# Parakeet: Automatic 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 staggering performance potential, which has led to an explosion in recent work on enabling the execution of general-purpose programs on GPUs. Despite various advances in making general-purpose GPU programming easier, the two commonly used frameworks (OpenCL and NVIDIA's CUDA) are cumbersome and require detailed architectural knowledge on the part of the programmer to fully harness this performance potential.

We aim to enable general purpose use of GPUs by observing that GPUs have been optimized for data parallel workloads. Array-oriented programming encourages the use of data parallel array operators (e.g. map, reduce, and scan) and discourages the use of explicit loops. Thus, array-oriented programming languages constitute a natural model for high level GPU programming.

We present Parakeet, an intelligent runtime and JIT compiler for high level array-oriented programs that transparently takes advantage of GPUs as accelerators. The heart of Parakeet is an interpreter, which upon reaching an array operator can automatically synthesize and execute a GPU program to implement that operator. Parakeet uses runtime information to: (1) decide where (CPU or GPU) to execute an array operator; (2) choose between different possible implementations of a GPU program; and (3) set execution parameters that greatly impact performance.

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 and the Rodinia GPU benchmark suite. Despite having orders of magnitude less code, the high level versions perform competitively when executed by Parakeet.

## 1. Introduction

Contemporary GPUs boast massive performance potential – in some cases orders of magnitude higher performance per dollar than that of CPUs – and most desktop systems today come with graphics processors. This has led to much interesting work on enabling the execution of general-purpose programs on GPUs (GPGPU) [7, 20, 22, 23, 28, 29]. Unfortunately, the two widely used GPGPU frameworks – NVIDIA's CUDA [23] and OpenCL [22] – require the programmer to have extensive knowledge of low level architectural details in order to fully harness the performance potential.



Our goal is to lower the barrier to GPGPU programming by allowing programmers to write in high level array languages that are transparently compiled into efficient GPU programs. By "array language" we mean any language equipped with:

1. First-class array values.
2. Succinct syntax for array creation/transformation.
3. Idiomatic preference for bulk array operations over explicit loops ("collection-oriented" [27] programming).

Array operations are higher level than explicit loops and thus tend to be easier for programmers to use. At the same time, array operations encode rich information about their access patterns that we can exploit to translate them into efficient parallel programs [17]. The prototypical array language is APL [15], which allows for extremely terse loop-free specification of algorithms. More commonly used array languages include Matlab [21] (the lingua franca of signal processing and machine learning research) as well as Python's NumPy [25] extensions.

In this paper, we present Parakeet, an intelligent runtime for executing high level array programs on GPUs. Parakeet dynamically compiles and specializes user functions via a typed intermediate language. The Parakeet interpreter then executes the program, using a simple cost model to decide whether to run operations on the CPU or GPU. For the GPU case, it transparently executes array operations by synthesizing and launching GPU kernels as needed.

A key target audience of Parakeet is scientific programmers. It is common for them to run computational experiments that are extrememely time consuming and operate on huge data inputs – codes which are natural candidates for GPU acceleration. However, it is highly preferable for them not to have to learn specialized tools like CUDA in order to take advantage of new hardware. In addition, dynamic and interactive languages such as Matlab are very popular as they allow for rapid prototyping of algorithms. Our main goal is to enable the use of such languages not only for prototyping, but production execution as well.

We have implemented our first Parakeet front end for Q [5], a descendant of APL that is widely used in financial computing. Q is a nearly "pure" array language – its idiomatic style makes sparser use of loops than even Matlab or NumPy – and is thus a natural first choice. Parakeet is designed to be front end language agnostic, and we are nearly finished with a front end for Python's NumPy.

The main contributions of this paper are the following:

- A system in which programmers can write complex, high level code that is automatically parallelized into efficient GPU programs.
- A dynamic specialization algorithm that translates a significant subset of higher-order programs into efficient (unboxed, function-free) GPU code.

