Copperhead: Compiling an Embedded Data Parallel Language

Bryan Catanzaro
University of California, Berkeley
catanzar@cs.berkeley.edu

Michael Garland
NVIDIA Research
mgarland@nvidia.com

Kurt Keutzer
University of California, Berkeley
keutzer@cs.berkeley.edu

Abstract

Modern parallel microprocessors deliver high performance on applications that expose substantial fine-grained data parallelism. Although data parallelism is widely available in many computations, implementing data parallel algorithms in low-level languages is often an unnecessarily difficult task. The characteristics of parallel microprocessors and the limitations of current programming methodologies motivate our design of Copperhead, a high-level data parallel language embedded in Python. The Copperhead programmer describes parallel computations via composition of familiar data parallel primitives supporting both flat and nested data parallel computation on arrays of data. Copperhead programs are expressed in a subset of the widely used Python programming language and interoperate with standard Python modules, including libraries for numeric computation, data visualization, and analysis.

In this paper, we discuss the language, compiler, and runtime features that enable Copperhead to efficiently execute data parallel code. We define the restricted subset of Python which Copperhead supports and introduce the program analysis techniques necessary for compiling Copperhead code into efficient low-level implementations. We also outline the runtime support by which Copperhead programs interoperate with standard Python modules. We demonstrate the effectiveness of our techniques with several examples targeting the CUDA platform for parallel programming on GPUs. Copperhead code is concise, on average requiring 3.6 times fewer lines of code than CUDA, and the compiler generates efficient code, yielding 45-100% of the performance of hand-crafted, well optimized CUDA code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Compilers; D.3.2 [*Language Classifications*]: Parallel languages; D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Algorithms, Design, Performance

Keywords Python, Data Parallelism, GPU

1. Introduction

As the transition from sequential to parallel processing continues, the need intensifies for high-productivity parallel programming tools and methodologies. Manual implementation of parallel programs using efficiency languages such as C and C++ can yield high performance, but at a large cost in programmer productivity, which some case studies show as being 2 to 5 times worse than productivity-oriented languages such as Ruby and Python [15, 25]. Additionally, parallel programming in efficiency languages is often viewed as an esoteric endeavor, to be undertaken by experts only. Without higher-level abstractions that enable easier parallel programming, parallel processing will not be widely utilized. Conversely, today's productivity programmers do not have the means to capitalize on fine-grained, highly parallel microprocessors, which limits their ability to implement computationally intensive applications. To enable widespread use of parallel processors, we need to provide higher level abstractions which are both productive and efficient.

Although the need for higher-level parallel abstractions seems clear, perhaps less so is the type of abstractions which should be provided, since there are many potential abstractions to choose from. In our view, nested data parallelism, as introduced by languages such as NESL [4], is particularly interesting. Nested data parallelism is abundant in many computationally intensive algorithms. It can be mapped efficiently to parallel microprocessors, which prominently feature hardware support for data parallelism. For example, mainstream x86 processors from Intel and AMD are adopting 8-wide vector instructions, Intel's Larrabee processor used 16-wide vector instructions, and modern GPUs from NVIDIA and AMD use wider SIMD widths of 32 and 64, respectively. Consequently, programs which don't take advantage of data parallelism relinquish substantial performance, on any modern processor.

Additionally, nested data parallelism as an abstraction clearly exposes parallelism. In contrast to traditional auto-parallelizing compilers, which must analyze and prove which operations can be parallelized and which can not, the compiler of a data-parallel language needs only to decide which parallel operations should be performed sequentially, and which should be performed in parallel. Accordingly, nested data parallel programs have a valid sequential interpretation, and are thus easier to understand and debug than parallel programs that expose race conditions and other complications of parallelism.

Motivated by these observations, Copperhead provides a set of nested data parallel abstractions, expressed in a restricted subset of the widely used Python programming language. Instead of creating an entirely new programming language for nested data parallelism, we repurpose existing constructs in Python, such as `map` and `reduce`. Embedding Copperhead in Python provides several important benefits. For those who are already familiar with Python, learning Copperhead is more similar to learning how to use a Python package than it is to learning a new language. There is no need to learn any new syntax, instead the programmer must learn only what subset of Python is supported by the Copperhead language and runtime. The Copperhead runtime is implemented as a standard Python extension, and Copperhead programs are invoked through the Python interpreter, allowing them to interoperate with the wide variety of existing Python libraries for numeric computa-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0119-0/11/02…$10.00

tion, file manipulation, data visualization, and analysis. This makes Copperhead a productive environment for prototyping, debugging, and implementing entire applications, not just their computationally intense kernels.

Of course, without being able to efficiently compile nested data parallel programs, Copperhead would not be of much use. To this end, the Copperhead compiler attempts to efficiently utilize modern parallel processors. Previous work on compilers for nested data parallel programming languages has focused on flattening transforms to target flat SIMD arrays of processing units. However, today's processors support a hierarchy of parallelism, with independent cores containing tightly coupled processing elements. Accordingly, the Copperhead compiler maps nested data parallel programs to a parallelism hierarchy, forgoing the use of flattening transforms. We find that in many circumstances, this is substantially more efficient than indiscriminate application of the flattening transform.

The current Copperhead compiler targets CUDA C++, running on manycore Graphics Processors from NVIDIA. We generate efficient, scalable parallel programs, performing within 45-100% of well optimized, hand-tuned CUDA code. Our initial implementation focus on CUDA does not limit us to a particular hardware platform, as the techniques we have developed for compiling data-parallel code are widely applicable to other platforms as well. In the future, we anticipate the development of Copperhead compiler backends for other parallel platforms. Additionally, the compiler techniques we propose are suitable for compiling other data parallel languages, such as NESL and Data Parallel Haskell.

2. Related Work

There is an extensive literature investigating many alternative methods for parallel programming. Data parallel approaches [2, 3, 13] have often been used, and have historically been most closely associated with SIMD and vector machines, such as the the CM-2 and Cray C90, respectively. Blelloch *et al.* designed the NESL language [4], demonstrating a whole-program transformation of nested data parallel programs into flat data parallel programs. The flattening transform has been extended to fully higher-order functional languages in Data Parallel Haskell [7]. In contrast to these methods, we attempt to schedule nested procedures directly onto the hierarchical structure of the machines we target.

The CUDA platform [23, 26] defines a blocked SPMD programming model for executing parallel computations on GPUs. Several libraries, such as Thrust [14], provide a collection of flat data parallel primitives for use in CUDA programs. Copperhead uses selected Thrust primitives in the code it generates.

