - \mathrm{y} \\
&\quad \mathrm{t}_2 = \mathrm{t}_1 * \mathrm{t}_1 \\
&\quad \mathrm{t}_3 = \textbf{reduce}(+, 0, \mathrm{t}_2) \\
&\quad \mathrm{z} = \ \mathrm{sqrt}(\mathrm{t}_3)
\end{aligned}
$$

Figure 5. Distance Function after untyped optimizations

5.3 Specialization

Specialization transform user programs from the untyped intermediate language into its typed variety. The purpose of this transformation is to eliminate polymorphism, to make manifest all implicit behavior (such as coercion and scalar promotion), and to assign simple unboxed types to all data used within a function body. Beyond the fact that the GPU requires its programs to be statically typed, these goals are also essential for the efficient execution of user code on the GPU. Thus, the specializer generates a different specialized version of a function for each distinct call string, with all of the function's variables receiving the appropriate types.

A specification of our intermediate language is shown in Figure 6. We take inspiration from [4] and allow functions to both accept and return multiple values. This feature simplifies the specification of certain optimizations and naturally models the simultaneous creation of multiple values on the GPU. By convention we will write $\overline{\tau_m}$ to denote a sequence of m simple types, $\overline{v_n}$ for a sequence of n values, etc. A single element is equivalent to a sequence of length 1 and sequences may be concatenated via juxtaposition, such that $\tau, \overline{\tau_n} = \overline{\tau_{m+1}}$.

The array operators **map**, **reduce**, and **scan** form a carefully confined higher-order subset of our otherwise first-order language and thus we elevate them to primitive syntax. These operators are important since they are the only constructs in our language that we attempt to parallelize automatically through GPU code synthesis. This isn't to say these are the only constructs executed in parallel on the GPU. The simple (first-order) array operators such as **sort** can be executed in parallel on the GPU as well via a fixed parallel standard library.

It is important to note that uncertainty with regard to function values has been made completely explicit in this intermediate language. Every higher-order operator carries known labels for its function arguments. Any uncertainty in the original source must be directly encoded as an enumeration over a finite set of possible function calls. Function labels are associated with a set of closure argument values, which are necessary in order to support partial function application. This representation of function values re-

Typed First-Order IL

program	p	::=	$d_1 \cdots d_n$
definition	d	::=	$f_i(x^m : \tau^m) \to (y^n : \tau^n) = \overline{s}$
statement	s	::=	$x^m : \tau^m = e$
		|	if v then $\overline{s}$ else $\overline{s}$
		|	while $\overline{s}, x_{cond}$ do $\overline{s}$
expression	e	::=	values$(\overline{v})$
		|	cast$(v, \tau_{src}, \tau_{dest})$
		|	$\text{prim}_{\langle \oplus \rangle}(\overline{v})$
		|	$\text{call}_{\langle f \rangle}(\overline{v})$
		|	$\textbf{map}_{\langle c \rangle}(\overline{v})$
		|	$\textbf{reduce}_{\langle c_i, c_r \rangle}(\overline{v_i}, \overline{v})$
		|	$\textbf{scan}_{\langle c_i, c_r, \rangle}(\overline{v_i}, \overline{v})$
value	v	::=	numeric constant
		|	x (data variable)
closure	c	::=	$f \times \overline{v}$
type	τ	::=	$bool \mid int \mid float \mid \textbf{vec}\ \tau$

Figure 6. Typed First-Order Language

sembles defunctionalization combined with specialization of *apply* functions [28].

In order for higher-order functions to be translated into the Parakeet IL, it must be possible to specialize away all higher-order arguments. There are situations where exhaustive specialization is not possible (e.g. combinator libraries); in these cases, we revert to the source language's interpreter for execution. This highlights an important point: since Parakeet augments (but does not replace) the interpreter of the source language, we are free to disallow problematic language constructs.

To continue the example, if the *dist* function is called with arguments of type $vec\ float$ the specializer will then generate the code shown in Figure 7.

dist (x : $vec\ float$, y : $vec\ float$) $\to$ (z : $float$) =
  t_1 = **map**($-_{\text{float}}$, x, y)
  t_2 = **map**($*_{\text{float}}$, x, y)
  t_3 = **reduce**($+_{\text{float}}$, 0, t_2)
  z = sqrt(t_3)

Figure 7. Distance Function After Specialization

The actual intermediate language associates type annotations with every binding, which we elide here for clarity. Note that the polymorphism inherent in math operations between dynamically typed values has been removed through the use of statically typed math operators, and implicit **map**s on vectors (such as the subtraction between x and y) have been expanded and made explicit.