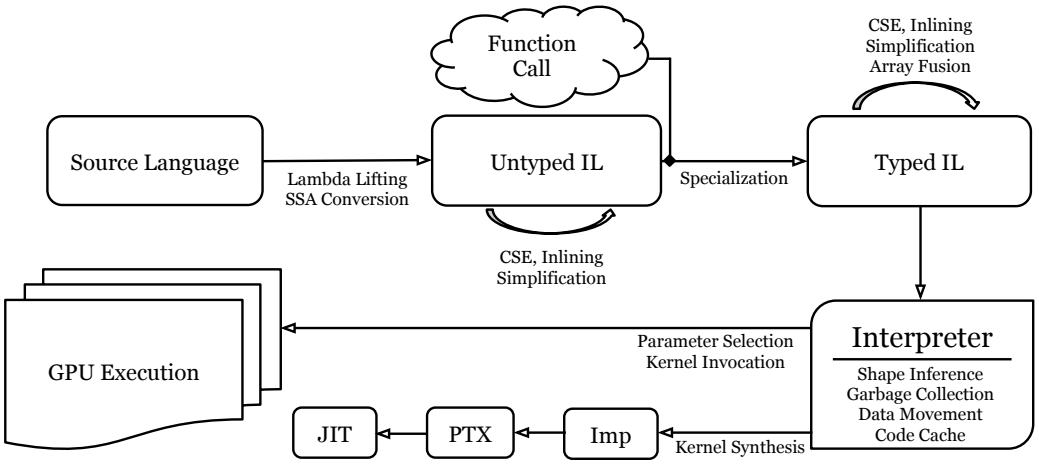


**Figure 1.** Overview of Parakeet

## 2. Overview

As mentioned, a key target audience for Parakeet is scientific programmers. In general, the target audience includes the large majority of programmers who aren't specialists in tuning low level code to get peak performance. The goal for Parakeet is to enable these programmers nonetheless to take advantage of the acceleration potential of GPUs by automating some the steps of the process that require highly specialized knowledge.

At present, a typical usage pattern in scientific computing is to prototype an algorithm in a high level interactive language such as Matlab due to the ease of rapid exploration of the problem domain in such an environment. Then, since run times for these programs are often on the order of hours or days, it is often necessary for performance performance reasons to port them to a high performance language such as C or CUDA. Beyond the need for specialized high performance programming skills, a problem with this procedure lies in the disconnect between the environment for domain exploration and the environment in which real results can be derived as it greatly hinders productivity.

Our thesis is that if we can JIT compile and transparently accelerate the critical paths of algorithms written in array-oriented languages, we can bring the best of both worlds together – viz., the ease of exploration of the dynamic, interactive environment and the speed of the hand optimized versions. A key constraint in such a setup is keeping JIT compilation overheads low enough so as not to impair the interactivity of the source language environment. It has been shown in the pioneering JIT work by Chambers [10] that in such scenarios the key psychological barrier for compilation overhead is roughly on the order of less than half a second: if overheads are kept below this threshold, the programmer doesn't notice them. Since our JIT costs are negligible in comparison with the run times of the large scientific codes, we feel it is reasonable to expect programmers to pay them in return for the resultant large runtime speedups so long as the JIT costs don't slow down their workflow.

### 2.1 Execution Pipeline

The program execution pipeline for our system (shown in Figure 1) begins in the standard interpreter of the source array language. Parakeet is designed as an accelerator library for dynamic, interpreted array languages, and is attached to the source language's interpreter via the interpreter's plugin interface. This allows the user to continue to use all of the language's normal tools and support libraries.

The core of the Parakeet framework is its internal typed intermediate language and an interpreter provided for that language. This IL includes data parallel array operators such as **map**, **reduce**, and **scan** that translate well into efficient data parallel GPU programs (a full table of the supported operators can be found in Figure 4).

Execution of a program proceeds as normal in the source language's interpreter, with Parakeet intercepting function calls. The source of these functions is then translated into an untyped intermediate representation using the Parakeet front end interface and the source language's facilities for introspection. At this step, various standard compiler optimizations are applied to the untyped IL representation such as common subexpression elimination.

