Parakeet: A Just-In-Time Parallel Accelerator for Python

Alex Rubinsteyn Eric Hielscher Nathaniel Weinman Dennis Shasha
Computer Science Department, New York University, New York, NY, 10003
{alexr,hielscher,nsw233,shasha} @ cs.nyu.edu

Abstract

High level productivity languages such as Python or Matlab enable the use of computational resources by non-expert programmers. However, these languages often sacrifice program speed for ease of use.

This paper proposes Parakeet, a library which provides a just-in-time (JIT) parallel accelerator for Python. Parakeet bridges the gap between the usability of Python and the speed of code written in efficiency languages such as C++ or CUDA. Parakeet accelerates data-parallel sections of Python that use the standard NumPy scientific computing library. Parakeet JIT compiles efficient versions of Python functions and automatically manages their execution on both GPUs and CPUs. We assess Parakeet on a pair of benchmarks and achieve significant speedups.

1 Introduction

Numerical computing is an indispensable tool to professionals in a wide range of fields, from the natural sciences to the financial industry. Often, users in these fields either (1) aren't expert programmers; or (2) don't have time to tune their software for performance. These users typically prefer to use productivity languages such as Python or Matlab rather than efficiency languages such as C++. Productivity languages facilitate non-expert programmers by trading off program speed for ease of use [23].

One problem, however, is that the performance tradeoff is often very stark – code written in Python or Matlab [19] often has much worse performance than code written in C++ or Fortran. This problem is getting worse, as modern processors (multicore CPUs as well as GPUs) are all parallel, and current implementations of productivity languages are poorly suited for parallelism. Thus a common workflow involves prototyping algorithms in a productivity language, followed by porting the performance-critical sections to a lower level language. This second step can be time-consuming, error-prone, and it diverts energy from the real focus of these users' work.

In this paper, we present Parakeet, a library that provides a JIT parallel accelerator for NumPy, the commonly-used scientific computing library for Python [22]. Parakeet accelerates performance-critical sections of numerical Python programs to be competitive with efficiency language code, obviating the need for the above-mentioned "prototype, port" cycle.

The Parakeet library intercepts programmer-marked functions and uses high-level operations on NumPy arrays (e.g. mapping a function over the array's elements) as sources of parallelism. These functions are just-in-time compiled to either x86 machine code using LLVM [17] or GPU code that can be executed on NVIDIA GPUs via the CUDA framework [20]. These native versions of the functions are then automatically executed on the appropriate hardware. Parakeet allows complete interoperability with all of the standard Python tools and libraries.

Parakeet currently supports JIT compilation to parallel GPU programs and single-threaded CPU programs. While Parakeet is a work in progress, our current results clearly demonstrate its promise.

2 Overview

Parakeet is an accelerator library for numerical Python algorithms written using the NumPy array extensions [22]. Parakeet does not replace the standard Python runtime but rather selectively augments it. To run a function within Parakeet a user must wrap it with the decorator `@PAR`. For example, consider the following NumPy code for averaging the value of two arrays:

```
@PAR
def avg(x,y):
  return (x+y) / 2.0
```

If the decorator `@PAR` were removed, then `avg` would run as ordinary Python code. Since NumPy's library functions are compiled separately they always allocate result arrays (even when the arrays are immediately consumed). By contrast, Parakeet specializes `avg` for any distinct input

type, optimizes its body into a singled fused map (avoiding unnecessary allocation) and executes it as parallel native code.

Parakeet is not meant as a general-purpose accelerator for all Python programs. Rather, it is designed to execute array-oriented numerical algorithms such as those found in machine learning, financial computing, and scientific simulation. In particular, the sections of code that Parakeet accelerates must obey the following constraints:

- Due to the difficulty of implementing efficient non-uniform data structures on the GPU, we require all values within Parakeet to be either scalars or NumPy arrays. No dictionaries, sets, or user-defined objects are allowed.
- To compile Python into native code we must assign types to each expression. We are still able to retain some of Python's polymorphism by specializing different typed versions of a function for each distinct set of argument types. However, expressions whose types depend on dynamic values are disallowed (e.g. `42 if bool_val else "sausage"`).
- Only functions which don't modify global state or perform I/O can be executed in parallel. Local mutable variables are always allowed.

