Parakeet: Fast GPU Programming with Array Languages

Alex Rubinsteyn, Eric Mann-Hielscher, and Dennis Shasha

New York University, New York, 10003, USA
{alexr,hielscher,shasha}@nyu.edu

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 think there is a better approach. 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 programming of GPUs.
We present Parakeet, an intelligent runtime for executing high level array-oriented programs on GPUs. The heart of Parakeet is an interpreter, which upon reaching an array operator synthesizes and executes a GPU program to implement that operator. Parakeet takes advantage of runtime information both to choose between different possible implementations of a GPU program and to set execution parameters which greatly impact performance. Data is moved transparently to and from the GPU, and garbage collected when no longer needed.
We evaluate our system 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 excellent recent work on enabling the execution of general-purpose programs on GPUs (GPGPU) [7, 19, 21, 22, 27, 28]. Unfortunately, the two widely used GPGPU frameworks–NVIDIA's CUDA [22] and OpenCL [21]–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 instead of explicit loops ("collection-oriented" [26] 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 [16]. The prototypical array language is APL [13], which allows for extremely terse loop-free specification of algorithms. More commonly used array languages include Matlab [20] (the lingua franca of signal processing and machine learning research) as well as Python's NumPy [24] 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 transparently executes array operations on the GPU by synthesizing and launching GPU kernels as needed.

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. We are also currently working on front ends for both Matlab and Python.

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.
- A low-level intermediate language called *Imp* that captures the notion of shapely computation [14] necessary in the absence of dynamic memory allocation.

2 Overview

The pipeline of the execution of a program (shown in Figure 1) begins in the standard interpreter of the source array language. The Parakeet framework is attached to the source language's interpreter as a module, allowing the user to exploit all of the language's normal tools and support libraries. The core of the Parakeet framework is its internal typed intermediate langauge and an interpreter provided for that language. This IL includes data parallel array operators

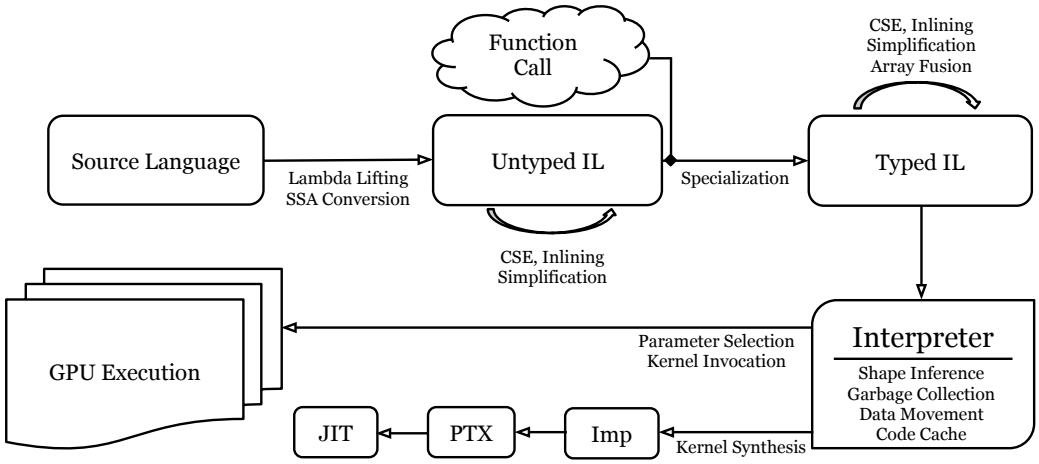

Fig. 1. Overview of Parakeet

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 in the source language's interpreter, with Parakeet intercepting function calls. The source of these functions is then translated into an untyped intermediate representation. At this step, various standard compiler optimizations are applied such as common subexpression elimination.

The function body is then 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, because it eliminates temporaries and therefore achieves more efficient use of GPU memory bandwidth.

