Parakeet: Efficient Interpretation of High Level Array Languages to Graphical Processors

Anonymous

Anonymous

Anonymous

Abstract

The outstanding performance of contemporary Graphical Processing Units (GPUs) has led to an explosion of recent work on enabling the execution of general-purpose programs on GPUs. Nevertheless, the two commonly used frameworks (NVIDIA's CUDA and OpenCL) are cumbersome and require detailed architectural knowledge on the part of the programmer to fully harness the hardware's 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. So, 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. Parakeet transparently moves data to and from the GPU and performs garbage collection of GPU memory.

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 in enabling the execution of general-purpose programs on GPUs (GPGPU)[7, 14–16, 18, 19]. Unfortunately, the two widely used GPGPU frameworks–NVIDIA's CUDA [16] and OpenCL [15]–require the programmer to have extensive knowledge of low level architectural details in order to fully harness the hardware's performance potential. While this might be acceptable for well-trained systems hackers, professionals in other domains such as the natural sciences who could potentially benefit from the performance boost from using GPUs are unable to do so.

Our goal is to enable programmers to use productivity languages for writing efficient GPGPU programs. In their standard use, programming languages such as Python or Matlab sacrifice program speed in order to allow programmers to write code more quickly and with less effort than efficiency languages such as C. We aim to deliver both performance and ease of use by focusing on a specific class of parallel computation: data parallelism. Data parallelism is present in many important algorithms and problem domains, e.g. machine learning and linear algebra, and has been well studied in the literature (e.g. in [5]). Data parallelism is also a natural model for GPU programming, because GPUs have been heavily optimized for data parallel workloads. We thus target languages with support for *array-oriented programming*. Array-oriented programming languages support data parallel array operators (such as map, reduce, and scan) for operating on first-class array types, and discourage programmers from using loops.

The core of Parakeet is an interpreter for its high level, typed intermediate language that includes native support for array operators. When the Parakeet interpreter reaches an array operator such as `Map`, it uses dynamic runtime information to decide whether to synthesize a GPU program to implement that operator, taking advantage of the GPU transparently to the programmer.

Parakeet is agnostic as to the source language used by the programmer, provided that the language supports the required array operators. We have implemented the first front end for the Parakeet system for Q, a high level, dynamically typed array language derived from APL and widely used in financial computing [6]. Parakeet's first back end is for PTX, the pseudoassembly language used by NVIDIA GPUs [16]. We are currently working on front ends for data parallel subsets of Matlab and Python, and plan on supporting multicore CPUs in the future as well.

Parakeet executes a program according to the following pipeline:

1. intercept function calls from the source language's interpreter and type-specialize them according to their argument types.
2. perform various standard compiler optimizations on this intermediate language (IL) form of the functions, as well as fuse array operators when desirable.
3. Interpret the typed function, determining whether to execute array operators on the GPU or CPU. For GPU snippets, Parakeet synthesizes those snippets from a library of efficient array operator implementation skeletons, with splice points where user-defined functions get inlined.

Parakeet supports the nesting of array operators, using heuristics to decide which level of nesting should be compiled to GPU

```
1  f: {[x] 5 + 3 * x}
2
3  / Create input array
4  a: til 10000
5
6  / Call f on different inputs
7  b: f[6]
8  c: f[a]
```

Figure 1. Q Map Example

kernels, hile the outer levels of nesting are translated into iterative kernel invocations by our CPU runtime. Data movement and GPU garbage collection are handled transparently by Parakeet.

The main contributions of this paper are the following:

- A working system in which programmers can write complex, high level code that is automatically parallelized into efficient GPU programs.
- A series of high-level fusion optimization rules for higher-order data parallel operators.
- Evidence that dynamic compilation of GPU programs can significantly improve their performance by taking advantage of runtime information to tailor code and execution to runtime information.

2. Parakeet's Intermediate Language

There's going to be a lot of text and code examples in this section. I want to push the Q examples down further, so that we see the bullet points from the Intro section where they belong. Hence the following ipso.

There's going to be a lot of text and code examples in this section. I want to push the Q examples down further, so that we see the bullet points from the

3. Q: A High-Level Array Language

While the Parakeet framework is source language agnostic, we chose to implement its first front end for Q, a high-level, sequential array programming language from the APL family [6]. Q is dynamically typed, and the standard proprietary implementation is interpreted. Q is a natural choice for Parakeet, since idiomatic Q code uses native array types and a rich set of array operators as well 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 has a large user base in high performance financial computing. Because our focus is the Parakeet runtime, however, we present only the salient features of Q to illustrate the level of programming and Q's support for array operators, as well as to make our later code examples clear.