5.4 Array Operator Fusion

In addition to standard compiler optimizations (such as constant folding, function inlining, and common sub-expression elimination), we employ fusion rules [14] to combine array operators. Fusion enables us to minimize kernel launches, boost the computational density of generated kernels, and avoid the generation of unnecessary array temporaries.

We present the fusion rules used by Parakeet in simplified form, such that array operators only consume and produce a single value. Our rewrite engine actually generalizes these rules to accept functions of arbitrary input and output arities.

Map Fusion

map$(g, \textbf{map}(f, x)) \rightsquigarrow \textbf{map}(g \circ f, x)$

Reduce-Map Fusion

reduce$(g_r, g_i, \textbf{map}(f, x)) \rightsquigarrow \textbf{reduce}(g_r \circ f, g_i \circ f, x)$

These transformations are safe if the following conditions hold:

1. All the functions involved are referentially transparent.
2. Every output of the predecessor function (f) is used by the successor (g).
3. The outputs of the predecessor are used *only* by the successor.

The last two conditions restrict our optimizer from rewriting anything but linear chains of produced/consumed temporaries. A large body of previous work [1] has demonstrated both the existence of richer fusion rules and cost-directed strategies for applying those rules in more general scenarios. Still, despite the simplicity of our approach, we have observed that many wasteful temporaries in idiomatic array code are removed by using only the above rules.

In Figure 8, we see the resulting optimized and specialized `dist` function. The two **map**s have been fused into the **reduce** operator, with a new function f_1 generated to perform the computation of all three original higher-order operators.

f_1(acc : $float$, x : $float$, y : $float$) $\to$ (z : $float$) =
  $t_1 = x - y$
  $t_2 = t_1 * t_1$
  z = acc + t_2

dist(x : $vec\ float$, y : $vec\ float$) $\to$ (z : $float$) =
  t_3 = **reduce**(f_1, 0.0, x, y)
  z = sqrt(t_3)

Figure 8. Distance Function After Fusion Optimization

6. The Parakeet Runtime

Once a function has been type specialized and fully optimized, it is handed off to the Parakeet runtime for intelligent execution. The heart of the runtime is a heavy-weight interpreter whose primary responsibility is to initiate GPU kernel synthesis and execution. The interpreter uses program analyses and performance heuristics in order to dynamically make decisions such as:

1. what portion of a user's program ought to run on the GPU
2. which level of nested parallelism ought to expressed as a GPU kernel
3. in which GPU memory space should a particular array reside
4. when should data be moved onto or off the GPU

6.1 Cost Model

When the Parakeet interpreter encounters an array operator, it uses a simple cost model to decide what is the best place to execute the operator. The cost model employs a recursive function that estimates the relative cost of executing that operator on the CPU versus the GPU. This function is not meant to measure the precise

expected run time of the operator; rather, the goal is to make the correct decision when the use of one or the other processor should result in much higher performance. We use the clock frequency of each processor to roughly estimate the cost of a single operation. For each array operator, we multiply the estimated cost of a single execution of the operator's function argument by some fixed function of the input size. For a **map** operation, for example, we multiply the estimated cost of performing the sequentialized version of the mapped function by the number of input elements. In addition, we estimated the time needed to transfer data to and from the GPU as a function of data size and add this cost to the total if the data is not already present in the respective processor's memory.

In the case of nested array operators – e.g. a **map** whose payload function is itself a **reduce** the interpreter needs to choose which operator, if any, will form the parallelization point for a GPU kernel while sequentializing all nested operators within that kernel. The possible choices include:

1. Running the **map** as a GPU kernel, with an embedded sequential `for` loop that implements the **reduce**.
2. Running the **map** as a `for` loop in the Parakeet interpreter, with each iteration of the loop calling a GPU kernel that implements the **reduce**.
3. Running everything in the Parakeet interpreter as two nested `for` loops.

In cases (1) and (3), the operator is executed entirely on a single processor. In case (2), however, the **map** runs as a loop on the CPU, generating each element of the resulting vector with a separate GPU program invocation. In this case, the interpreter creates an interpreter array object on the CPU whose elements are references to the values generated on the GPU. The final linear CPU array is only constructed lazily as needed or when the GPU garbage collector needs the space. At that point, a linear CPU array is allocated and filled in with the computed values. This is precisely what happens in our implementation of K-Means clustering discussed in Section 8.2.