When a call is made to such an intercepted function, the untyped function body is type specialized according to its argument types. Further standard optimizations are performed at this stage, as well as an optimization we call *array operator fusion*. Here, nested array operators are fused according to various rewriting rules into fewer operators. This fusion step can be extremely beneficial to the final GPU program's performance, as it eliminates temporaries and therefore achieves more efficient use of GPU memory bandwidth.

Next, the Parakeet interpreter interprets the typed IL version of the function. When Parakeet reaches an array operator node in the code tree, it employs a simple cost-based heuristic (which includes things such as data size and memory transfer costs) to decide whether to execute that array operator on the GPU or CPU (for more details see Section 7.1). Some code simply cannot be efficiently translated into GPU programs (such as that which performs I/O or modifies global state), and such code is kept on the CPU. In the case where Parakeet executes the operator on the CPU, the operator is either (1) translated into an IL implementation and then executed by the Parakeet interpreter; or (2) passed back to the source language's interpreter for execution. In this way, we are able to support the entire source language by simply deferring to its interpreter all computation we don't wish to perform or don't yet support. The Parakeet interpreter's execution of IL code hasn't been heavily optimized as of yet – our effort has been focused on the generation of efficient GPU kernels. This is reasonable for now, as its IL is very dense since array operators express large computations extremely concisely.

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 an efficient GPU skeleton that implements the pattern of applying the same function to each element of an array. The flattened payload function argument to the **map** 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. If a kernel's data doesn't fit on the GPU, we default to CPU execution (for now). We are currently implementing an approach to divide an array operator's execution into pieces to allow execution of kernels on arbitrarily large inputs.

## 3. GPU Hardware

To set the stage and motivate the design choices of Parakeet, we first discuss some important features of modern GPU hardware. A GPU consists of an array of tens of multiprocessors. Within these multiprocessors are various resources such as local memories and instruction issue units that are shared among its simple cores. Since the issue units are shared, the threads running on a single multiprocessor execute instructions in lockstep – hence the name Single Instruction Multiple Thread (SIMT) for the execution model. The typical pattern is to issue a short program to be executed in parallel by thousands of lightweight threads that run on these hundreds of simple cores, each operating on different subsets of some input data. With all these cores, a typical graphics card has a peak throughput of many hundreds of GFLOP/s – an order of magnitude more than typical high end CPUs at a fraction of the cost.

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 colocated 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 irrelevent 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 quite cumbersome, but is also essential to good performance for many 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 accelerators. However, 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 [13], 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 bracket 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

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

**Figure 2.** K-Means Clustering implemented in Q

the centroid to calculate, respectively), and calculates that cluster's centroid.

This function illustrates some of the array-oriented features of Q. First, Q allows implicit mappings of functions elementwise across vectors – for example, to test elementwise 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 elementwise 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, the `each` keyword modifies a function by applying it elementwise to its argument. In the `calc_centroids` function, we calculate each cluster's new centroid by using `each` to apply the `calc_centroid` function to a list of integers from `0` to `k-1`. The other adverbs used in K-Means are the directional each variants, `each-left` and `each-right`. These adverbs modify a binary function and apply it to each element of the respective input argument. By combining both, we are able to apply the `dist` function to each pair of rows from the two input arrays–here to each data point and centroid to calculate all the needed distances for reassigning the centroids.

We see that array programming languages are 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 17 even though we wrote in a verbose style in the interest of clarity. Importantly, the Q program is sequential as well.

## 5. Typed Intermediate Language

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

In deference to the above-mentioned GPU hardware restrictions we disallow from our intermediate language:

- Polymorphism of all kinds
- Recursion
- User-specified higher-order functions
- Compound data types other than arrays

These restrictions are not necessarily as severe as they initially seem, since the programmer needn't know about them. This is simply the internal format Parakeet uses in order to generate efficient GPU back end code. We explain the specialization algorithm later on in this paper.

Our intermediate language is shown in Figure 3. The parallelism abstraction we prefer to maintain is the use of the higher-order array operators **map**, **reduce**, and **scan**. The higher-order array operators 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 array operators such as **sort** (a full list can be found in Figure 4) are executed in parallel on the GPU as well. They simply aren't higher-order, and thus are implemented via a fixed parallel standard library.