3.1 Q Examples

Figure 1 gives an example of a simple Q program. In line 1, we define a function `f` which takes a single argument `x` and computes a simple linear function of it. In Q syntax, the colon operator is used for assignment, and curly braces surround function definitions. Brackets are used to declare function arguments, with semicolons serving as list delimiters between the various arguments. Q functions return the result of their final statement. On line 3 we see a comment, denoted by a single `/`, and on line 4 we use the built-in function `til` to generate a 10000-element array of integers ranging from `0` to `9999`.

In line 7, we finally perform our first computation, evaluating `f` on the input `6` with the bracket function call syntax, storing the value `23` in variable `b`. Line 8 evaluates `f` element-wise on the entire array `a`, storing the result in `c`.

This code snippet illustrates a key feature of Q for our purposes. Functions are agnostic to the shape of their input arguments. In this example, `f` was legally applied both to a scalar input as well as to an array. In effect, the function `f` was *mapped* element-wise over the array `a`. Thus we see that Q programs can express data parallel computations in a compact, declarative, high level fashion. Looping is unnecessary and, in this case, discouraged.

```
1  a: til 10000
2
3  b: 0 +/ a
4  c: 0 +\ a
```

Figure 2. Q Reductions Example

```
1  sqr: {[x] x*x}
2  dist: {[x;y] sqrt sum sqr x - y}
3
4  / Generate two 100x100 matrices
5  a: (100;100) # til 10000
6  b: (100;100) # til 10000
7
8  all_dists: dist/:\:[a;b]
```

Figure 3. Q All-Pairs Distance Example

In Figure 2, we introduce the concept of an *adverb*, a higher-order function modifier. In line 3, we see an example of the reduction adverb, denoted with the syntax '`/`', which modifies a binary function. When the modified function is then applied to an array input, the result is as though the function were inserted infix between all elements of the array. Thus this adverb corresponds to the `Reduce` array operator of the Parakeet language. The reduce adverb also takes an initial value argument, which is treated as though it were prepended to the input of the reduce. Thus, after executing line 3 `b` will contain the sum of the array `a`.

In line 4, we see the *scan* adverb, with syntax '`\`'. Scan computes the running value of the function it modifies when applied element-per-element to an input array. Scan also takes an initial value argument. In Parakeet, this corresponds to the `Scan` array operator, an extremely useful building block for parallel algorithms[5].

Figure 3 shows a pair of functions–one that computes the square of an input variable, and another that computes the Euclidean distance of two input vectors. The `sum` function used in the `dist` function body is a built-in reduction function, which computes the sum of all elements of its input argument. The `dist` function also illustrates that Q code is evaluated *left of right*–the input vectors `x` and `y` are subtracted, then their square is computed, this is summed, and finally the square root of this sum is returned.

In lines 5 and 6, we generate two 100x100 matrices. The `#` operator is used to modify the shape of a variable: its left argument is a list of lengths for the dimensions, and the right argument is a variable whose shape is to be modified. Lists in Q are enclosed in parentheses.

In line 8, we evaluate the `dist` function on the matrices `a` and `b` using the *all pairs* adverb (corresponding to a cross-product).

This adverb–which is technically a combination of the adverbs *left-each* (`/:`) and *right-each* (`\:`)–applies its function argument to all pairs of each of its inputs–in this case, all pairs of rows from `a` and rows from `b`. The resulting `all_dists` variable will contain a matrix of all the Euclidean distances between rows of `a` and `b`.

We now turn to how Parakeet executes Q programs, including synthesizing GPU kernels to accelerate them.

4. Compilation Pipeline

Do All-Pairs distances here.

Here's some placeholder text for the compilation pipeline section. This section should easily take up the most of any section, around 2.5-3 pages I would imagine. Here's some placeholder text for the compilation pipeline section. This section should easily take up the most of any section, around 2.5-3 pages I would imagine.Here's some placeholder text for the compilation pipeline section. This section should easily take up the most of any section, around 2.5-3 pages I would imagine.

4.1 CUDA Back End

In this section we give an overview of modern GPU architecture and the CUDA programming model. Modern graphics processors are made up of hierarchical arrays of simple processors, with each level of the hierarchy sharing some commmon hardware resources. When performing graphics computations these processors execute shader programs for performing rendering tasks. With the advent of GPGPU programming environments such as NVIDIA's CUDA [16] and OpenCL [15], programmers can execute general-purpose programs directly on these processors as well.