Systems for compiling flat data parallel programs for GPU targets have been built in a number of languages, including C# [28], C++ [21, 22], and Haskell [19]. Such systems typically define special data parallel array types and use operator overloading and metaprogramming techniques to build expression trees describing the computation to be performed on these arrays. The Ct [11] library adopts a similar model for programming multicore processors. However, these systems have not provided a means to automatically map nested data parallel programs to a hierarchically nested parallel platform.

Rather than providing data parallel libraries, others have explored techniques that mark up sequential loops to be parallelized into CUDA kernels [12, 20, 30]. In these models, the programmer writes an explicitly sequential program consisting of loops that are parallelizable. The loop markups, written in the form of C pragma preprocessor directives, indicate to the compiler that loops can be parallelized into CUDA kernels.

There have also been some attempts to compile various subsets of Python to different platforms. The Cython compiler [9] compiles a largely Python-like language into sequential C code. Clyther [8] takes a similar approach to generating OpenCL kernels. In both cases, the programmer writes a Python program which is transliterated into an isomorphic C program. Rather than transliterating Python programs, PyCUDA [18] provides a metaprogramming facility for textually generating CUDA kernel programs, as well as Python/GPU bindings. Theano [29] provides an expression tree facility for numerical expressions on numerical arrays. Garg and Amaral [10] recently described a technique for compiling Python loop structures and array operations into GPU-targeted code.

These Python/GPU projects are quite useful; in fact Copperhead's runtime relies on PyCUDA. However, to write programs using these tools, programmers must write code equivalent to the *output* of the Copperhead compiler, with all mapping and scheduling decisions made explicit in the program itself. Copperhead aims to solve a fairly different problem, namely compiling a program from a higher level of abstraction into efficiently executable code.

3. Copperhead Language

A Copperhead program is a Python program that imports the Copperhead language environment:

```
from copperhead import *
```

A Copperhead program is executed, like any other Python program, by executing the sequence of its top-level statements. Selected procedure definitions within the body of the program may be marked with the Copperhead decorator, as in the following:

```
@cu
def add_vectors(x, y):
    return map(lambda xi,yi: xi+yi, x, y)
```

This `@cu` decorator declares the associated procedure to be a Copperhead procedure. These procedures must conform to the requirements of the Copperhead language, and they may be compiled for and executed on any of the parallel platforms supported by the Copperhead compiler. Once defined, Copperhead procedures may be called just like any other Python procedure, both within the program body or, as shown below, from an interactive command line.

```
>>> add_vectors(range(10), [2]*10)
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

The `@cu` decorator interposes a wrapper object around the native Python function object that is responsible for compiling and executing procedures on a target parallel platform.

Copperhead is fundamentally a data parallel language. It provides language constructs, specifically `map`, and a library of primitives such as `reduce`, `gather`, and `scatter` that all have intrinsically parallel semantics. They operate on 1-dimensional arrays of data that we refer to as *sequences*.

3.1 Restricted Subset of Python

The Copperhead language is a restricted subset of the Python 2.6 language. Every valid Copperhead procedure must also be a valid Python procedure. Portions of the program outside procedures marked with the `@cu` decorator are normal Python programs that will be executed by the standard Python interpreter, without any restrictions imposed by the Copperhead runtime or compiler. The Copperhead language supports a very restricted subset of Python in order to enable efficient compilation.

Copperhead adopts the lexical and grammatical rules of Python. In summarizing its restricted language, we denote expressions by E and statements by S. We use lower-case letters to indicate identifiers and F and A to indicate function-valued and array-valued expressions, respectively.

3.1.1 Expressions

The most basic expressions are literal values, identifier references, tuple constructions, and accesses to array elements.

E : x | (E_1, ..., E_n) | A[E]
 | **True** | **False** | integer | floatnumber

The basic logical and arithmetic operators are allowed, as are Python's and, or, and conditional expressions.

| $E_1 + E_2$ | $E_1 < E_2$ | **not** E | ...
| E_1 **and** E_2 | E_1 **or** E_2 | E_1 **if** E_p **else** E_2

Expressions that call and define functions must use only positional arguments. Optional and keyword arguments are not allowed.

| F(E_1, ..., E_n) | **lambda** x_1, ..., x_n: E

Copperhead relies heavily on map as the fundamental source of parallelism and elevates it from a built-in function (as in Python) to a special form in the grammar. Copperhead also supports a limited form of Python's list comprehension syntax, which is de-sugared into equivalent map constructions during parsing.

| **map**(F, A_1, ..., A_n)
| [E **for** x **in** A]
| [E **for** x_1, ..., x_n **in zip**(A_1, ..., A_n)]

In addition to these grammatical restrictions, Copperhead expressions must be *statically well-typed*. We employ a standard Hindley-Milner style polymorphic type system. Static typing provides richer information to the compiler that it can leverage during code generation, and also avoids the run-time overhead of dynamic type dispatch.

3.1.2 Statements

The body of a Copperhead procedure consists of a *suite* of statements: one or more statements S which are nested by indentation level. Each statement S of a suite must be of the following form:

S : **return** E
 | x_1, ..., x_n = E
 | **if** E: suite **else**: suite
 | **def** f(x_1, ..., x_n): suite

Copperhead further requires that every execution path within a suite must return a value, and all returned values must be of the same type. This restriction is necessary since every Copperhead procedure must have a well-defined static type.

All data in a Copperhead procedure are considered immutable values. Thus, statements of the form x = E bind the value of the expression E to the identifier x; they *do not* assign a new value to an existing variable. All identifiers are strictly lexically scoped.

Copperhead does not guarantee any particular order of evaluation, other than the partial ordering imposed by data dependencies in the program. Python, in contrast, always evaluates statements from top to bottom and expressions from left to right. By definition, a Copperhead program must be valid regardless of the order of evaluations, and thus Python's mandated ordering is one valid ordering of the program.

Because mutable assignment is forbidden and the order of evaluation is undefined, the Copperhead compiler is free to reorder and transform procedures in a number of ways. As we will see, this flexibility improves the efficiency of generated code.

3.2 Data Parallel Primitives

Copperhead is a data parallel language. Programs manipulate data sequences by applying aggregate operations, such as map, or reduce. The semantics of these primitives are implicitly parallel: they may always be performed by some parallel computation, but may also be performed sequentially.