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 $\tau, \overline{\tau_n} = \overline{\tau_{m+1}}$.

Scalar operators include the usual boolean operations as well as scalar and floating point arithmetic operators and functions. Examples of simple array operators include indexing, replication, and inspecting an array's shape.

Salient features of our intermediate language include:

- 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 from Parakeet.
- It is important to note that we can still represent function values within our intermediate language (which are created by partial application of top-level functions). However, since user functions cannot themselves accept other functions as arguments, the only purpose of function values is to parameterize the **map**, **scan**, and **reduce** higher-order primitives.
- It is also worth highlighting the curious nature of the type we assign to function values: $\{f_i, \overline{\tau_c}\} \Rightarrow \overline{\tau_m} \rightarrow \overline{\tau_n}$. The terms occurring before the double arrow ($\{f_i, \overline{\tau_c}\}$) represent both the function label of a closure and the types of partially applied function arguments. We attach this information to a closure's

**Intermediate Language Syntax**

| | | | |
|---|---|---|---|
| program | $p$ | ::= | $d_1 \cdots d_n$ |
| definition | $d$ | ::= | $f_i(\overline{x_m} : \overline{\tau_m}) \rightarrow (\overline{y_n} : \overline{\tau_n}) = s^+$ |
| statement | $s$ | ::= | $\overline{x_m} : \overline{\rho_m} = e$ |
| | | \| | **if** $v$ **then** $s^*$ **else** $s^*$ |
| | | \| | **while** $e$ **do** $s^+$ |
| expression | $e$ | ::= | $v_f(v_1, \ldots, v_m)$ |
| | | \| | values$(\overline{v_m})$ |
| | | \| | cast $(v, \tau)$ |
| | | \| | **map**$_m(v_f, \overline{v_m})$ |
| | | \| | **reduce**$_m(v_f, v_g, v_{init}, \overline{v_m})$ |
| | | \| | **scan**$_m(v_f, v_g, v_{init}, \overline{v_m})$ |
| value | $v$ | ::= | numeric constant |
| | | \| | $x$ (data variable) |
| | | \| | $f$ (function label) |
| | | \| | $\oplus$ (scalar operator) |
| | | \| | $a$ (simple array operator) |

**Type System**

| | | | |
|---|---|---|---|
| data | $\tau$ | ::= | $int \mid float \mid \textbf{vec}\ \tau$ |
| closures | $\rho$ | ::= | $\{f_i, \overline{\tau_c}\} \Rightarrow \overline{\tau_m} \rightarrow \overline{\tau_n}$ |
| | | \| | $\tau$ |
| higher-order | $\theta$ | ::= | $\overline{\rho_m} \rightarrow \overline{\rho_n}$ |

**Figure 3.** Intermediate Language and Type System

type to facilitate efficient passing of closure arguments to the GPU and avoid any actual indirection which might otherwise be associated with function invocation. The need to statically determine a function label for any closure restricts which programs Parakeet can compile. We must reject any code where the function that executes at a particular program point depends on some dynamic condition.

## 6. Translation, Specialization, and Optimization

In this section we describe the various program transformations we perform before executing a user's function. Some of these transformations (such as lambda lifting and specialization) are necessary in order to 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 targetting 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 5.

### 6.1 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 [26]. Classical SSA cannot be directly executed since the $\phi$-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. Figure 6 shows the `dist` function after it has been translated to Untyped SSA.

| **Higher-Order Array Operators** | |
|---|---|
| **Array Operator** | **Meaning** |
| **map**($f$,$x$) | Apply function $f$ to all elements of $x$ |
| **reduce**($f$,$init$,$x$) | Return result of inserting binary $f$ between all elements of $x$ prefixed by $init$ |
| **scan**($f$,$init$,$x$) | Return running application of $f$ to $init$ and each element of $x$ |

| **Simple Array Operators** | |
|---|---|
| **Array Operator** | **Meaning** |
| **sort**($x$) | Return a sorted version of $x$ |
| **find**($x$,$i$) | Return index of first occurence of $i$ in $x$ |
| **where**($x$) | Return an array of indices where $x$ = 1 |
| **index**($x$,$i$) | Return the value of $x$ at index $i$ |

**Figure 4.** Parakeet's Supported Array Operators

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

**Figure 5.** Distance Function in Q

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

**Figure 6.** Untyped Distance Function in SSA form

### 6.2 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

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.