A Parakeet function cannot call any other function which violates these restrictions or one which is not implemented in Python. To enable the use of NumPy library functions Parakeet must provide equivalent functions written in Python. In general, these restrictions would be onerous if applied to an entire program but Parakeet is only intended to accelerate the computational core of an algorithm. All other code is executed as usual by the Python interpreter.

Though Parakeet supports loops, it does not parallelize them in any way. Parallelism is instead achieved through the use of the following data parallel operators:

- **map**(f, $X_1, ..., X_n$, *fixed*=[], *axis*=None)
  Apply the function f to each element of the array arguments. By default, f is passed each scalar element of the array arguments.
  The *axis* keyword can be used to specify a different iteration pattern (such as applying f to all columns).
  The *fixed* keyword is a list of closure arguments for the function *f*.
- **allpairs**(f, X_1, X_2, *fixed*=[], *axis*=0)
  Apply the function f to each pair of elements from the arrays X_1 and X_2.
- **reduce**(f, $X_1, ..., X_n$, *fixed*=[], *axis*=None, *init*=None)
  Combine all the elements of the array arguments using the $(n+1)$-ary commutative function f. The *init* keyword is an optional initial value for the reduction. Examples of reductions are the NumPy functions `sum` and `product`.
- **scan**(f, $X_1, ..., X_n$, *fixed*=[], *axis*=None, *init*=None)
  Combine all the elements of the array arguments and return an array containing all cumulative intermediate values of the combination. Examples of scans are the NumPy functions `cumsum` and `cumprod`.

For each occurrence of a data parallel operator in a program, Parakeet may choose to synthesize parallel code which implements that operator combined with its function argument. It is not always necessary, however, to explicitly use one of these operators in order to achieve parallelization. Parakeet implements NumPy's array broadcasting semantics by implicitly inserting calls to **map** into a user's code. Furthermore, NumPy library functions are reimplemented in Parakeet using the above data parallel operators and thus expose opportunities for parallelism.

3 Parakeet Runtime and Internals

We will refer to the following code example to help illustrate the process by which Parakeet transforms and executes code.

```
def add(x,y):
  return x + y

def sum(x):
  return parakeet.reduce(add, x)

@PAR
def norm(x):
  return math.sqrt(sum(x*x))
```

Listing 1: Vector Norm in Parakeet

When the Python interpreter reaches the definition of `norm`, it invokes the `@PAR` decorator which parses the function's source and translates it into Parakeet's untyped internal representation. There is a fixed set of primitive functions from NumPy and the Python standard library, such as `math.sqrt`, which are translated directly into Parakeet syntax nodes. The helper functions `add` and `sum` would normally be in the Parakeet module but they are defined here for clarity. These functions are non-primitive, so they themselves get recursively parsed and translated. In general, the `@PAR` decorator will raise an exception if it encounters a call to a non-primitive function which either can't be parsed or violates Parakeet's semantic re-

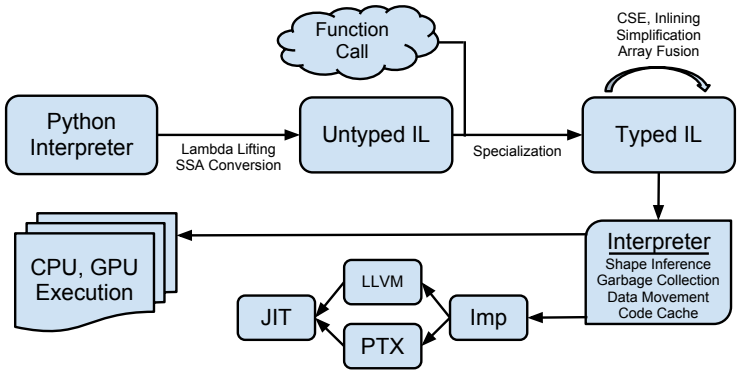

Figure 1: Parakeet Pipeline

strictions. Lastly, before returning execution to Python, Parakeet converts its internal representation to Static Single Assignment form [12].

3.1 Type Specialization

Parakeet intercepts calls to `norm` and uses the argument types to synthesize a typed version of the function. During specialization, all functions called by `norm` are themselves specialized for particular argument types. In our code example, if `norm` were called with a 1D `float` array then sum would also be specialized for the same input type, whereas `add` would be specialized for pairs of scalar `float`s.