Next, the Parakeet interpreter interprets the typed IL 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. 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 translated into an IL implementation and then further interpreted by Parakeet, or executed by the source language's interpreter itself. The Parakeet interpreter itself hasn't been heavily optimized–our effort has been focused on enabling it to generate efficient GPU kernels. This is reasonable for now, as its IL is very dense because array operators express large computations very concisely.

If an array operator's computation is deemed a good candidate for GPU execution, Parakeet flattens all computations nested 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**, Parakeet provides a GPU skeleton that implements the pattern of applying the same function to each element of an array. The flattened payload computation of the **map** is inlined into this skeleton, and a complete GPU program is synthesized.

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. 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 GLOP/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 prefferred 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 graphics accelerators. However, this optimization 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 [11], 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 our design of efficient high level abstractions 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 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. Since our focus is the Parakeet runtime, we omit many details of the Q language and present only the salient features in order to illustrate the level of programming and Q's support for array operators.

4.1 K-Means Clustering Example

```
calc_centroid:{[X;a;i] avg X[where a = i]}
calc_centroids:{[X;a;k]
  calc_centroid[X;a] each til k}

dist:{[x;y] sqrt sum (x-y) * (x-y)}
minidx:{[x] x ? min x}

kmeans:{[X;k;a]
  C: calc_centroids[X;a;k];
  converged: 0b;
  while[not converged;
    lastAssignment: a;
    D: X dist/:\: C;
    a: minidx each D;
    C: calc_centroids[X;a;k];
    converged: all lastAssignment = a];
  C}
```

Fig. 2. K-Means Clustering implemented in Q

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 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 performs the elementwise equality between 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 adverb used in K-Means is what we call *all-pairs*, written '`/:\:`'. (Technically all-pairs is a combination of two adverbs in Q: each-left and each-right.) All-pairs modifies a binary function and applies it to each pairs of elements from two input arrays. We use all-pairs to apply the `dist` function to each pair of 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 and could be made more compact. And, of course, the Q program is sequential.

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 is 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 illustrates 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.

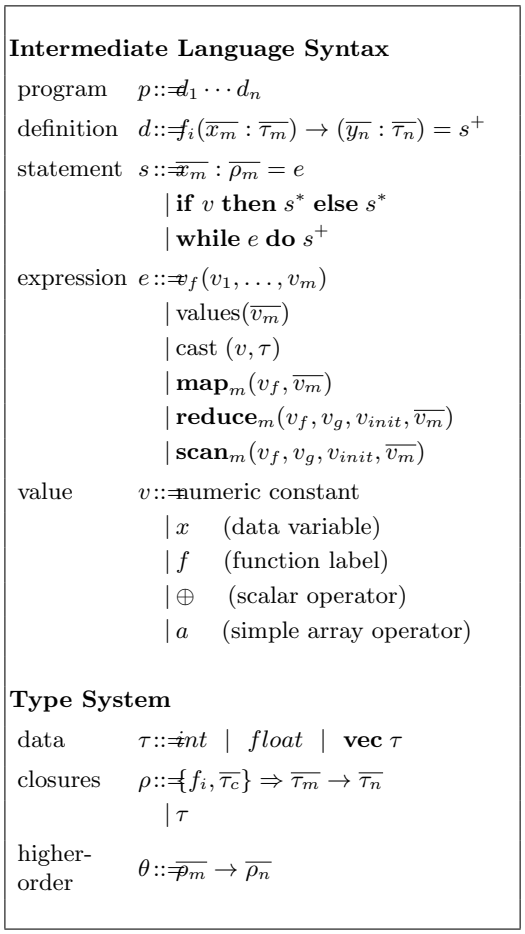

Fig. 3. Intermediate Language and Type System

– 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 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.

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

Fig. 4. Parakeet's Supported Array Operators

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.

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

Fig. 5. Distance Function in Q

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 [25]. 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. Figure 6 shows the `dist` function after it has been translated to Untyped SSA.

```
dist ( x, y ) → ( z ) =
  t1 = x − y
  t2 = x − y
  t3 = t1 ∗ t2
  t4 =sum(t2)
  z = sqrt(t4)
```

Fig. 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.