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

**Figure 7.** Distance Function after untyped optimizations

### 6.3 Specialization

The purpose of specialization 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 all 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.

The signature of data is one of our built-in dynamic value types such as $float$ or $vec\,vec\,int$. The signature of functions is a closure that includes a function tag and a list of data signatures. It is important to note that specialization is thus not just on types, but also on function tags and the types of their associated closure arguments. This is equivalent to performing defunctionalization (Reynolds) and then specializing exhaustively on the constant values of closure records. One caveat here is that non-constant closure values are disallowed. This prevents us from having to implement them as large switch statements on the GPU, which would be very inefficient due to branch divergence. Finally, only data is allowed to cross the boundary from our system to the source language – specialized Parakeet functions remain enclosed in our runtime.

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 8.

$$
\begin{array}{l}
\text{dist ( x : } vec\ float\text{, y : } vec\ float\text{)} \rightarrow \text{( z : } float\text{)} = \\
\quad \mathrm{t}_1 = \textbf{map}(-_{\mathrm{float}}, \mathrm{x}, \mathrm{y}) \\
\quad \mathrm{t}_2 = \textbf{map}(*_{\mathrm{float}}, \mathrm{x}, \mathrm{y}) \\
\quad \mathrm{t}_3 = \textbf{reduce}(+_{\mathrm{float}}, 0, \mathrm{t}_2) \\
\quad \mathrm{z} = \ \mathrm{sqrt}(\mathrm{t}_3)
\end{array}
$$

**Figure 8.** 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.

### 6.4 Array Operator Fusion

In addition to standard compiler optimizations (such as constant folding, function inlining, and common sub-expression elimination), we employ fusion rules [16] 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

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

Reduce-Map Fusion

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

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 9, 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.

$$
\begin{array}{l}
\mathrm{f}_1\text{( acc : } float\text{, x : } float\text{, y : } float\text{)} \rightarrow \text{( z : } float\text{)} = \\
\quad \mathrm{t}_1 = x - y \\
\quad \mathrm{t}_2 = t_1 * t_1 \\
\quad \mathrm{z} = \mathrm{acc} + \mathrm{t}_2 \\
\\
\mathrm{dist}\text{( x : } vec\ float\text{, y : } vec\ float\text{)} \rightarrow \text{( z : } float\text{)} = \\
\quad \mathrm{t}_3 = \textbf{reduce}(\mathrm{f}_1, 0.0, \mathrm{x}, \mathrm{y}) \\
\quad \mathrm{z} = \mathrm{sqrt}(\mathrm{t}_3)
\end{array}
$$

**Figure 9.** Distance Function After Fusion Optimization

## 7. The Parakeet Runtime and Implementation

In this section, we discuss the internals of the Parakeet runtime in more detail. We implemented the Parakeet runtime and intepreter in OCaml due to its amenability to compiler writing. We spent considerable effort tuning the OCaml code to be efficient and to prevent OCaml's garbage collector from becoming a source of large overheads. The intepreter uses an interface written in C to allow it to interact with CUDA and with NVIDIA's GPU JIT compiler. Parakeet also exposes a front end interface written in C for translating functions from the source language into Parakeet ASTs and to pass data between the source language's interpreter and the Parakeet runtime.

When a function call is made to a function registered with Parakeet, the Parakeet interpreter intercepts the call and type specializes the function for the argument types passed in. The interpreter then walks this typed IL version of the function. When it encounters a

simple scalar math operation, the standard behavior is simply to execute that operation itself. Any I/O or other operations that aren't supported by Parakeet's IL are simply executed by the source language's interpreter.