6.2 Shape Inference

Accurate prediction of array shapes is necessary both for the allocation of intermediate values on the GPU as well as for the above cost model determining placement of computations. We employ a simple abstract interpreter which propagates shape information through a function using the obvious shape semantics for each operator. For example, a **reduce** operation collapses the outermost dimension of its argument whereas a **map** preserves a shape's outermost dimension.

6.3 Data Movement and Garbage Collection

Arrays are moved to the GPU whenever they are used as the argument to some GPU computation. The specific GPU memory space into which an array is loaded depends on both the other data already on the GPU and the individual characteristics of the computation in which that array is being used. Unlike some systems similar to Parakeet [8], we do not by default preallocate the entirety of the GPU's memory. Rather, Parakeet has a runtime flag that can enable such preallocation for better performance in dedicated compute server environments. This is because we expect a typical use case to be in a desktop environment, where the GPU serves a double function of graphics processing and array operator acceleration and we don't want steal all of the GPU's resources.

Tracking and collection of unused memory is achieved by counting array references. Parakeet does not immediately free an array whose reference count drops to zero but rather maintains a cache of free data blocks which can be cheaply reused. Such caching of free pointers can be advantageous when a computation repeatedly allocate arrays of the same size.

6.4 Column-Major Transposition

Parakeet is able to leverage dynamic information in various ways to optimize both the implementation and the execution of the GPU programs it executes. One important such optimization is the use of data transposition. On GPUs, poor memory access patterns can result in orders of magnitude lower performance. In particular, neighboring threads in an executing GPU kernel should access adjacent memory words in order to get maximum performance. Use of column major data layouts as a performance optimization is well known in the CPU high performance computing world. Parakeet's data layout is row major by default. However, when a **map** computation is performed over a two-dimensional structure, a row major layout would result in the worst possible memory access pattern for the GPU kernel as its threads would concurrently access memory at some large stride. Thus in this case, Parakeet specializes the function in question to create a transposed column-major version of the data input. This column-major version is strictly an intermediate within the Parakeet runtime. We have found this optimization to be extremely beneficial to performance, and it contributes immensely to the good performance Parakeet delivers on the K-Means clustering benchmark discussed in Section 8.2.

6.5 First Order Array Operators

In addition to implementation skeletons for higher order array operators, we provide a library of precompiled parallel implementations of first-order array operators. Since these operators don't require synthesis with embedded payload functions, they don't need to be dynamically generated. For some operations (such as **sort**) we use the Thrust library [13], while providing our own hand-tuned implementations of others (such as **where**).

7. GPU Back End

Several systems similar to Parakeet [7, 8] generate GPU programs by emitting CUDA code which is then compiled by NVIDIA's CUDA nvcc toolchain. Parakeet on the other hand, targets PTX, which is NVIDIA's GPU pseudoassembly language. Parakeet's PTX code is dynamically compiled by the NVIDIA graphics driver before being executed. Parakeet caches compiled binaries so that when a particular array operator is run multiple times on the GPU with similar arguments it needn't incur the code generation and JIT compilation costs more than once.

We prefer compiling PTX instead of CUDA since this enables us to generate binaries more quickly, achieving a fully dynamic compiler without any noticeable stalls. The NVIDIA CUDA compiler is a wrapper around the GCC C++ compiler, and CUDA supports all of C++ in the host code and a large subset (including C++ templates) in the GPU code. Thus, the compile times for even simple kernels can be on the order of 5–10 seconds. There are, of course, advantages to using the NVIDIA compiler. The main ones are that we would be able to take advantage of all of the NVIDIA and GCC compiler optimizations and that it would simplify our implementation effort.

7.1 Imp

As mentioned above, we implement the higher-order array operators as skeletons of code with splice points where their payload functions get inlined. Rather than implement these skeletons directly in CUDA or PTX, Parakeet utilizes an embedded DSL called Imp, which operates at a semantic level similar to CUDA. Imp is intentionally a very simple language, since its purpose is to facilitate quick compilation from the typed IL to PTX.