In Parakeet's typed representation, every function must have unambiguous input and output types. To eliminate polymorphism Parakeet inserts casts and **map** operators where necessary. When `norm` is specialized for vector arguments, its use of the multiplication operator is rewritten into a 1D **map** of a scalar multiply.

The actual process of type specialization is implemented by interleaving an abstract interpreter, which propagates input types to infer local types, and a rewrite engine which inserts coercions where necessary.

3.2 Optimization

In addition to standard compiler optimizations (such as constant folding, function inlining, and common subexpression elimination), we employ fusion rules [2, 15, 16] to combine array operators. Fusion enables us to increase the computational density of generated code and to avoid the creation of unnecessary array temporaries.

3.3 Execution

We have implemented three backends for Parakeet thus far: CPU and GPU backends for JIT compiling native code, as well as an interpreter for handling functions that cannot themselves be parallelized but may contain nested parallelizable operators.

Once a function has been type specialized and optimized, it is handed off to Parakeet's scheduler which is responsibile for choosing among these three backends. For each array operator, the scheduler employs a cost-based heuristic which considers nested array operators, data sizes, and memory transfer costs to decide where to execute it.

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

When the scheduler encounters a nested array operator – e.g. a **map** whose payload function is itself a **reduce** – it needs to choose which operator, if any, will be parallelized. If an array operator is deemed a good candidate for native hardware execution, the function argument to the operator is then inlined into a program skeleton that implements the operator. Parakeet flattens all nested array computations within the function argument into sequential loops.

Several systems similar to Parakeet [8, 9] generate GPU programs by emitting CUDA code which is then compiled by NVIDIA's CUDA nvcc toolchain. Instead, Parakeet emits PTX (a GPU pseudo-assembly) directly, since the compile times are dramatically shorter. To generate CPU code we use the LLVM compiler framework [17].

4 Evaluation

We evaluate Parakeet on two benchmarks: Black-Scholes option pricing, and K-Means Clustering. We compare Parakeet against hand-tuned CPU and GPU implementa-

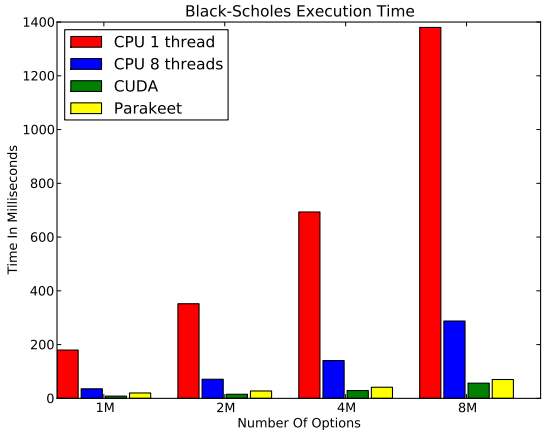

Figure 2: Black Scholes Total Times

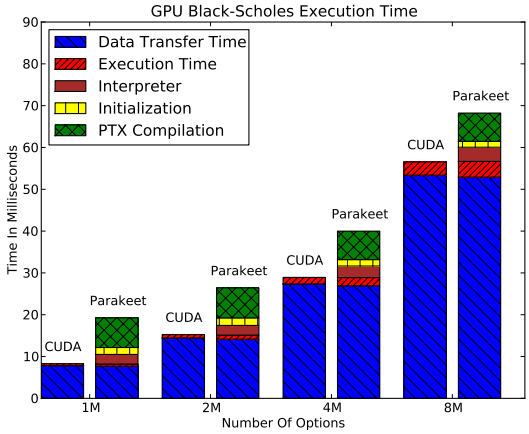

Figure 3: Black Scholes GPU Execution Times

tions. Due to space constraints, and since at the time of writing our CPU backend only supports single-threaded execution, we only present GPU results for Parakeet. For Black-Scholes, the CPU reference implementation is taken from the PARSEC [4] benchmark suite, and the GPU implementation is taken from the CUDA SDK [21]. For K-Means Clustering both the CPU and GPU reference versions come from the Rodinia benchmark suite [11]. For both benchmarks, we ported the reference implementations as directly as possible from their source languages to Parakeet.