The current back end for Parakeet targets the CUDA platform by emitting PTX, the pseudoassembly language in which CUDA programs are ultimately implemented. In CUDA, a program is organized into *kernels* and *host* code. Kernels are parallel code *Eric: you said sequential* that is run by typically many thousands of ultra lightweight threads on the GPU's, or *device*'s, processors. Host code is the code run on the CPU. Typically, a host thread launches a kernel onto the GPU, waits for results, and then potentially launches further kernels.

A CUDA 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 block can perform barrier synchronizations with each other via a CUDA intrinsic, but threads from different blocks have no way to perform synchronization other than for the kernel to terminate. Thus algorithms that require global synchronization 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 agnostic 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 datasizes without requiring them to be rewritten. 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.

The GPU consists of a series–typically a few hundred–simple processors that are organized into small groups (on current chips between 8 and 48) of processors called *symmetric multiprocessors* or SMs. SMs form the basic hardware scheduling unit, with processors within an SM sharing instruction fetch and decode units as well as thread blocks executing in their entirety on a single SM.

At the lowest level of the processor hierarchy are symmetric processors or SPs, which are very simple sequential processors. At the next level up are symmetric multiprocessors or SMs, which are groups of 8 SPs that share an instruction issue and decode unit. This means that all SPs within an SM execute their instructions in lockstep. In the CUDA model, thread blocks are scheduled to run in their entirety on a single SM. The SPs are then grouped to share texture processing units *Eric: first mention of these*, and then these groups along with L2 caches make up the GPU chip. The graphics card bundles the GPU along with a large block of off-chip DRAM, typically on the order of a few gigabytes in modern cards.

The CUDA model is a Single Instruction Multiple Thread or SIMT model, meaning that multiple threads with a block each execute the same instruction at the same time. This differs from SIMD models in that branching instructions are allowed which diverge with a thread block. However, this results in a significant performance degradation, as all threads in the block are then forced to execute both paths of the branch, with logical no-ops being executed by the threads on the path which didn't satisfy their conditional test.

In order to get maximum performance on a GPU, it is very important to manage memory usage. While the peak memory bandwidth on modern GPUs is up to 150GB/sec or more–up to an order of magnitude higher than that of CPUs–this bandwidth can be reached only with the proper access patterns on the part of threads within a block. In addition, each SM has a 16KB scratchpad called *shared memory*. The programmer must explicitly manage shared memory, and careful usage of it can result in orders of magnitude performance improvements over naively-written CUDA programs. Further, data must be manually moved between system RAM and the graphics card's DRAM. Modern GPUs use PCI-Express, which has a maximum transfer rate of around 4GB/sec, which can easily become the major bottleneck in CUDA programs if memory transfers aren't managed carefully. Finally, graphics cards tend to have limited global memory (typically between 500MB and 2GB), and a user's windowing system can consume much of this on its own (on our office desktops, X Windows tends to consume around 600MB). Thus GPU RAM is an important scarce resource to be managed.

Various constraints limit how a programmer can structure his or her CUDA programs. Each SM is able to execute a maximum of 8 thread blocks and a maximum of 768-1536 threads at a time. In addition, a group of thread blocks can be co-scheduled on an SM only if they consume at most 8K-32K registers, as each SM shares a register pool. Finally, no more than 16-48KB of shared memory per SM can be used at a time, and so thread blocks which in aggregate exceed this limit cannot be co-scheduled.

The complexity of programming in the CUDA platform should be evident. Parakeet aims to manage these low level details efficiently and transparently for the programmer.

5. 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 [3] benchmark suite–which we used as the basis of our Q implementation–and the GPU implementation is taken from the CUDA SDK [17]. 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 3 Intel Nehalem 2.67GHz X5650 4-core CPUs (for a total of 12 cores) with 24GB of RAM. The peak throughput of this machine is thus 32GLOP/s, and it has a peak memory bandwidth of 32GB/s.