Imp differs from CUDA in the following important respects:

1. Arrays are not associated with a particular GPU memory space (global, texture, constant, etc..), allowing us to compile variants of an Imp kernel where inputs reside in different memory spaces.
2. Space requirements of a function call (all outputs and local arrays it must allocate) can be determined as a function of input sizes. This is necessary as GPU computations have access only to memory which is allocated before their launch and cannot "dynamically" allocate more memory.
3. Local temporaries can be arrays in addition to scalars. This generalizes CUDA's use of "local" memory for spilled scalar variables.

Imp kernels are "shapely" by construction, meaning they specify their memory requirements as deterministic functions of input size. This obviates the need for ad-hoc allocation logic (the bulk of most CUDA host code in practice) or for auxiliary size inference on higher level code. If a function can be translated to Imp then we can always determine its memory requirements.

We perform staged synthesis of Imp kernels by parameterizing them with payload functions. An Imp kernel can be seen as a "skeleton" [11] for a particular implementation strategy of some array operator.

8. Evaluation

We evaluated Parakeet on two standard benchmark programs: Black-Scholes option pricing, and K-Means Clustering. We compare Parakeet against both hand-tuned CPU and GPU implementations. For Black-Scholes, the CPU implementation is taken from the PARSEC [2] benchmark suite – which we used as the basis of our Q implementation – and the GPU implementation is taken from the CUDA SDK [22]. For K-Means Clustering, we wrote our own Q version in 14 lines of code. Both the CPU and GPU benchmark version come from the Rodinia benchmark suite [10].

Our experimental setup is as follows. We ran all of our benchmarks on a machine running 64-bit Linux with an Intel Core i7 3.2GHz 960 4-core CPU, each core having 2 thread contexts for a total of 8, and with 16GB of RAM. The theoretical peak scalar throughput of this CPU is 25.6 GFLOP/s, and it has a theoretical peak memory bandwidth of 32GB/s [16]. The GPU used in our system was an NVIDIA Tesla C1060. This card has 240 processor cores with clock speeds of 1.296 GHz and 4GB of memory, with peak theoretical execution throughput of 933 GFLOP/s and peak memory bandwidth of 102 GB/s. We ran all of our experiments without the X windowing system running so that all of the GPU's resources were available to Parakeet – in our experience, X can consume upwards of 650MB of GPU memory in a standard desktop setup.

8.1 Black-Scholes

Black-Scholes option pricing [3] is a standard benchmark for data parallel workloads, since it is embarrassingly parallel – the calculation of the price of a given option doesn't impact that of any other, and the benchmark consists of simply running thousands of independent threads in parallel for computing the prices of thousands of options.

We compare our system against the multithreaded OpenMP CPU implementation from the PARSEC [2] benchmark suite with both 1 and 8 threads and the CUDA version in the NVIDIA CUDA SDK [22]. We modified the benchmarks to all accept the input data from the PARSEC implementation as their input so as to have a direct comparison of the computation alone. We also modified the CUDA SDK version to calculate only one of the call or put price per option, as that matches the behavior in PARSEC.

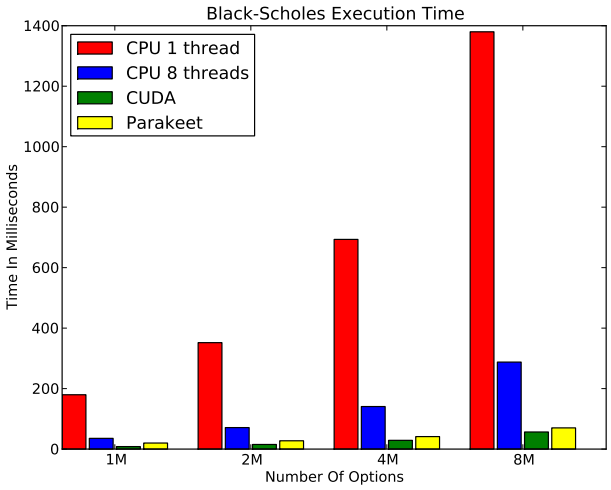

Figure 9. Black Scholes Total Times

In Figure 9, we see the total run times of the various systems. These times include the time it takes to transfer data to and from the GPU in the GPU benchmarks. As expected, Black Scholes performs very well on the GPU as compared with the CPU, since it is a purely data parallel benchmark. We see that Parakeet performs very similarly to the hand-written CUDA version, with overheads decreasing as a percentage of the run time as the data sizes grow. This is due to the fact that most of these overheads are fixed costs related to dynamic compilation.