```
@cu
def spmv_csr(vals, cols, x):
    def spvv(Ai, j):
        z = gather(x, j)
        return sum(map(lambda Aij, xj: Aij*xj, Ai, z))

    return map(spvv, vals, cols)
```

Figure 1. Procedure for computing Ax for a matrix A in CSR form and a dense vector x. Underlined operations indicate potential sources of parallel execution.

Table 1 summarizes the main aggregate operations used by the examples in this paper. They mirror operations found in most other data parallel languages. With two minor exceptions, these functions will produce the same result regardless of whether they are performed in parallel or sequentially. The permute primitive is the primary exception. If the indices array given to permute contains one or more repeated index values, the resulting sequence is non-deterministic since we guarantee no particular order of evaluation. The other exception is reduce. If given a truly commutative and associative operation, then its result will always be identical. However, programs often perform reductions using floating point values whose addition, for instance, is not truly associative. These problems are commonly encountered in general parallel programming, and are not unique to Copperhead.

To demonstrate our use of these operators, Figure 1 shows a simple Copperhead procedure for computing the sparse matrix-vector product (SpMV) $y = Ax$. Here we assume that A is stored in Compressed Sparse Row (CSR) format—one of the most frequently used representation for sparse matrices—and that x is a dense vector. The matrix representation simply records each row of the matrix as a sequence containing its non-zero values along with a corresponding sequence recording the column index of each value. A simple example of this representation is:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

```
vals = [[1,7],[2,8],[5,3,9],[6,4]]
cols = [[0,1],[1,2],[0,2,3],[1,3]]
```

The body of spmv_csr applies a sparse dot product procedure, spvv, to each row of the matrix using map. The sparse dot product itself produces the result y_i for row i by forming the products $A_{ij}x_j$ for each column j containing a non-zero entry, and then summing these products together. It uses gather to fetch the necessary values x_j from the dense vector x and uses sum, a convenient special case of reduce where the operator is addition, to produce the result.

This simple example illustrates two important issues that are a central concern for our Copperhead compiler. First, it consists of a number of potentially parallel aggregate operations, which are underlined. Second, these aggregate operators are nested: those within spvv are executed within an enclosing map operation invoked by spmv_csr. One of the principal tasks of our compiler is to decide how to schedule these operations for execution. Each of them may be performed in parallel or by sequential loops, and the nesting of operations must be mapped onto the hardware execution units in an efficient fashion.

4. Compiler

The Copperhead compiler is a source-to-source compiler responsible for converting a suite of one or more Copperhead procedures

```
y = replicate(a, n)          Return an n-element sequence whose every element has value a.
y = map(f, x1, ..., xn)      Returns sequence [f(x1[0], ..., xn[0]), f(x1[1], ..., xn[1]), ...].
y = zip(x1, x2)              Return sequence of pairs [(x1[0],x2[0]), (x1[1],x2[1]), ...].
y = gather(x, indices)       Produce sequence with y[i] = x[indices[i]].
y = permute(x, indices)      Produce y where y[indices[i]] = x[i].
s = reduce(⊕, x, prefix)     Computes prefix ⊕ x[0] ⊕ x[1] ⊕ ... for a commutative and associative operator ⊕.
y = scan(f, x)               Produce y such that y[0]=x[0] and y[i] = f(y[i-1], x[i]). Function f must be associative.
```

Table 1. Selected data-parallel operations on sequences.

```
# lambda0 :: (a, a) → a
def lambda0(Aij, xj):
    return op_mul(Aij, xj)

# spvv :: ([a], [Int], [a]) → [a]
def spvv(Ai, j, _k0):
    z0 = gather(_k0, j)
    tmp0 = map(lambda0, Ai, z0)
    return sum(tmp0)