We ran all of our benchmarks on a machine running 64-bit Linux with an Intel Core i7 3.2GHz 960 4-core CPU and 16GB of RAM. The GPU used in our system was an NVIDIA Tesla C1060 with 240 vector lanes, a clock speed of 1.296 GHz, and 4GB of memory.

4.1 Black-Scholes

Black-Scholes option pricing [5] is a standard algorithm used for data parallel benchmarking. We compare Parakeet against the multithreaded OpenMP CPU implementation from the PARSEC [4] suite with both 1 and 8 threads and the GPU version in the CUDA SDK [21]. We modified the benchmarks to all use the input data from the PARSEC implementation so as to have a direct comparison of the computation alone. We also modified the CUDA version to calculate only one of the call or put price per option so as to match the behavior in PARSEC.

In Figure 2, we see the total execution times of the various versions. These times include the time it takes to transfer data to and from the GPU in the GPU benchmarks. As expected, Black Scholes performs very well on the GPU as compared with the CPU. We see that Parakeet performs very similarly to the hand-written CUDA version, with overheads decreasing as a percentage of the run time as the data sizes grow since most of them are fixed costs related to dynamic compilation.

In Figure 3, we break down Parakeet's performance as compared with the hand-written CUDA version. The Parakeet run times range from 24% to 2.4X slower than those of CUDA, with Parakeet performing better as the data size increases. We can see that transferring data to and from the GPU's memory is expensive and dominates the runtime of this benchmark. The GPU programs that Parakeet generates are slightly less efficient than those of the CUDA version, with approximately 50% higher run time on average. Most of this slowdown is due to compilation overheads.

4.2 K-Means Clustering

We also tested Parakeet on K-Means clustering, a commonly used unsupervised learning algorithm. We chose K-Means since it includes both loops and nested array operators, and thus illustrates Parakeet's support for both. The Parakeet code we used to run our benchmarks can be seen in Listing 2.

In Figure 4, we see the total run times of K-Means for the CPU and GPU versions with K = 3 clusters and 30 features on varying numbers of data points. Here, the distinction between the GPU and the CPU is far less stark. In fact, for up to 64K data points the 8-thread CPU version outperforms the GPU. Further, we see that for more than 64K data points, Parakeet actually performs better than both the CPU and GPU versions.