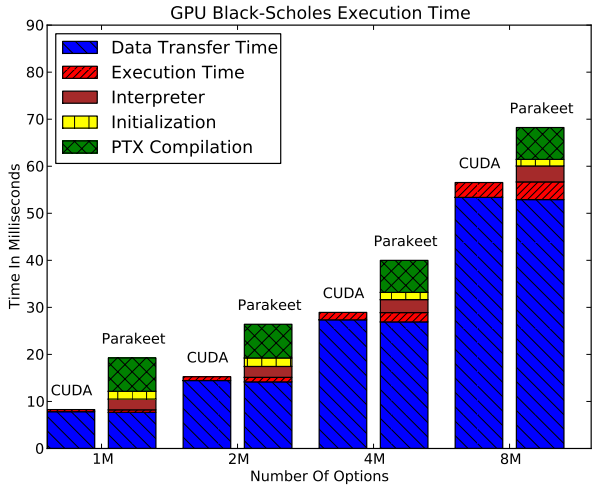

Figure 10. Black Scholes GPU Execution Times

In Figure 10, we break down Parakeet's performance as compared with the hand-written CUDA version. The Parakeet run times range from 24% to 2.4X slower than those of CUDA, with Parakeet performing better as the data size increases. These results illustrate an important aspect of GPU acceleration, viz. that the cost of transferring data to and from the GPU's memory is rather expensive. This large data transfer cost is, however, the same for Parakeet and

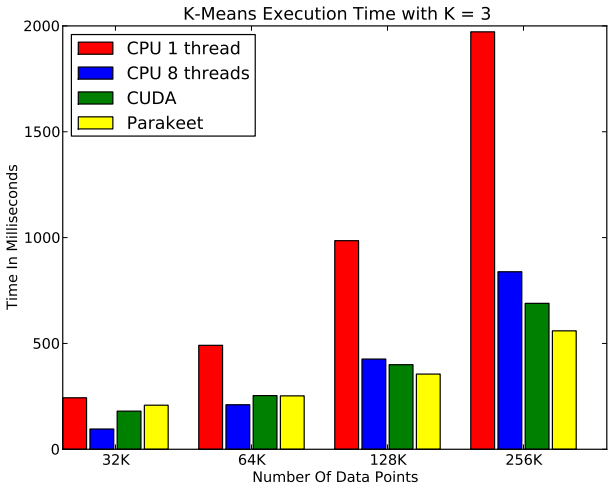

Figure 11. K-Means Total Times with 30 Features, K = 3

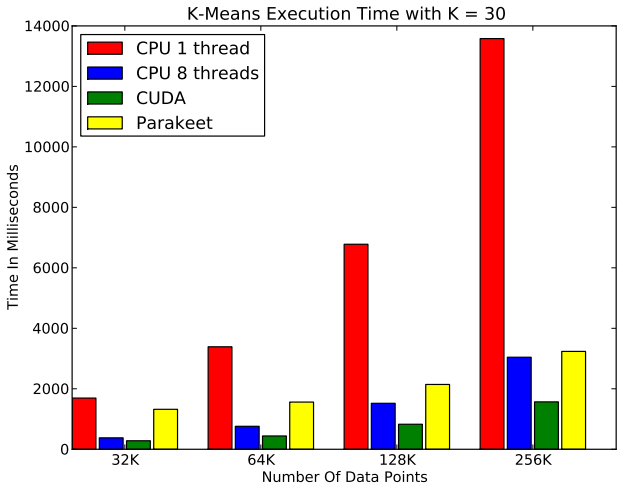

Figure 12. K-Means Total Times with 30 Features, K = 30

CUDA. The GPU programs that Parakeet generates are slightly less efficient than those of the CUDA version, with approximately 50% higher run time on average. However, the data transfer costs overwhelm the GPU execution times for Black Scholes, so this overhead isn't very important.

There are three other sources of overhead for Black Scholes that Parakeet introduces:

1. Initialization costs related to registering the Black Scholes function with the Parakeet runtime.
2. Time spent in the Parakeet IL interpreter.
3. The cost of running the JIT compiler on the generated PTX.

The PTX compilation time is rather significant for this benchmark – ranging from 33% of the total runtime for the 1 million option case to roughly 10% in the 8 million option case. This cost can't be helped, but it is important to note that it as well as the initialization costs are only paid once per function since Parakeet caches the compiled version of the function. Thus, in the case of workloads where a single function is repeatedly called, this costs will be insignificant.