# spmv_csr :: ([[a]], [[Int]], [a]) → [a]
def spmv_csr(vals, cols, x):
    return map(closure([x], spvv), vals, cols)
```

Figure 2. SpMV procedure from Figure 1 after transformation by the front end compiler.

into a code module that can execute on the target parallel platform. We have designed it to support three basic usage patterns. First is what we refer to as JIT (Just-In-Time) compilation. When the programmer invokes a @cu-decorated function either from the command line or from a Python code module, the Copperhead runtime may need to generate code for this procedure if none is already available. Second is batch compilation, where the Copperhead compiler is asked to generate a set of C++ code modules for the specified procedures. This code may be subsequently used either within the Copperhead Python environment or linked directly with external C++ applications. The third common scenario is one where the compiler is asked to generate a collection of variant instantiations of a Copperhead procedure in tandem with an autotuning framework for exploring the performance landscape of a particular architecture.

In the following discussion, we assume that the compiler is given a single top level Copperhead function—referred to as the "entry point"—to compile. It may, for instance, be a function like spmv_csr that has been invoked at the Python interpreter prompt. For the CUDA platform, the compiler will take this procedure, along with any procedures it invokes, and produce a single sequential *host* procedure and one or more parallel *kernels* that will be invoked by the host procedure.

4.1 Front End

After parsing the program text using the standard Python ast module, the compiler front end converts the program into a simplified abstract syntax tree (AST) form suitable for consumption by the rest of the compiler. It applies a sequence of fairly standard program transformations to: lift all nested lambdas and procedures into top level definitions, make closures explicit, convert all statements to single assignment form with no nested expressions, and infer static types for all elements of the program. Programs that contain syntax errors or that do not conform to the restricted language outlined in Section 3 are rejected.

For illustration, Figure 2 shows a program fragment representing the AST for the Copperhead procedure shown in Figure 1. Each procedure is annotated with its most general polymorphic type, which we determine using a standard Hindley-Milner style type inference process. Lexically captured variables within nested procedures are eliminated by converting free variables to explicit arguments—shown as variables _k0, _k1, etc.—and replacing the original nested definition point of the function with a special closure([k_1, ..., k_n], f) form where the k_i represent the originally free variables. In our example, the local variable x was free in the body of the nested spvv procedure, and is thus converted into an explicit closure argument in the transformed code. Single assignment conversion adds a unique subscript to each local identifier (e.g., z_0 in spvv), and flattening nested expressions introduces temporary identifiers (e.g., tmp_0) bound to the value of individual sub-expressions. Note that our single assignment form has no need of the ϕ-functions used in SSA representations of imperative languages since we disallow loops and every branch of a conditional must return a value.

4.2 Scheduling Nested Parallelism

The front end of the compiler carries out platform independent transformations in order to prepare the code for scheduling. The middle section of the compiler is tasked with performing analyses and scheduling the program onto a target platform.

At this point in the compiler, a Copperhead program consists of possibly nested compositions of data parallel primitives. A Copperhead procedure may perform purely sequential computations, in which case our compiler will convert it into sequential C++ code. Copperhead makes *no attempt* to auto-parallelize sequential codes. Instead, it requires the programmer to use primitives that the compiler will *auto-sequentialize* as necessary. Our compilation of sequential code is quite straightforward, since we rely on the host C++ compiler to handle all scalar code optimizations, and our restricted language avoids all the complexities of compiling a broad subset of Python that must be addressed by compilers like Cython [9].

Copperhead supports both flat and nested data parallelism. A flat Copperhead program consists of a sequence of parallel primitives which perform purely sequential operations to each element of a sequence in parallel. A nested program, in contrast, may apply parallel operations to each sequence element. Our spmv_csr code provides a simple concrete example. The outer spmv_csr procedure applies spvv to every row of its input via map. The spvv procedure itself calls gather, map, and sum, all of which are potentially parallel. The Copperhead compiler must decide how to map these potentially parallel operations onto the target hardware: for example, they may be implemented as parallel function calls or sequential loops.

One approach to scheduling nested parallel programs, adopted by NESL [4] and Data Parallel Haskell [7] among others, is to apply a flattening (or vectorization) transform to the program. This converts a nested structure of vector operations into a sequence of flat operations. In most cases, the process of flattening replaces the nested operations with segmented equivalents. For instance, in our spmv_csr example, the nested sum would be flattened into a segmented reduction.

The flattening transform is a powerful technique which ensures good load balancing. It also enables data parallel compilation for flat SIMD processors, where all lanes in the SIMD array work in

lockstep. However, we take a different approach. Flattening transformations are best suited to machines that are truly flat. Most modern machines, in contrast, are organized hierarchically. The CUDA programming model [23, 24], for instance, provides a hierarchy of execution formed into four levels:

1. A *host thread* running on the CPU, which can invoke
2. Parallel *kernels* running on the GPU, which consist of
3. A grid of *thread blocks* each of which is comprised of
4. Potentially hundreds of *device threads* running on the GPU.

The central goal of the Copperhead compiler is thus to map the nested structure of the program onto the hierarchical structure of the machine.

Recent experience with hand-written CUDA programs suggests that direct mapping of nested constructions onto this physical machine hierarchy often yields better performance. For instance, Bell and Garland [1] explored several strategies for implementing SpMV in CUDA. Their Coordinate (COO) kernel is implemented by manually applying the flattening transformation to the `spmv_csr` algorithm. Their other kernels represent static mappings of nested algorithms onto the hardware. The flattened COO kernel only delivers the highest performance in the exceptional case where the distribution of row lengths is extraordinarily variable, in which case the load balancing provided by the flattening transform is advantageous. However, for 13 out of the 14 unstructured matrices they examine, applying the flattening transform results in performance two to four times slower than the equivalent nested implementation.

Experiences such as this lead us to the conclusion that although the flattening transform can provide high performance in certain cases where the workload is extremely imbalanced, the decision to apply the transform should be under programmer control, given the substantial overhead the flattening transform imposes for most workloads.

Our compiler thus performs a static mapping of nested programs onto a parallelism hierarchy supported by the target parallel platform.

Returning to our `spmv_csr` example being compiled to the CUDA platform, the `map` within its body will become a CUDA kernel call. The compiler can then chose to map the operations within `spvv` to either individual threads or individual blocks. Which mapping is preferred is in general program and data dependent; matrices with very short rows are generally best mapped to threads while those with longer rows are better mapped to blocks.

Because this information is data dependent, we currently present the decision of which execution hierarchy to target as a compiler option. The programmer can arrange the code to specify explicitly which is preferred, an autotuning framework can explore which is better on a particular machine, or a reasonable default can be chosen based on any static knowledge about the problem being solved. In the absence of a stated preference, the default is to map nested operations to sequential implementations. This will give each row of the matrix to a single thread of the kernel created by the call to `map` in `spmv_csr`.

4.3 Synchronization Points and Fusion

The compiler must also discover and schedule synchronization points between aggregate operators. We assume that our target parallel platform provides a SPMD-like execution model, and therefore synchronization between a pair of parallel operations is performed via barrier synchronization across parallel threads. Similarly, a synchronization point between a pair of operators that have been marked as sequentialized implies that they must be implemented with separate loop bodies.

The most conservative policy would be to require a synchronization point following every parallel primitive. However, this can introduce far more synchronization, temporary data storage, and memory bandwidth consumption than necessary to respect data dependencies in the program. Requiring a synchronization point after every parallel primitive can reduce performance by an order of magnitude or more, compared with equivalent code that synchronizes only when necessary. In order to produce efficient code, our compiler must introduce as few synchronization points as possible, by fusing groups of parallel primitives into single kernels or loop bodies.

In order to determine where synchronization points are necessary, we need to characterize the data dependencies between individual elements of the sequences upon which the parallel primitives are operating. Consider an aggregate operator of the form `x = f(y`$_1$`, ..., y`$_n$`)` where `x` and the `y`$_i$ are all sequences. Since the elements of `x` may be computed concurrently, we need to know what values of `y`$_i$ it requires.

We call the value `x[i]` *complete* when its value has been determined and may be used as input to subsequent operations. The array `x` is complete when all its elements are complete.

We categorize the bulk of our operators into one of two classes. The first class are those operators which perform induction over the domain of their inputs. This class is essentially composed of `permute`, `scatter`, and their variants. We cannot in general guarantee that any particular `x[i]` is complete until the entire operation is finished. In other words, either the array `x` is entirely complete or entirely incomplete.

The second class are those operators which perform induction over the domain of their outputs. This includes almost all the rest of our operators, such as `map`, `gather`, and `scan`. Here we want to characterize the portion of each `y`$_j$ that must be complete in order to complete `x[i]`. We currently distinguish three broad cases:

- *None*: `x[i]` does not depend on any element of `y`$_j$
- *Local*: completing `x[i]` requires that `y`$_j$`[i]` be complete
- *Global*: completing `x[i]` requires that `y`$_j$ be entirely complete

With deeper semantic analysis, we could potentially uncover more information in the spectrum between local and global completion requirements. However, at the moment we restrict our attention to this simple classification.

Each primitive provided by Copperhead declares its completion requirements on its arguments (none, local, global) as well as the completion of its output. Completing the output of class 1 operators requires synchronization, while the output of class 2 operators is completed locally. The compiler then propagates this information through the body of user-defined procedures to compute their completion requirements. For example, a synchronization point is required between a class 1 primitive and any point at which its output is used, or before any primitive that requires one of its arguments to be globally complete. We call a sequence of primitives that are not separated by any synchronization points a *phase* of the program. All operations within a single phase may potentially be fused into a single parallel kernel or sequential loop by the compiler.

4.4 Shape Analysis

Shape analysis determines, where possible, the sizes of all intermediate values. This allows the back end of the compiler to statically allocate and reuse space for temporary values, which is critical to obtaining efficient performance. For the CUDA backend, forgoing shape analysis would require that every parallel primitive be executed individually, since allocating memory from within a CUDA kernel is not feasible. This would lead to orders of magnitude lower performance.

Our internal representation gives a unique name to every temporary value. We want to assign to each a *shape* of the form ⟨[$\mathtt{d}_1$, ..., $\mathtt{d}_n$], s⟩ where $\mathtt{d}_i$ is the array's extent in dimension i and s is the shape of each of its elements. Although Copperhead currently operates on one-dimensional sequences, the shape analysis we employ applies to higher dimensional arrays as well. The shape of a 4-element, 1-dimensional sequence [5,4,8,1] would, for example, be ⟨[4], Unit⟩ where Unit=⟨[], ·⟩ is the shape reserved for indivisible types such as scalars. Nested sequences are not required to have fully determined shapes: in the case where subsequences have differing lengths, the extent along that dimension will be undefined, for example: ⟨[2], ⟨[*], Unit⟩⟩. Note that the dimensionality of all values are known as a result of type inference. It remains only to determine extents, where possible.

We approach shape analysis of user-provided code as an abstract interpretation problem. We define a shape language consisting of Unit, the shape constructor ⟨D, s⟩ described above, identifiers, and shape functions extentof(s), elementof(s) that extract the two respective portions of a shape s. We implement a simple environment-based evaluator where every identifier is mapped to either (1) a shape term or (2) itself if its shape is unknown. Every primitive f is required to provide a function, that we denote f.shape, that returns the shape of its result given the shapes of its inputs. It may also return a set of static constraints on the shapes of its inputs to aid the compiler in its analysis. Examples of such shape-computing functions would be:

```
gather.shape(x, i) = ⟨extentof(i), elementof(x)⟩
zip.shape(x1, x2)  = ⟨extentof(x1),
                       (elementof(x1),elementof(x2))⟩
                        with extentof(x1)==extentof(x2)
```

The gather rule states that its result has the same size as the index array i while having elements whose shape is given by the elements of x. The zip augments the shape of its result with a constraint that the extents of its inputs are assumed to be the same. Terms such as extentof(x) are left unevaluated if the identifier x is not bound to any shape.

To give a sense of the shape analysis process, consider the spvv procedure shown in Figure 2. Shape analysis will annotate every binding with a shape like so:

```
def spvv(Ai, j, _k0):
    z0 :: ⟨extentof(j), elementof(_k0)⟩
    tmp0 :: ⟨extentof(Ai), elementof(Ai)⟩
    return sum(tmp0) :: Unit
```

In this case, the shapes for z_0, $\mathtt{tmp}_0$, and the return value are derived directly from the shape rules for gather, map, and sum, respectively.

Shape analysis is not guaranteed to find all shapes in a program. Some identifiers may have data dependent shapes, making the analysis inconclusive. For these cases, the compiler must treat computations without defined shapes as barriers, across which no fusion can be performed, and then generate code which computes the actual data dependent shape before the remainder of the computation is allowed to proceed.

Future work involves allowing shape analysis to influence code scheduling: in an attempt to better match the extents in a particular nested data parallel problem to the dimensions of parallelism supported by the platform being targeted. For example, instead of implementing the outermost level of a Copperhead program in a parallel fashion, if the outer extent is small and the inner extent is large, the compiler may decide to create a code variant which sequentializes the outermost level, and parallelizes an inner level.

```
struct lambda0 {
    template<typename T_Aij, typename T_xj >
    __device__ T_xj operator()(T_Aij Aij, T_xj xj)
        { return Aij * xj; }
};

struct spvv {
    template<typename T_Ai,
             typename T_j,
             typename T_k0 >
    __device__ typename T_Ai::value_type
    operator()(T_Ai Ai,
               T_j j,
               T_k0 _k0) {
        typedef typename T_Ai::value_type _a;
        gathered<...> z0 = gather(_k0, j);
        transformed<...> tmp0 =
            transform<_a>(lambda0(), Ai, z0);
        return sequential::sum(tmp0);
    }
};

template<typename T2>
struct spvv_closure1 {
    T2 k0;
    __device__ spvv_closure1(T2 _k0) : k0(_k0) { }

    template<typename T0, typename T1>
    __device__ typename T0::value_type
    operator()(T0 arg0, T1 arg1) {
        return spvv()(arg0, arg1, k0);
    }
};