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

Fig. 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.

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

Fig. 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 [15] 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

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

Reduce-Map Fusion

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

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.

$$
\begin{array}{l}
\mathrm{f_1}(\ \mathrm{acc} : float,\ \mathrm{x} : float,\ \mathrm{y} : float) \rightarrow (\ \mathrm{z} : float) = \\
\quad \mathrm{t_1} = x - y \\
\quad \mathrm{t_2} = t_1 * t_1 \\
\quad \mathrm{z} = \mathrm{acc} + \mathrm{t_2} \\
\\
\mathrm{dist}(\ \mathrm{x} : vec\ float,\ \mathrm{y} : vec\ float) \rightarrow (\ \mathrm{z} : float) = \\
\quad \mathrm{t_3} = \mathbf{reduce}(\mathrm{f_1}, 0.0, \mathrm{x}, \mathrm{y}) \\
\quad \mathrm{z} = \mathrm{sqrt}(\mathrm{t_3})
\end{array}
$$

Fig. 9. Distance Function After Fusion Optimization

7 GPU Back End

Before we discuss our GPU code generation, we first walk through the CUDA programming model in some detail. Our GPU back end targets NVIDIA GPUs by emitting PTX–NVIDIA's GPU pseudoassembly language. NVIDIA GPUs are typically programmed using CUDA, a higher level C-like wrapper around PTX. As we will see in section 7.2, we have one additional level of intermediate language in Parakeet before we emit PTX code. This PTX is then finally JIT compiled by the NVIDIA driver before being launched.

7.1 CUDA Programming Model

In CUDA, a program is organized into *kernels* and *host* code. Kernels typically run by many thousands of lightweight threads on the GPU. Host code is the code run on the CPU. Typically, a host thread on the CPU launches a kernel onto the GPU via a C function call to the CUDA API. The host thread waits for results and potentially launches futher kernels.

CUDA Kernel Organization As is typical for data parallel models, all threads in a CUDA kernel execute the same program code. This code is specified in a C function using the CUDA C API. The kernel's threads are organized into *thread blocks*, which are groups of at most a few hundred threads that can communicate and share local resources. Threads within a given block are all scheduled to

run on a single GPU multiprocessor throughput their lifetime. Threads within a block can perform barrier synchronizations with each other, but threads from different blocks have no efficient way to perform synchronization other than termination. Thus algorithms that require global synchronization (such as parallel reduction) must be broken up into multiple kernel launches, each of which incurs a small fixed performance overhead.

The body of a kernel can be written so as to be generic to the number of threads and blocks with which it is launched, with the programmer specifying at runtime how many thread blocks are required and how many threads to launch per block. This allows the kernel programs to be scalable to different data sizes without requiring rewrites. The standard idiom is for each thread to be assigned a small computation to be performed on a small subset of some large input data set. The threads are each assigned a unique index in the kernel's *grid* of threads, which is used to determine the data for which they are responsible.

In order to be efficient, CUDA kernels must be dense. The kernel is executed by thousands of threads, and any small inefficiencies are greatly magnified. Manual loop unrolling and rewriting algorithms to have more work per thread are usually necessary to obtain performance near the GPU's peak potential.

CUDA Memory Management The graphics card has its own memory and GPU kernels cannot access data stored in the CPU's RAM. In typical use, data must be manually copied back and forth between the CPU and GPU memory spaces by the CUDA programmer. This data transfer is roughly two orders of magnitude slower than memory accesses performed by kernels to the GPU's memory, and so it is critical for performance that these data transfers are minimized. It is also possible and beneficial to overlap memory transfers with computation.

The main memory space on GPUs is called *global memory* in CUDA. GPUs have very high peak memory bandwidth between their processors and global memory–usually over 100GB/s. This is made possible because the hardware groups adjacent memory accesses by adjacent threads in a block into single *memory transactions*. If adjacent threads access misaligned or scattered memory addresses in a single cycle, these accesses all get serialized, greatly reducing memory bandwidth for the kernel. Thus the programmer must be very careful regarding access patterns. Global memory may or may not be cached depending on the particular card.