### 7.1 Cost Model

When the Parakeet interpreter encounters an array operator, it uses a simple cost model to decide on 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 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. 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 operators – e.g. a **map** whose payload function is itself a **reduce** (as is present in the `calc_centroids` function in the K-Means benchmark) – the interpreter needs to choose which operator, if any, will form the parallelization point for a GPU kernel while sequentializing all nested operators within the kernel. The choices here are:

1. Running the **map** as a GPU kernel, with an embedded sequential `for` loop for 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 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 result 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 lineary CPU array is allocated and filled in with the computed values. This is precisely what happens in our implementation of K-Means clustering.

### 7.2 GPU Back End

Our GPU back end only supports NVIDIA GPUs at the moment. NVIDIA GPUs are typically programmed using CUDA, a C API for writing and executing GPU programs. In CUDA, a program is organized into *kernels* and *host* code. Kernels are typically executed by many thousands of lightweight threads on the GPU. Host code is the name for the code run on the CPU. Typically, a host thread acts as a master that asynchronously launches worker kernels onto the GPU via function calls to the CUDA API. The host thread in our case is simply the Parakeet interpreter itself.

Unlike many other systems similar to Parakeet, our back end doesn't emit CUDA code to be compiled by the NVIDIA CUDA compiler. Instead, we emit PTX, NVIDIA's GPU pseudoassembly language. (As we will see in the following section, we also have one additional level of intermediate language in Parakeet between the typed IL discussed in Section 6 and our emission of PTX code.) This PTX code is JIT compiled by the NVIDIA graphics driver before being launched. We keep these compiled binary versions of functions in a code cache so that when they are invoked multiple times with the same argument types we needn't incur the code generation and JIT compilation costs more than once.

The reason that we emit PTX instead of CUDA code is largely our latency requirements, as our goal is to provide a framework for transparent acceleration of interactive languages. 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 a 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. However, the strict latency requirements meant that we needed to emit PTX directly.

### 7.3 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 includes a second intermediate language that we call Imp (short for imperative) which we use for this purpose. This means in addition that in order to inline a function into a higher-order Imp skeleton we must first translate the function's body from the higher level Parakeet IL into Imp.

Imp is largely a syntactic sugar wrapper around PTX that simplifies our job of implementing efficient GPU versions of these operator skeletons by hiding some of the architectural details PTX exposes. However, Imp does differ from PTX in some 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" [12] for a particular implementation strategy of some array operator. An example of a simplified Imp skeleton for **map** is shown in Figure 10. The lines with the `SPLICE` keyword are the points at which the payload function gets added to the skeleton to form a complete kernel.

**Figure 10.** Imp Map Skeleton

### 7.4 Shape Inference

### 7.5 Parallel Library

In addition to these skeletons, we provide a library of precompiled parallel implementations of our simple (first-order) array operators,