template<typename _a > __global__
void spmv_csr_phase0(nested_sequence<_a, 1> vals,
                     nested_sequence<int,1> cols,
                     stored_sequence<_a> x,
                     stored_sequence<_a> _return) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    if( i < vals.size() )
        _return[i] = spvv_closure1<stored_sequence<_a> >
                        (x)(vals[i], cols[i]);
}
```

Figure 3. Sample CUDA C++ code generated for spmv_csr. Ellipses (...) indicate incidental type and argument information elided for brevity.

4.5 CUDA C++ Back End

The back end of the Copperhead compiler generates platform specific code. As mentioned, we currently have a single back end, which generates code for the CUDA platform. Figure 3 shows an example of such code for our example spmv_csr procedure. It consists of a sequence of function objects for the nested lambdas, closures, and procedures used within that procedure. The templated function spmv_csr_phase0 corresponds to the Copperhead entry point.

Not shown here is the Copperhead generated C++ host code that invokes the parallel kernels. It is the responsibility of the host code to marshal data where necessary, allocate any required temporary storage on the GPU, and make the necessary CUDA calls to launch the kernel.

We generate templated CUDA C++ code that makes use of a set of sequence types, such as stored_sequence and nested_sequence types, which hold sequences in memory. Fusion of sequential loops and block-wise primitives is performed through the construction of

compound types. For example, in the `spvv` functor shown above, the calls to `gather`, and `transform` perform no work, instead they construct `gathered_sequence` and `transformed_sequence` structures that lazily perform the appropriate computations upon dereferencing. Work is only performed by the last primitive in a set of fused sequential or block-wise primitives. In this example, the call to `seq::sum` introduces a sequential loop which then dereferences a compound sequence, at each element performing the appropriate computation. When compiling this code, the C++ compiler statically eliminates all the indirection present in this code, yielding machine code which is as efficient as if we had generated the fused loops directly.

We generate fairly high level C++ code, rather than assembly level code, for two reasons. Firstly, existing C++ compilers provide excellent support for translating well structured C++ to efficient machine code. Emitting C++ from our compiler enables our compiler to utilize the vast array of transformations which existing compilers already perform. But more importantly, it means that the code generated by our Copperhead compiler can be reused in external C++ programs. We believe that an important use case for systems like Copperhead is to prototype algorithms in a high-level language and then compile them into template libraries that can be used by a larger C++ application.

5. Runtime

Copperhead code is embedded in standard Python programs. Python function decorators indicate which procedures should be executed by the Copperhead runtime. When a Python program calls a Copperhead procedure, the Copperhead runtime intercepts the call, compiles the procedure, and then executes it on a specified execution place. The Copperhead runtime uses Python's introspection capabilities to gather all the source code pertaining to the procedure being compiled. This model is inspired by the ideas from Selective, Embedded, Just-In-Time Specialization [6].

The Copperhead compiler is fundamentally a static compiler that may be optionally invoked at runtime. Allowing the compiler to be invoked at runtime matches the no-compile mindset of typical Python development. However, the compiler does not perform dynamic compilation optimizations specific to the runtime instantiation of the program, such as treating inputs as constants, tracing execution through conditionals, etc. Forgoing these optimizations enables the results of the Copperhead compiler to be encapsulated as a standard, statically compiled binary, and cached for future reuse or incorporated as libraries into standalone programs which are not invoked through the Python interpreter.

Consequently, the runtime compilation overhead we incur is analogous to the build time of traditional static compilers, and does not present a performance limitation. The Copperhead compiler itself typically takes on the order of 300 milliseconds to compile our example programs, and the host C++ and CUDA compilers typically take on the order of 20 seconds to compile a program. Both of these overheads are not encountered in performance critical situations, since Copperhead's caches obviate recompilation. The actual runtime overhead, compared with calling an equivalent C++ function from within C++ code, is on the order of 100 microseconds per Copperhead procedure invocation.

For each Copperhead procedure, the CUDA runtime generates a host C++ function which coordinates the launch and synchronization between multiple parallel phases, as well as allocates data for temporary results. Copperhead uses PyCUDA [18] and CodePy [17] to provide mechanisms for compiling, persistent caching, linking and executing CUDA and C++ code. The Copperhead runtime uses Thrust [14] to implement fully parallel versions of certain data parallel primitives, such as `reduce` and variations of `scan`.

5.1 Places

In order to manage the location of data and kernel execution across multiple devices, the Copperhead runtime defines a set of *places* that represent these heterogeneous devices. Data objects are created at a specific place. Calling a Copperhead procedure will execute a computation on the current target place, which is controlled utilizing the Python `with` statement.

Currently we support two kinds of places: CUDA capable GPUs and the native Python interpreter. Copperhead is designed to allow other types of places, with corresponding compiler back ends to be added. For instance, multi-core x86 back end would be associated with a new place type.

To faciliate interoperability between Python and Copperhead, all data is duplicated, with a *local* copy in the Python interpreter, and a *remote* copy which resides at the place of execution. Data is lazily transferred between the local and remote place as needed by the program.

5.2 Data Structures

Copperhead adopts a common technique for representing arbitrarily nested sequences as a flat sequence of data, along with a descriptor sequence for each level of nesting. The descriptor sequences provide the necessary information to build a view of each subsequence, including empty subsequences.

In addition to supporting arbitrarily nested sequences, Copperhead also allows programmers to construct uniformly nested sequences, which support the important special case where the shape is completely defined. For such sequences, a set of strides and lengths are sufficient to describe the nesting structure - descriptor sequences are not required. Uniformly nested sequences also allow the data to be arbitrarily ordered: when the programmer creates a uniformly nested sequence, they either specify the data ordering or provide a tuple of strides directly. This allows the programmer to express data layouts analogous to row- and column-major ordering for doubly nested sequences, as well as their analogues for more deeply nested sequences. The programmer may provide the strides directly, which allows the subsequences to be aligned arbitrarily. This can provide important performance benefits when data access patterns with standard nested sequences are not matched well to the processor's memory hierarchy.

We have seen have seen two to three fold performance improvements by using the correct nested data structure type. Future work may investigate autotuning over alignments and data layouts.

6. Examples

In this section, we investigate the performance of Copperhead code in several example programs. As we compare performance, we compare Copperhead compiled code to published, well-optimized, hand-crafted CUDA implementations of the same computation. Our compiler can't apply all the transformations which a human can, so we don't expect to achieve the same performance as well-optimized code. Still, we aim to show that high-level data parallel computations can perform within striking distance of human optimized efficiency code.

The Copperhead compiler and runtime is freely available at `http://code.google.com/p/copperhead`.

6.1 Sparse Matrix Vector Multiplication

Continuing with the Sparse Matrix Vector Multiplication example, we examine the performance of Copperhead generated code for three different SpMV kernels: compressed sparse row, vector compressed sparse row, and ELL. The CSR kernel is generated by compiling the Copperhead procedure for CSR SpMV onto the standard parallelism hierarchy, which distributes computations along

the outermost parallelism dimension to independent threads. Data parallel operations are sequentialized into loops inside each thread. For the CSR kernel, each row of the matrix is then processed by a different thread, and consequently, adjacent threads are processing widely separated data. On the CUDA platform, this yields suboptimal performance for most datasets.

The vector CSR kernel improves on the performance of the CSR kernel by mapping to a different parallelism hierarchy: one where the outermost level distributes computations among independent thread blocks, and subsequent data parallel operations are then sequentialized to block-wise operations. As mentioned earlier, the Copperhead programmer can choose which hierarchy to map to, or can choose to autotune over hierarchies. In this case, mapping to the block-wise hierarchy improves memory performance on the CUDA platform, since adjacent threads inside the same thread block are processing adjacent data, allowing for vectorized memory accesses.

The ELL representation stores the nonzeros of the matrix in a dense M-by-K array, where K bounds the number of nonzeros per row, and rows with fewer than K nonzeros are padded. The array is stored in column-major order, so that after the computation has been mapped to the platform, adjacent threads will be accessing adjacent elements of the array. For example:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