For all GPU benchmarks, we ran the programs on both an NVIDIA GTX2XX and an NVIDIA GTX460. The GTX2XX has 240 processor cores with clock speeds of Y.YY GHz and YMB of memory, with peak execution throughput of Y.YY GFLOP/s and peak memory bandwidth of Y.YY GB/s. The GTX460 has 336 processor cores with clock speeds of 1.35GHz and 1GB of memory, and has a peak execution throughput of 907 GFLOP/s and a peak memory bandwidth of 115.2 GB/s.

```
calc_single_centroid: {[X;a;i] avg X[where a = i]}
calc_centroids:{[X;a;k]
  calc_single_centroid[X;a] each til k}
dist:{[x;y] sqrt sum (x-y) * (x-y)}
minidx:{[x] x ? min x}

kmeans:{[X;k;assignment;maxiters]
  C: calc_centroids[X;assignment;k];
  i: 0;
  converged: 0b;
  while[(i<maxiters) & not converged;
    lastAssignment: assignment;
    D: X dist/:\: C;
    assignment: minidx each D;
    C: calc_centroids[X;assignment;k];
    converged: all lastAssignment = assignment;
    i+: 1];
  (C; converged)}
```

Figure 4. Q K-Means Clustering

For our Parakeet benchmarks, the desktop system we used had a BLAH processor with BLAH specs.

5.1 Black-Scholes

Black-Scholes option pricing [4] 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 [3] benchmark suite for 1 to 12 threads and

5.2 K-Means Clustering

5.3 Dynamic Tuning

In order to test the effects of our dynamic tuning optimizations,

6. Related Work

There's Copperhead [7]. They're almost as good as we are. *We can differ along several dimensions. Higher level source code, better compiler optimizations, or better gpu handling*

There has been much recent work dedicated to making parallel programming easier. The various approaches to addressing this problem include libraries and APIs which provide parallel functionality as well as new programming languages meant for parallel programming.

Skeletons are cool [11].

Recent work on parallel programming languages includes Fortress [2], Unified Parallel C [8], and X10 [10]. Our approach differs from all of these in two important ways. First, we are focusing on compiler techniques for sequential array programming languages which do not require the programmer to explicitly manage parallelism. These techniques are meant to be generally applicable to any suitable array language, not just Q. Second, our approach doesn't introduce a new programming language specifically for writing GPGPU or parallel programs. Q is already in wide use in the financial computing community. *I'm sure we're using other optimizations as well. This is not enough.*

There has also been other work directed specifically at raising the level of abstraction above that of CUDA and OpenCL for doing GPGPU programming. Conal Elliot's Vertigo [12] was a declarative parallel language embedded in Haskell which was used for GPGPU programming. It was designed for older GPUs (pre-CUDA), and only parallelized map computations (not handling, for example, parallel reductions or all-pairs type computations). Microsoft's Accelerator [19] is a declarative GPU language embedded in C# which creates a directed acyclic graph of LINQ operations–a collection of common data querying operations such as filtering, grouping, and joining–and compiles them to (pre-CUDA) shader programs. Joel Svensson's Obsidian [18] is another declarative language embedded in Haskell which compiles at runtime to the GPU. The language semantics are inspired by hardware description languages and while it is a higher level of abstraction than directly coding CUDA, it is still much lower level than an array language such as Q. Finally, Jacket by Accelereyes [1] provides a similar setup for Matlab as we do for Q. However, Jacket requires programmers to explicitly declare data and computation to reside on the GPU, whereas our framework takes advantage of the GPU completely transparently to the programmer.

Our approach is very similar in spirit to that of the skeletal programming community [11]. We both seek to use generic parallel programming constructs which capture most of the common parallel patterns in a concise and easy-to-use language. In our specific case, these constructs are embodied by Q's adverbs, which are second-order function modifiers. We describe these adverbs in more detail in the next section. *Where?*

7. Conclusion

We have presented Parakeet, an intelligent runtime for executing high level array-oriented programs on GPUs. Parakeet alleviates programmers from having to write code that explicitly manages low level architectural details, pushing the level of abstraction for GPGPU programming up to the level of productivity languages. Parakeet automatically synthesizes GPU programs from data parallel array operators, transparently running these programs and 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 a working 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 [13].

References

[1] ACCELEREYES. Jacket. `http://www.accelereyes.com`.

[2] ALLEN, E., CHASE, D., HALLET, J., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE JR., G., AND TOBIN-HOCHSTADT, S. The Fortress language specification version 1.0, March 2008. `http://research.sun.com/projects/plrg/fortress.pdf`.

[3] 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).

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

[5] BLELLOCH, G. E. Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov 1990.

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

[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] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2005), ACM, pp. 519–538.

[9] 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.

[10] CHEN, W.-Y., IANCU, C., AND YELICK, K. Communication optimizations for fine-grained UPC applications. In *PACT '05: Proceedings of the 14th International Conference on Processors, Architectures, and Compilation Techniques* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 267–278.

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

[12] ELLIOT, C. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop* (2004), ACM Press.

[13] 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`.

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

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

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

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

[18] 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).

[19] 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).