8.2 K-Means Clustering

We also tested Parakeet on K-Means clustering, a standard unsupervised learning algorithm. K-Means is interesting as it is a fairly dense and complicated algorithm with loops and nested array operators. Thus it serves to illustrate that Parakeet is a usable system for implementing real world programs. K-Means has a number of parameters that can greatly affect run times and algorithmic characteristics. These include: the number of data points being clustered; the number of clusters K; and the number of features (i.e. the dimensionality of the data points).

In Figure 11, we see the total run times of K-Means for the CPU and GPU versions with K = 3 clusters and 30 features on varying numbers of data points. Here, the distinction between the GPU and the CPU is far less stark. In fact, for two smaller numbers of data points the 8-thread CPU version actually outperforms the GPU. More importantly for the present discussion, we see that on all but the smallest input size, Parakeet actually performs better than both the CPU and GPU versions.

The reason Parakeet is able to perform so well as compared with the CUDA version is due to the difference in how the two versions end up executing the code that computes the new average centroid for the newly computed clusters in each iteration. The CUDA version brings the GPU-computed assignment vectors back to the CPU in order to perform this reduction, as it involves many poor memory accesses and so has the potential to perform poorly on the GPU. Parakeet executes code to perform this function on the GPU instead, prefering to avoid the data transfer penalty. For such a small number of clusters, the Parakeet method ends up performing far better. For larger numbers of clusters, the fixed overhead of launching an individual kernel to average each cluster's points overwhelms the performance advantage the GPU gives and Parakeet ends up performing worse. While we don't take advantage of it yet, this illustrates the potential for us to use dynamic information to get the best of both worlds. In the future, in the case of a small number of clusters, Parakeet could execute the averaging code on the GPU, while for larger numbers, it could use a CPU implementation.

In Figure 12, we present the run times of K-Means for 30 features and K = 30 clusters. In this case, the hand-written CUDA version performs best in all cases, though its advantage over the CPU increases and its advantage over Parakeet decreases with increasing data size. As discussed, the Parakeet implementation suffers from the poorly performing averaging computation that it executes on the GPU, with a best case of an approximate 2X slowdown over the CUDA version. However, Parakeet exhibits the best scaling of any of the implementations, and would likely close the gap even further at larger data sizes.

In Figure 13, we see a breakdown of the run times for CUDA and Parakeet on the K = 3 clusters case. We can see that this benchmark is much more compute bound than Black Scholes, with the memory transfer costs contributing a much smaller percentage of the overall run time. Both Parakeet and the CUDA version use the CPU to perform a significant amount off the computation. The CUDA version uses the CPU to compute the averages of the new clusters in each iteration, while Parakeet spends a lot of time in the interpreter launching many GPU kernels and performing garbage collection of GPU memory. Even though the performance of Parakeet is fantastic, we imagine that with some more tuning we could push this gap even further.

K-Means clustering illustrates a key aspect of our approach: the ability to exploit dynamic information to tailor code generation and execution. When calculating centroids, we have a **map** operator