```
vals = [[1,2,5,6],[7,8,3,4],[*,*,9,*]]
cols = [[0,1,0,1],[1,2,2,3],[*,*,3,*]]
```

ELL generally performs best on matrices where the variance of the number of non-zeros per row is small. If exceptional rows with large numbers of non-zeros exist, ELL will perform badly due to the overhead resulting from padding the smaller rows.

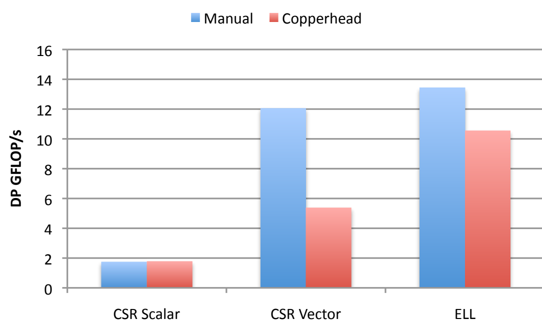

Figure 4. Average Double Precision Sparse Matrix Vector Multiply Performance

We compare against Cusp [1], a C++ library for Sparse Matrix Vector multiplication, running on an NVIDIA GeForce GTX 480 GPU. We use a suite of 8 unstructured matrices which were used by Bell and Garland [1], and which are amenable to ELL storage. Copperhead generated code achieves identical performance for the scalar CSR kernel, and on average provides 45% and 79% of Cusp performance for the vector CSR and ELL kernels, respectively.

Our relatively low performance on the vector CSR kernel is due to a specialized optimization which the Cusp vector CSR implementation takes advantage of, but the Copperhead compiler does not: namely the ability to perform "warp-wise" reductions without synchronization, packing multiple rows of the SpMV computation into a single thread block. This optimization is an important workaround for the limitations of today's CUDA processors, but we consider it too special purpose to implement in the Copperhead compiler.

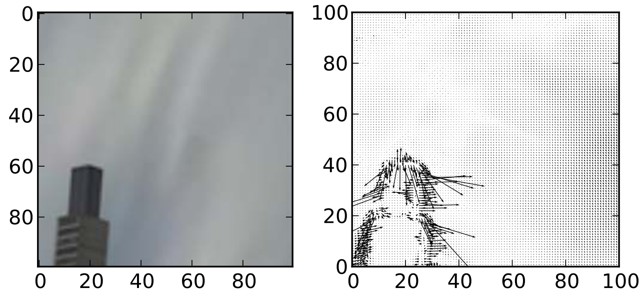

Figure 5. Left: closeup of video frame. Right: gradient vector field for optical flow

6.2 Preconditioned Conjugate Gradient Linear Solver

The Conjugate Gradient method is widely used to solve sparse systems of the form $Ax = b$. We examine performance on a preconditioned conjugate gradient solver written in Copperhead, which forms a part of an fixed-point non-linear solver used in Variational Optical Flow methods [27]. Figure 5 was generated using the `matplotlib` Python library, operating directly from the Copperhead solver, highlighting the ability of Copperhead programs to interoperate with other Python modules, such as those for plotting.

Conjugate gradient performance depends strongly on matrix-vector multiplication performance. Although we could have used a preexisting sparse-matrix library and representation, in this case we know some things about the structure of the matrix, which arises from a coupled 5-point stencil pattern on a vector field. Taking advantage of this structure, we can achieve significantly better performance than any library routine by creating a custom matrix format and sparse matrix-vector multiplication routine. This is an ideal scenario for Copperhead, since the productivity gains enable programmers to write custom routines that take advantage of special knowledge about a problem, as opposed to using a prebuilt library.

In addition to writing a custom sparse-matrix vector multiplication routine, practically solving this problem requires the use of a preconditioner, since without preconditioning, convergence is orders of magnitude slower. We utilize a block Jacobi preconditioner. Figure 6 shows the Copperhead code for computing the preconditioner, which involves inverting a set of symmetric 2x2 matrices, with one matrix for each point in the vector field, as well as applying the preconditioner, which involes a large number of symmetric 2x2 matrix multiplicaitons.

We implemented the entire solver in Copperhead. The custom SpMV routine for this matrix runs within 10% of the hand-coded CUDA version, achieving 49 SP GFLOP/s on a GTX 480, whereas a hand-tuned CUDA version achieves 55 SP GFLOP/s on the same hardware. Overall, for the complete preconditioned conjugate gradient solver, the Copperhead generated code yields 71% of the custom CUDA implementation.

6.3 Quadratic Programming: Nonlinear Support Vector Machine Training

Support Vector Machines are a widely used classification technique from machine learning. Support Vector Machines classify multi-dimensional data by checking where the data lies with respect to a decision surface. Nonlinear decision surfaces are learned via a Quadratic Programming optimization problem, which we implement in Copperhead.

We make use of the Sequential Minimal Optimization algorithm, with first-order variable selection heuristic [16], coupled with the RBF kernel, which is commonly used for nonlinear SVMs.

```
@cu
def vadd(x, y):
    return map(lambda a, b: return a + b, x, y)
@cu
def vmul(x, y):
    return map(lambda a, b: return a * b, x, y)
@cu
def form_preconditioner(a, b, c):
    def det_inverse(ai, bi, ci):
        return 1.0/(ai * ci - bi * bi)
    indets = map(det_inverse, a, b, c)
    p_a = vmul(indets, c)
    p_b = map(lambda a, b: -a * b, indets, b)
    p_c = vmul(indets, a)
    return p_a, p_b, p_c
@cu
def precondition(u, v, p_a, p_b, p_c):
    e = vadd(vmul(p_a, u), vmul(p_b, v))
    f = vadd(vmul(p_b, u), vmul(p_c, v))
    return e, f
```

Figure 6. Forming and applying the block Jacobi preconditioner

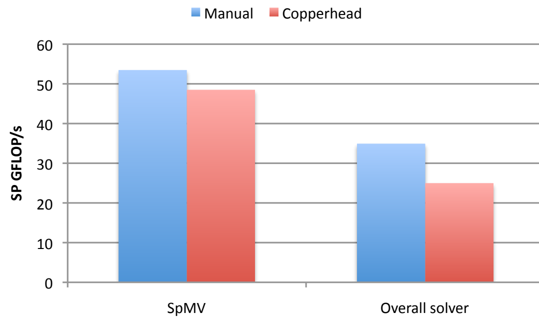

Figure 7. Performance on Preconditioned Conjugate Gradient Solver

Computationally, this is an iterative algorithm, where at each step the majority of the work is to evaluate distance functions between all data points and two active data points, update an auxiliary vector and then to select the extrema from data-dependent subsets of this vector. More details can be found in the literature [5]. Space does not permit us to include the Copperhead code that implements this solver, but we do wish to point out the RBF kernel evaluation code, shown in Figure 9, is used both on its own, where the compiler creates a version that instantiates the computation across the entire machine, as well as nested as an inner computation within a larger parallel context, where the compiler fuses it into a sequential loop.

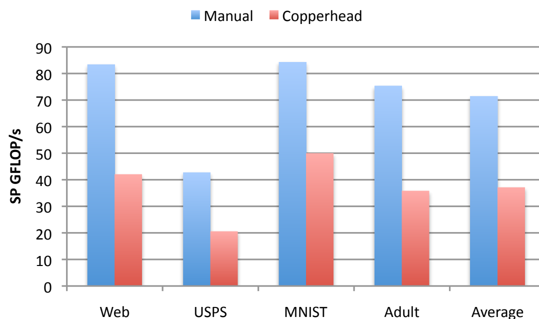

Figure 8. Support Vector Machine Training Performance