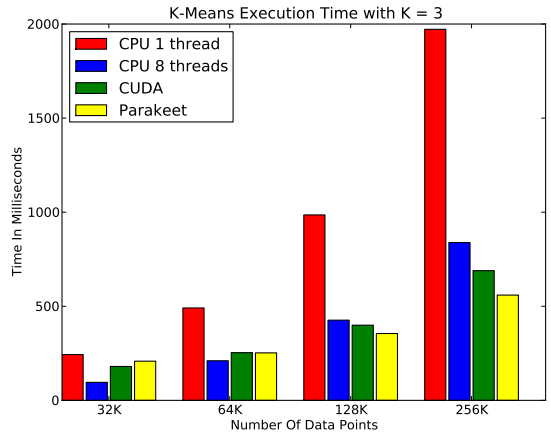

Figure 4: K-Means Total Times with 30 Features, K = 3

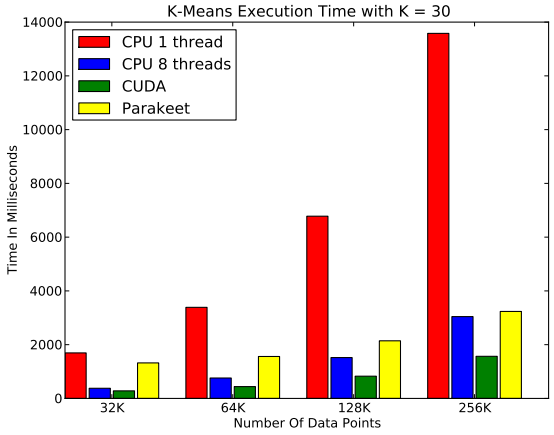

Figure 5: K-Means Total Times with 30 Features, K = 30

The reason Parakeet is able to perform so well with respect to the CUDA version is due to the difference in how they compute the new average centroid for the new clusters in each iteration. The CUDA version brings the GPU-computed assignment vectors back to the CPU in order to perform this reduction, as it involves many unaligned memory accesses and so has the potential to perform poorly on the GPU. Parakeet's scheduler chooses to execute this code on the GPU instead, prefering to avoid the data transfer penalty. For such a small number of clusters, the Parakeet method ends up performing far better. However, for larger numbers of clusters (roughly 30 and above), the fixed overhead of launching an individual kernel to average each cluster's points overwhelms the performance advantage of the GPU and Parakeet ends up performing worse than the CUDA version. We are currently implementing better code placement heuristics based on dynamic information, in order to use the GPU only when it would actually be advantageous.

```
import parakeet as par
from parakeet import PAR

def sqr_dist(x, y):
  sqr_diff = (x-y)*(x-y)
  return par.sum(sqr_diff)

def minidx(C, x):
  dists = par.map(sqr_dist, C, fixed=[x])
  return par.argmin(dists)

def calc_centroid(X, a, cluster_id):
  return par.mean(X[a == cluster_id])

@PAR
def kmeans(X, assignments):
  k = par.max(assignments)
  cluster_ids = par.arange(k)
  C = par.map(calc_centroid, cluster_ids,
      fixed=[X, a])
  converged = False
  while not converged:
    last = assignments
    a = par.map(minidx, X, fixed=[C])
    C = par.map(calc_centroid,
      cluster_ids, fixed=[X, a])
    converged = par.all(last == assignments)
  return C
```

Listing 2: K-Means Parakeet Code

In Figure 5, we present the run times of K-Means for 30 features and K = 30 clusters. In this case, the hand-written CUDA version performs best in all cases, though its advantage over the CPU increases and its advantage over Parakeet decreases with increasing data size. As discussed, the Parakeet implementation suffers from the poorly performing averaging computation that it executes on the GPU, with a best case of an approximate 2X slowdown over the CUDA version.

Figure 6 shows a breakdown of the different factors that contribute to the overall K-Means execution times for the K = 3 clusters case. Both Rodinia's CUDA version and Parakeet use the CPU to perform a significant amount of the computation. The Rodinia version uses the CPU to compute the averages of the new clusters in each iteration, while Parakeet spends much of its CPU time on program analysis and transformation; garbage collection; and setting up GPU kernel launches. As mentioned above, Parakeet spends much more time on the GPU than the Rodinia version does. In this benchmark, as opposed to Black-Scholes, little of the execution time is spent on data

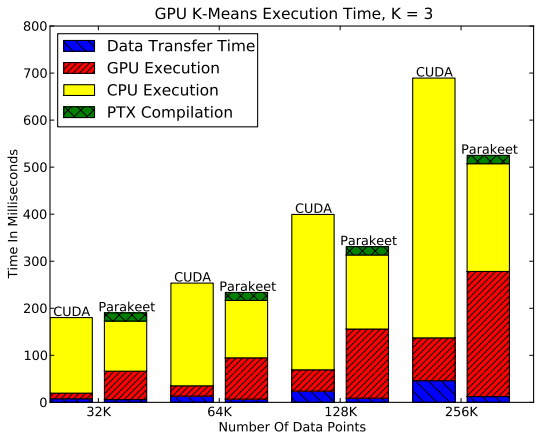

Figure 6: K-Means GPU Times with 30 Features, K = 3

transfers. Importantly, the Parakeet implementation of K-Means also has orders of magnitude fewer lines of code than the CUDA implementation.

5 Related Work

The earliest attempts to accelerate array-oriented programs were sequential compilers for APL. Abrams [1] designed an innovative two-stage compiler wherein APL was first statically translated to a high-level untyped intermediate representation and then lazily compiled to a low-level scalar architecture. Inspired by Abrams' work, Van Dyke created an APL environment for the HP 3000 [25] which also dynamically compiled delayed expressions and cached generated code using the type, rank, and shape of variables.

More recently, a variety of compilers have been created for the Matlab programming language [19]. FALCON [14] performs source-to-source translation from Matlab to Fortran 90 and achieves parallelization by adding dependence annotations into its generated code. MaJIC [3] accelerates Matlab by pairing a lightweight dynamic compiler with a static translator in the same spirit as FALCON. Both FALCON and MaJIC infer simultaneous type and shape information for local variables by dataflow analysis.