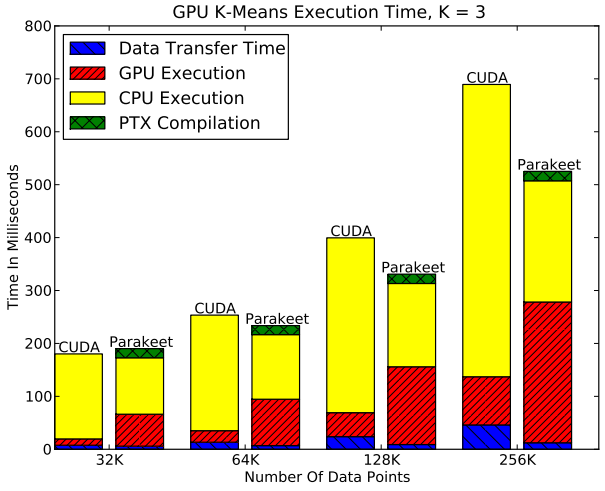

Figure 13. K-Means GPU Times with 30 Features, K = 3

using a nested function that in turns contains array operators. Our interpreter is able to make the most efficient choice regarding execution placement: the outer **map**, since it is applied to a small array of integers and involves little computation of its own, gets executed on the CPU as a loop. The inner function, which since it has dense computation applied to large inputs, gets synthesized into a kernel that is repeatedly launched. This type of dynamic decision based on data size would not be possible in a static environment. In addition, Parakeet uses transposition to a column-major data layout (see Section 6.4) in order to greatly improve the performance of this benchmark – transposition results in an average 12.5X speedup of K-Means clustering as a whole. The function which calculates centroid assignments is implemented as a map kernel where each thread is responsible for calculting the distance from its data point to every centroid. The Parakeet runtime identifies maps over 2D structures with nested maps over the rows, and transposes their inputs. This optimization is crucial for attaining coalesced memory accesses, and its absence can be ruinous for GPU performance. Since a transposed data matrix can be reused across loop iterations, the relative overhead of performing a transposition is very low.

9. Related Work

The use of graphics hardware for non-graphical computation has a long history [17], though convenient frameworks for general purpose GPU programming have only recently emerged. The first prominent GPU backend for a general purpose language was "Brook for GPUs" [6]. The Brook language extended C with "kernels" and "streams", exposing a programming model similar to what is now found in CUDA and OpenCL.

Microsoft's Accelerator [27] was the first project to use high level (collection-oriented) language constructs as a basis for GPU execution. Accelerator is a declarative GPU language embedded in C# which creates a directed acyclic graph of LINQ operations – such as filtering, grouping, and joining – and compiles them to (pre-CUDA) shader programs. Accelerator's programming model does not support function abstractions (only expression trees) and its only underlying parallelism construct is limited to the production of **map**-like kernels.

Three more recent projects all translate domain specific embedded array languages to CUDA backends:

- **Nikola** [18] is a first-order array-oriented language embedded within Haskell. Nikola provides a convenient syntax for expressing single-kernel computations, but requires the programmer to manually coordinate computations which require multiple kernel launches. Nikola also does not support partially applied functions and its parallelization scheme, which first serializes array operators into loops and then parallelizes loop iterations, seems ill-suited for complex array operations.
- **Accelerate** [9] is also an embedded array language within Haskell. Unlike Nikola, Accelerate does allow the expression of computations which span multiple kernel launches. Accelerate also has a much richer set of array operators (including the higher-order trio **map**, **reduce**, **scan**). Accelerate, however, does not seem to support closures or the nesting of array operators. Accelerate's backend is similar to ours in that they use a simple interpreter whose job is to initiate skeleton-based kernel compilation, transfer data to and from the GPU, and to perform simple CPU-side computations.
- **Copperhead** [7] parallelizes a statically typed purely functional array subset of Python through the dynamic compilation/execution of CUDA kernels. Copperhead supports nested array computations, and even has a sophisticated notion of where these computations can be scheduled. In addition to sequentializing nested array operators within CUDA kernels (as done in Parakeet), Copperhead can also share the work of a nested computation between all the threads in CUDA block. Unfortunately, Copperhead does not utilize any dynamic information (such as size) when making these scheduling decisions and thus must rely on user annotations. Copperhead's compiler generates kernels through parameterization of operator-specific C++ template classes. By using C++ as their backend target, Copperhead has been able to easily integrate the Thrust [13] GPGPU library and to offload the bulk of their code optimizations onto a C++ compiler. Runtime generation of templated C++ can, however, be a double-edged sword. Copperhead experiences the longest compile times of any project mentioned here (orders of magnitude longer than the compiler overhead of Parakeet).

Unlike the three approaches mentioned above, Parakeet does not require its source language to be purely functional nor statically typed. Being able to program in a dynamically typed language seems particularly important for array computations, since static type systems are generally unable to support the syntactic conveniences which make array programming appealing in the first place.

10. Conclusion

Parakeet allows the programmer to write in a high level sequential array language while taking advantage of the acceleration potential of using GPUs as coprocessors. Parakeet automatically synthesizes and executes very efficient GPU programs from the high level code while transparently moving data back and forth to the GPU's memory and performing GPU memory garbage collection. Parakeet includes a series of optimizations to generate more efficient GPU programs, including array operator fusion, data transposition, and the use of texture memory on the GPU. Parakeet is a usable system in which complex programs can be written and executed efficiently. On two benchmark programs, Parakeet delivers performance very competitive with even hand-tuned GPU implementations without impacting the interactivity of the source language interpreter environment.