Further, the GPU includes various other memory spaces with different performance characteristics. There is *constant memory*, which is read-only and cached, with the cache optimized for 2D spatial locality. Each multiprocessor has a small amount of *shared memory* which all threads running on that multiprocessor can access and use to synchronize. Shared memory is much faster to access than global memory, and is often used as a programmer-managed cache. Tiling across input data by placing tiles into shared memory is a typical trick for reducing global memory traffic. Register usage must also be managed, as each multipro-

cessor's cores shares a common register pool. Overallocation of registers causes spillover into global memory, greatly reducing performance.

7.2 Imp

As mentioned above, we implement our higher-order array operators as skeletons of code with splice points where the functions they modify get inlined. Rather than implement these skeletons directly in PTX/CUDA, Parakeet provides a second intermediate language that we call Imp (for imperative). In order to inline a function into a higher-order skeleton, we first translate the function body from the higher level IL into Imp before splicing.

Imp plays the role of a syntactic sugar wrapper around PTX/CUDA–including, e.g., aspects of CUDA such as the special index registers–that simplifies our job of implementing efficient GPU versions of these operator skeletons. However, Imp does differ from CUDA 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. Local temporaries can be arrays in addition to scalars. This generalizes CUDA's use of "local" memory for spilled scalar variables.
3. 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. The shape-related aspects of Imp's syntax are shown in Figure 10.

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) or for auxilliary 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" [10] for a particular implementation strategy of some array operator. An example of a simplified Imp skeleton for **map** is shown in Figure 11. The lines with the `SPLICE` keyword are the points at which the payload function gets added to the skeleton to form a complete kernel.

7.3 Parallel Library

In addition to these skeletons, we provide a library of precompiled parallel implementations of our simple (first-order) array operators, 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 [12] for some of these operators such as **sort**, while providing our own hand-tuned implementations of others such as **where**.

Imp Shape-Related Language Syntax

definition	d	$::= f(\overline{x_m} : \overline{\tau_m}) \rightarrow (\overline{y_n} : \overline{\tau_n}, \overline{\sigma_n}) = s^+$	
shape	σ	$::= []$	(shape of scalar)
		$\mid \sigma +\!\!+ \sigma$	(concatenate)
		$\mid [z]$	(singleton)
		$\mid \sigma[\text{const} \ldots \text{const}]$	(slice)
size	z	$::= \text{dimsize}(x_i, \text{const})$	(input dimension)
		$\mid \text{rank}(x_i)$	(input rank)
		$\mid \oplus(\overline{z_m})$	(scalar primitive)

Fig. 10. Imp Language Shape-Related Syntax

Fig. 11. Imp Map Skeleton

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 [23]. 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 [9].

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. One important metric for comparing CPUs to GPUs is also performance per dollar. At a current market rate of roughly $983, the Nehalem has a peak of 0.087 GFLOP/s per dollar.

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. At a current market price of around $160, the GTX260 has a peak of over 5 GFLOP/s per dollar–over 55 times that of the Nehalem.

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 [23].