The use of graphics hardware for non-graphical computation has a long history [18]. The Brook language extended C with "kernels" and "streams", exposing a programming model similar to what is now found in CUDA and OpenCL [7]. Over the past decade, there have been several attempts to use an embedded DSL to simplify the arduous task of GPGPU programming. Microsoft's Accelerator [24] was the first project to use high level (collection-oriented) language constructs as a basis for GPU execution. Accelerator's programming model does not support function abstractions (only expression trees) and its underlying parallelism construct is limited to the production of **map**-like kernels. Accelerate [10] is a first-order array-oriented language embedded in Haskell which attains good performance on preliminary benchmarks but is unable to describe nested computations and requires user annotations to move data onto the GPU. A more sophisticated runtime has been developed for the Delite project [6], which is able to schedule complex computations expressed in Scala across multiple backends.

Parakeet and Copperhead [8] both attempt to parallelize a numerical subset of Python using runtime compilation structured around higher-order array primitives. Parakeet and Copperhead also both support nested array computations and are able to target either GPUs or CPUs. Since Copperhead uses a purely functional intermediate language, its subset of Python is more restrictive than the one which Parakeet aims to accelerate. Unlike Copperhead, Parakeet allows loops and modification of local variables. Copperhead infers a single type for each function using an extension of the Hindley-Milner [13] inference algorithm. This simplifies compilation to C++ templates but disallows both polymorphism between Python scalars and "array broadcasting" [22]. Parakeet, on the other hand, is able to support polymorphic language constructs by specializing a function's body for each distinct set of argument types and inserting coercions during specialization. Copperhead does not use dynamic information when making scheduling decisions and thus must rely on user annotations. Parakeet's scheduler dynamically chooses which level of a nested computation to parallelize.

6 Conclusion

Parakeet allows the programmer to write Python code using a widely-used numerical computing library while achieving good performance on modern parallel hardware. Parakeet automatically synthesizes and executes efficient native binaries from Python code. Parakeet is a usable system in which moderately complex programs can be written and executed efficiently. On two benchmark programs, Parakeet delivers performance competitive with hand-tuned GPU implementations. Parakeet code can coexist with standard Python code, allowing full interoperability with all of Python's tools and libraries. We are currently extending our CPU backend to use multiple cores and improving dynamic scheduling and compilation decisions.

References

[1] ABRAMS, P. S. *An APL machine*. PhD thesis, Stanford, CA, USA, 1970. AAI7022146.

[2] ABU-SUFAH, W. A.-K. *Improving the performance of virtual memory computers.* PhD thesis, Champaign, IL, USA, 1979. AAI7915307.

[3] ALMÁSI, G., AND PADUA, D. Majic: Compiling matlab for speed and responsiveness. In *PLDI '02: Proceedings of the 2002 ACM SIGPLAN conference on Programming Languages Design and Implementation* (New York, NY, USA, 2002), ACM, pp. 294–303.

[4] 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).

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

[6] BROWN, K., SUJEETH, A., LEE, H. J., ROMPF, T., CHAFI, H., ODERSKY, M., AND OLUKOTUN, K. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (oct. 2011), pp. 89 –100.

[7] 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.

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

[9] 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.

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

[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] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13* (October 1991), 451–490.

[13] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1982), POPL '82, ACM, pp. 207–212.

[14] DEROSE, L., GALLIVAN, K., GALLOPOULOS, E., MARSOLF, B., AND PADUA, D. Falcon: A matlab interactive restructuring compiler. In *IN LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING* (1995), Springer-Verlag, pp. 269–288.

[15] 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).

[16] KENNEDY, K., AND MCKINLEY, K. S. Typed fusion with applications to parallel and sequential code generation. Tech. rep., 1993.

[17] 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`.

[18] 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.

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

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

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

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

[23] PRECHELT, L. Are scripting languages any good? a validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. *Advances in Computers 57* (2003), 205–270.

[24] 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).

[25] VAN DYKE, E. J. A dynamic incremental compiler for an interpretive language. *Hewlett-Packard Journal* (July 1977), 17–24.