In future work, we hope to support both more front ends and more back ends. At the moment, we are nearing completion of a front end for Python's NumPy, and we plan to build a front end for Matlab as well. Further, we envision building a back end for

multicore CPUs, likely targeting LLVM [15], which will allow us to simultaneously take advantage of the particular strengths of both types of processors.

In addition, we plan to increase efficiency by iteratively tuning components of hierarchical algorithms, splitting data inputs to take advantage of more of the texturing hardware, and doing more to overlap computation with data transfers.

References

[1] ALDINUCCI, M., GORLATCH, S., LENGAUER, C., AND PELAGATTI, S. Towards parallel programming by transformation: the FAN skeleton framework. *Parallel Algorithms Appl. 16*, 2-3 (2001), 87–121.

[2] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Processors, Architectures, and Compilation Techniques* (October 2008).

[3] BLACK, F., AND SCHOLES, M. The pricing of options and corporate liabilities. *The Journal of Political Economy 81*, 3 (1973), 637–654.

[4] BOLINGBROKE, M. C., AND JONES, S. L. P. Types are calling conventions. In *Proceedings of the 2009 Haskell Workshop* (September 2009).

[5] BORROR, J. A. *Q For Mortals: A Tutorial In Q Programming*. CreateSpace, 2008.

[6] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM, pp. 777–786.

[7] CATANZARO, B., GARLAND, M., AND KEUTZER, K. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (2011), pp. 47–56.

[8] CHAFI, H., SUJEETH, A. K., BROWN, K. J., LEE, H., ATREYA, A. R., AND OLUKOTUN, K. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2011), ACM, pp. 35–46.

[9] CHAKRAVARTY, M. M., KELLER, G., LEE, S., MCDONNEL, T. L., AND GROVER, V. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (2011), pp. 3–14.

[10] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)* (October 2009), pp. 44–54.

[11] COLE, M. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing 30*, 3 (2004), 389–406.

[12] FENG, W.-C., AND XIAO, S. To GPU Synchronize or Not GPU Synchronize? In *IEEE International Symposium on Circuits and Systems (ISCAS)* (Paris, France, May 2010).

[13] HOBEROCK, J., AND BELL, N. Thrust: A parallel template library, 2010. Version 1.3.0.

[14] JONES, S. P., TOLMACH, A., AND HOARE, T. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2004 Haskell Workshop* (2001).

[15] LATTNER, C. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[16] LEE, V. W., KIM, C., CHHUGANI, J., DEISHER, M., KIM, D., NGUYEN, A. D., SATISH, N., SMELYANSKIY, M., CHENNUPATY, S., HAMMARLUND, P., SINGHAL, R., AND DUBEY, P. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual International Symposium on Computer Architecture* (2010), pp. 451–460.

[17] LENGYEL, J., REICHER, M., DONALD, B. R., AND GREENBERG, D. P. Real-time robot motion planning using rasterizing computer graphics hardware. In *In Proc. SIGGRAPH* (1990), pp. 327–335.

[18] MAINLAND, G., AND MORRISETT, G. Nikola: Embedding compiled GPU functions in haskell. In *Proceedings of the 2010 Haskell Workshop* (September 2010).

[19] MOLER, C. B. MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science, 1980.

[20] MUNSHI, A. The OpenCL specification version 1.1, September 2010.

[21] NVIDIA. CUDA ZONE. `http://www.nvidia.com/cuda`.

[22] NVIDIA. NVIDIA CUDA SDK 3.2. `http://www.nvidia.com/cuda`.

[23] OLIPHANT, O. Python for scientific computing. *Computing in Science and Engineering 9* (2007), 10–20.

[24] OTTENSTEIN, K. J., BALLANCE, R. A., AND MACCABE, A. B. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not. 25* (June 1990), 257–271.

[25] SIPELSTEIN, J., AND BLELLOCH, G. E. Collection-Oriented languages. In *Proceedings of the IEEE* (1991), pp. 504–523.

[26] SVENSSON, J., SHEERAN, M., AND CLAESSEN, K. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Implementation and Application of Functional Languages, 20th International Symposium, IFL 2008* (2008).

[27] TARDITI, D., PURI, S., AND OGLESBY, J. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2006).

[28] TOLMACH, A., AND OLIVA, D. P. From ml to ada: Strongly-typed language interoperability via source translation. *J. Funct. Program. 8* (July 1998), 367–412.