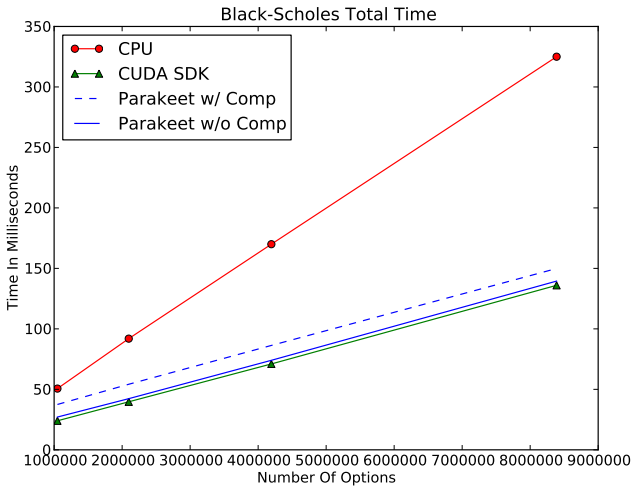

Fig. 12. Black Scholes Total Times

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

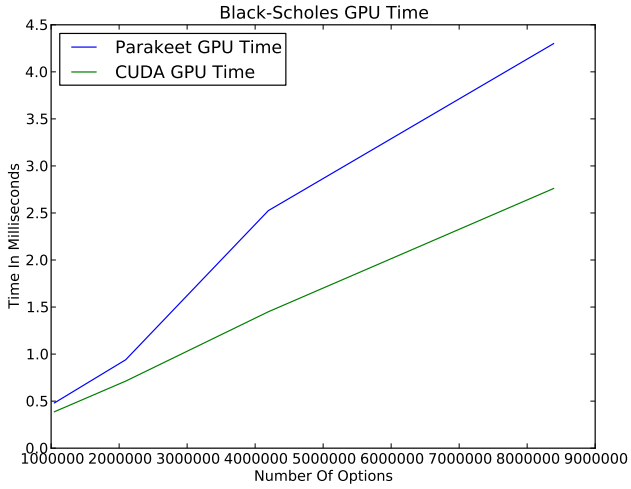

Fig. 13. Black Scholes GPU Execution Times

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 [18], 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 [28] 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** [19] 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** [8] 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 [12] 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 [17].

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. Marco Aldinucci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti. Towards parallel programming by transformation: the FAN skeleton framework. *Parallel Algorithms Appl.*, 16(2-3):87–121, 2001.
2. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 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. Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973.
4. Maximilian C. Bolingbroke and Simon L. Peyton Jones. Types are calling conventions. In *Proceedings of the 2009 Haskell Workshop*, September 2009.
5. Jeffrey A. Borror. *Q For Mortals: A Tutorial In Q Programming.* CreateSpace, 2008.
6. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.
7. Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010.
8. Manuel M.T. Chakravarty, Gabrielle Keller, Sean Lee, Trevor L. McDonnel, and Vinod Grover. Accelerating haskell array codes with multicore gpus.
9. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy Sheaffer, Sang-ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, October 2009.
10. Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
11. Wu-chun Feng and Shucai Xiao. To GPU Synchronize or Not GPU Synchronize? In *IEEE International Symposium on Circuits and Systems (ISCAS)*, Paris, France, May 2010.
12. Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.
13. Kenneth E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, AIEE-IRE '62 (Spring), pages 345–351, New York, NY, USA, 1962. ACM.
14. C. Barry Jay and Milan Sekanina. Shape checking of array programs. Technical report, In Computing: the Australasian Theory Seminar, Proceedings, 1997.
15. Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2004 Haskell Workshop*, 2001.

16. D. C. R. Ju, C. L. Wu, and P. Carini. The classification, fusion, and parallelization of array language primitives. *IEEE Trans. Parallel Distrib. Syst.*, 5:1113–1120, October 1994.
17. Chris Lattner. 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`.
18. Jed Lengyel, Mark Reicher, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *In Proc. SIGGRAPH*, pages 327–335, 1990.
19. Geoffrey Mainland and Greg Morrisett. Nikola: Embedding compiled GPU functions in haskell. In *Proceedings of the 2010 Haskell Workshop*, September 2010.
20. Cleve B. Moler. MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science, 1980.
21. Aaftab Munshi. The OpenCL specification version 1.1, September 2010.
22. NVIDIA. CUDA ZONE. `http://www.nvidia.com/cuda`.
23. NVIDIA. NVIDIA CUDA SDK 3.2. `http://www.nvidia.com/cuda`.
24. Oli07 Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9:10–20, 2007.
25. Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not.*, 25:257–271, June 1990.
26. Jay Sipelstein and Guy E. Blelloch. Collection-Oriented languages. In *Proceedings of the IEEE*, pages 504–523, 1991.
27. Joel Svensson, Mary Sheeran, and Koen Claessen. 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.
28. David Tarditi, Sidd Puri, and Jose Oglesby. 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.