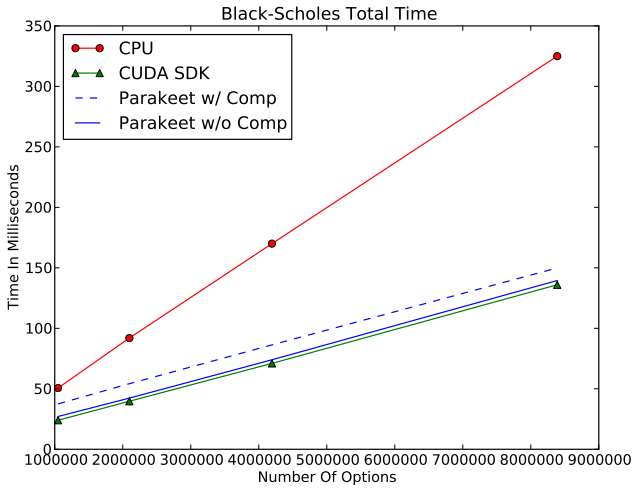


**Figure 11.** Black Scholes Total Times

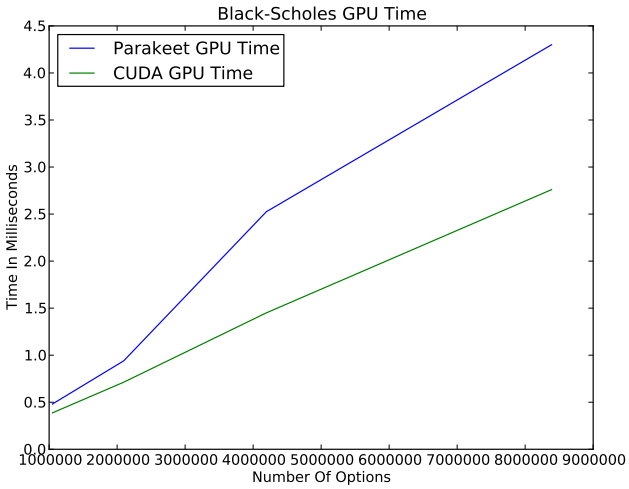


**Figure 12.** Black Scholes GPU Execution Times

which are listed in Figure 4. Since these operators don't require synthesis with embedded payload functions, we needn't dynamically generate them. We use the Thrust library [14] for some of these operators such as **sort**, while providing our own hand-tuned implementations of others such as **where**.

## 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 [24]. For K-Means Clustering, we wrote our own Q version in 17 lines of code. Both the CPU and GPU benchmark version come from the Rodinia benchmark suite [11].

Our experimental setup is as follows. We ran the CPU benchmarks on a machine with an Intel Nehalem 2.67GHz X5550 4-core CPU with 24GB of RAM. The theoretical peak throughput of this machine is 85.12 GFLOP/s, and it has a theoretical peak memory bandwidth of 32GB/s.

For all GPU benchmarks, we ran the programs on an NVIDIA GTX260. This card has 216 processor cores with clock speeds of 1.3 GHz and 896MB of memory, with peak execution throughput of 805 GFLOP/s and peak memory bandwidth of 112 GB/s.

For all of our Parakeet benchmarks, the desktop system we used for running the interpreter had an Intel Core 2 6600 2.4GHz processor with 2 cores and 4GB of RAM.

### 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 4 threads and the CUDA version in the NVIDIA CUDA SDK [24].

In Figure 12, 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.

### 8.2 K-Means Clustering

K-Means illustrates a key aspect of our approach: the ability to exploit dynamic information to tailor code generation and execution. In the `calc_centroids` function, we have a **map** operator applied to a nested function that in turns contains array operators. Our interpreter is able to make the most efficient choice regarding where to execute things: the outer **map**, since it's 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.

## 9. Related Work

The use of graphics hardware for non-graphical computation has a long history [19], 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 [29] 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.

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

- **Nikola** [20] 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 ap-

plied 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.
- **OptiML and Delite** [8] are a domain-specific language and runtime framework designed to
- **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 [14] 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. Parakeet automatically synthesizes GPU programs from the resulting program and executes by 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 and the use of shared and texture memory on the GPU. Parakeet is system in which complex programs can be written and executed efficiently. On two benchmark programs, Parakeet delivers performance competitive with even hand-tuned CPU and GPU implementations.

In future work, we hope to support more front ends and back ends. At the moment, we are building front ends for both Matlab and Python. We envision building a back end for multicore CPUs as well, likely targeting LLVM [18].

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 overlapping computation with data transfer.

## 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. Tech. Rep. UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010.

[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.

[10] CHAMBERS, C. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford, CA, USA, 1992.

[11] 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.

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

[13] 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).

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

[15] IVERSON, K. E. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference* (New York, NY, USA, 1962), AIEE-IRE '62 (Spring), ACM, pp. 345–351.

[16] 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).

[17] JU, D. C. R., WU, C. L., AND CARINI, P. The classification, fusion, and parallelization of array language primitives. *IEEE Trans. Parallel Distrib. Syst. 5* (October 1994), 1113–1120.

[18] 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`.

[19] 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.

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

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

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

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

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

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

[26] 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.

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

[28] 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).

[29] 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).