```
@cu
def norm2_diff(x, y):
    def el(xi, yi):
        diff = xi - yi
        return diff * diff
    return sum(map(el, x, y))
@cu
def rbf(gamma, x, y):
    return exp(-gamma * norm2_diff(x,y))
```

Figure 9. RBF Kernel evaluation

We compare performance against GPUSVM, a publically available CUDA library for SVM training. Figure 8 shows the throughput of the Copperhead implementation of SVM training over four datasets. On average, the Copperhead implementation attains 51% of hand-optimized, well tuned CUDA performance, which is quite good, given the Copperhead implementation's simplicity. The main advantage of the hand coded CUDA implementation on this example compared to the Copperhead compiled code is the use of on-chip memories to reduce memory traffic, and we expect that as the Copperhead compiler matures, we will be able to perform this optimization in Copperhead as well, thanks to our existing shape analysis, which can inform data placement.

6.4 Productivity

Productivity is difficult to measure, but as a rough approximation, Table 2 provides the number of lines of code needed to implement the core functionality of the examples we have put forth, in C++/CUDA as well as in Python/Copperhead. On average, the Copperhead programs take about 4 times fewer lines of code than their C++ equivalents, which suggests that Copperhead programming is indeed more productive than the manual alternatives.

Example	CUDA & C++	Copperhead & Python
Scalar CSR	16	6
Vector CSR	39	6
ELL	22	4
PCG Solver	172	79
SVM Training	429	111

Table 2. Number of Lines of Code for Example Programs

7. Future Work

There are many avenues for improving the Copperhead runtime and compiler. The first set involves improving the quality of code which the compiler emits. For example, the compiler currently does not reorder parallel primitives to increase fusion and reduce synchronization. The compiler also does not attempt to analyze data reuse and place small, yet heavily reused sequences in on-chip memories to reduce memory bandwidth, nor does it attempt to take advantage of parallelism in the dataflow graph or parallel primitives. Another avenue for future work involves broadening the scope of programs which can be compiled by the Copperhead compiler, such as supporting multi-dimensional arrays, adding new data parallel primitives, and supporting forms of recursion other than tail recursion.

We also intend to create other back-ends for the Copperhead compiler, in order to support other platforms besides CUDA. Some potential choices include support for multicore x86 processors and OpenCL platforms.

8. Conclusion

This paper has shown how to efficiently compile a nested data parallel language to modern parallel microprocessors. Instead of using flattening transforms by default, we take advantage of nested parallel hardware by mapping data parallel computations directly onto the hardware parallelism hierarchy. This mapping is enabled by synchronization and shape analyses, which which we introduce. Instead of creating a new data parallel language, we have embedded Copperhead into a widely used productivity language, which makes Copperhead programs look, feel and operate like Python programs, interoperating with the vast array of Python libraries, and lowering the amount of investment it takes for a programmer to learn data parallel programming.

We then demonstrated that our compiler generates efficient code, yielding from 45 to 100% of the performance of hand crafted, well-optimized CUDA code, but with much higher programmer productivity. On average, Copperhead programs require 3.6 times fewer lines of code than CUDA programs, comparing only the core computational portions.

Large communities of programmers choose to use productivity languages such as Python, because programmer productivity is often more important than attaining absolute maximum performance. We believe Copperhead occupies a worthwhile position in the tradeoff between productivity and performance, providing performance comparable to that of hand-optimized code, but with a fraction of the programmer effort.

9. Acknowledgments

Research partially supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding from U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. Bryan Catanzaro was supported by an NVIDIA Graduate Fellowship.

References

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proc. Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. ACM, 2009.

[2] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-02313-X. URL `http://www.cs.cmu.edu/~guyb/papers/Ble90.pdf`.

[3] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39 (3):85–97, 1996.

[4] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[5] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111, 2008.

[6] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, 2010.

[7] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel Haskell: a status report. In *Proc. 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18. ACM, 2007.

[8] Clyther. Clyther: Python language extension for opencl. http://clyther.sourceforge.net/, 2010.

[9] Cython. Cython: C-extensions for python. http://cython.org/, 2010.

[10] R. Garg and J. N. Amaral. Compiling python to a hybrid execution environment. In *GPGPU '10: Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 19–30. ACM, 2010.

[11] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical Report White Paper, Intel Corporation, 2007.

[12] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *GPGPU-2: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[13] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986. ISSN 0001-0782.

[14] J. Hoberock and N. Bell. Thrust: A parallel template library, 2009. URL `http://www.meganewtons.com/`. Version 1.2.

[15] P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs...an experiment in software prototyping productivity. Technical Report YALEU/DCS/RR-1049, Yale University Department of Computer Science, New Haven, CT, 1994.

[16] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Comput.*, 13(3):637–649, 2001. ISSN 0899-7667.

[17] A. Klöckner. Codepy. `http://mathema.tician.de/software/codepy`, 2010.

[18] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda: Gpu run-time code generation for high-performance computing. *CoRR*, abs/0911.3456, 2009.

[19] S. Lee, M. M. T. Chakravarty, V. Grover, and G. Keller. GPU kernels as data-parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM 2009)*, 2009.

[20] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110. ACM, 2009.

[21] M. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proc. GSPx Multicore Applications Conference*, Nov. 2006.

[22] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proc. Graphics Hardware 2002*, pages 57–68. Eurographics Association, 2002.

[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar/Apr 2008.

[24] *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, May 2010. URL `http://www.nvidia.com/CUDA`. Version 3.1.

[25] L. Prechelt. Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. In *Advances in Computers*, volume 57, pages 205 – 270. Elsevier, 2003.

[26] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, July 2010. ISBN 978-0-13-138768-3.

[27] N. Sundaram, T. Brox, and K. Keutzer. Dense Point Trajectories by GPU-accelerated Large Displacement Optical Flow. In *European Conference on Computer Vision*, pages 438–445, September 2010.

[28] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGARCH Comput. Archit. News*, 34(5):325–335, 2006.

[29] Theano. Theano: A python array expression library. http://deeplearning.net/software/theano/, 2010.

[30] M. Wolfe. Implementing the PGI accelerator model. In *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.