Portable Performance on Heterogeneous Architectures

Phitchaya Mangpo Phothilimthana Jason Ansel Jonathan Ragan-Kelley Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{mangpo, jansel, jrk, saman}@csail.mit.edu

Abstract

Trends in both consumer and high performance computing are bringing not only more cores, but also increased heterogeneity among the computational resources within a single machine. In many machines, one of the greatest computational resources is now their graphics coprocessors (GPUs), not just their primary CPUs. But GPU programming and memory models differ dramatically from conventional CPUs, and the relative performance characteristics of the different processors vary widely between machines. Different processors within a system often perform best with different algorithms and memory usage patterns, and achieving the best overall performance may require mapping portions of programs across all types of resources in the machine.

To address the problem of efficiently programming machines with increasingly heterogeneous computational resources, we propose a programming model in which the best mapping of programs to processors and memories is determined empirically. Programs define *choices* in how their individual algorithms may work, and the compiler generates further choices in how they can map to CPU and GPU processors and memory systems. These choices are given to an empirical autotuning framework that allows the space of possible implementations to be searched at installation time. The rich choice space allows the autotuner to construct polyalgorithms that combine many different algorithmic techniques, using both the CPU and the GPU, to obtain better performance than any one technique alone. Experimental results show that algorithmic changes, and the varied use of both CPUs and GPUs, are necessary to obtain up to a 16.5x speedup over using a single program configuration for all architectures.

Categories and Subject Descriptors I.2.5 [*Artificial Intelligence*]: Programming Languages and Software; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Experimentation, Languages, Performance

Keywords Autotuning, Compilers, GPGPU, Heterogeneous

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

1. Introduction

The past decade has seen an explosion in processor parallelism, with the move to multicores. The next major architectural trend is the move from symmetric multiprocessors to *heterogeneous* systems, with multiple *different* processors and memories.

A major source of heterogeneity today is the widespread availability graphics coprocessors (GPUs), which can now be used for general purpose computation. GPUs can offer large speedups, thanks to high processor density and high memory bandwidth. Conventional wisdom often holds that, if a program can be mapped into GPU execution, it should be, since it will benefit from the immense computational resources. Instead, our results show that the choices in mapping to heterogeneous machines are much more complex.

A major challenge in utilizing heterogeneous resources is the diversity of devices on different machines, which provide widely varying performance characteristics. A program optimized for one coprocessor may not run as well on the next generation of processors or on a device from a different vendor, and a program optimized for GPU execution is often very different from one optimized for CPU execution. The relative performance of the CPUs and GPUs also varies between machines. On one machine, a specific portion of a computation may run best on the CPU, while on another machine it may run best on the GPU. There is also a scheduling problem: even if a specific kernel may perform better on the GPU, if the GPU is overloaded and the CPU is idle, it may be best to balance the workload between the two, or it may be best to place computation somewhere it runs more slowly but nearer to where its output will be used.

For many applications, mapping across all of a machine's resources—in different ways per-machine, and per-program—is essential to achieving high performance. In our SVD benchmark, the best performing programs used both the CPU and GPU concurrently for a significant subset of total work on some machines but not on the others.

Researchers are proposing even more asymmetric heterogeneous architectures with varied types of cores on a single chip [16]. But even on symmetric multicore systems without a GPU, the model of a multicore as simply a collection of identical independent cores does not hold in practice. Many modern processors dynamically scale up the frequency of a core when the cores around it are idle and share different amounts of resources between pairs of cores, making the performance of one core highly dependent on the workload of its neighbors [7]. While these techniques improve performance for many workloads, it implies that naively allocating a program more threads is not the optimal strategy for every algorithm on every system.

The traditional goal of a compiler is to map a program to a specific piece of hardware. Most compiler optimizations reason about program transformations using a simplified model of a spe-

cific processor target. This simple model of the performance of the machine is often implied by the heuristics that guide optimization. In heterogeneous machines, there are now multiple processors, with wildly different features and performance characteristics. It is extremely difficult for a single, comprehensible model to capture behavior across many processors and memory subsystems on which code may run, in order to reason about mapping a program to a heterogeneous machine. Compiling a program for a heterogeneous machine requires complex, interdependent choices of *algorithm:* which algorithm to use; *placement:* on which resources to place computations and data; and *mapping:* how much parallelism to exploit and how to use specialized memories.

We present a solution to the problem of efficiently programming diverse heterogeneous systems based on empirical autotuning. At installation time, a search space of possible program implementations is explored using an evolutionary autotuner that tests performance on representative inputs. This search space includes not only when to use different coprocessors, cores, and memory subsystems, but also how to use them. The autotuner is able to switch between entirely different algorithmic techniques encoded in the program, and between multiple strategies for mapping them to heterogeneous computational resources, and to combine the different techniques encoded in the program into a large search space of poly-algorithms.

Our work extends the PetaBricks language and compiler [3], which allows the programmer to specify algorithmic choices at the language level. Using this mechanism, PetaBricks programs define not only a single algorithm, but a search space of possible algorithms. The extensions to PetaBricks presented in this paper include:

- Compiler passes and static analyses to automatically convert subsets of existing PetaBricks programs into OpenCL kernels that can be run on a variety of architectural backends, and coexist as choices available to the autotuner. This includes static analyses to help efficiently manage the memory of these coprocessing devices, and reduce data movement between kernel executions.
- A GPU management runtime that allows coprocessor devices to be efficiently utilized in the PetaBricks workstealing runtime without requiring the calling CPU thread to block on GPU operations, providing automatic memory management, and allowing efficient overlapping of computation and communication.
- Autotuning enhancements to search different divisions of GPU and CPU work, and to allow the exploration of choices on the GPU, such as choices of in which of the many memory spaces to place different data, and choices of how much data to run on the GPU.

We show experimentally how different heterogeneous systems require different algorithmic choices for optimal performance. We show that, even when migrating between two architectures with modern GPUs, one can obtain up to a 16.5x speedup by retraining for each architecture. We show how diverse architectures perform best with differing divisions of work between the GPU and the CPU on each machine. The different configurations found are often non-obvious, and suggest that empirical tuning techniques are required to get the best performance on modern machines.

1.1 Contributions

We believe this is the first system which automatically determines the best mapping of programs in a high level language across a heterogeneous mix of parallel processing units, including placement of computation, choice of algorithm, and optimization for specialized memory hierarchies. With this, a high-level, architecture independent language can obtain comparable performance to architecture specific, hand-coded programs.

To realize this system,

- we integrate choices of devices, memories, and execution parameters into the PetaBricks autotuning compiler;
- we automate both the OpenCL code generation and selection process for a traditionally hand-written scratchpad memory optimization that requires significant memory access rewriting and the generation of multi-phase cooperative loads and stores;
- we introduce a runtime system that incorporates the use of a GPU coprocessor into a work-stealing system without the need to block on copies and GPU operations;
- we introduce a compiler analysis to identify when to use no copy, lazy copy, and eager copy memory management techniques.

This paper focuses not on the impact of individual optimizations, but on how to integrate many techniques together to automate the process of generating code and to achieve good performance on any heterogeneous machine.

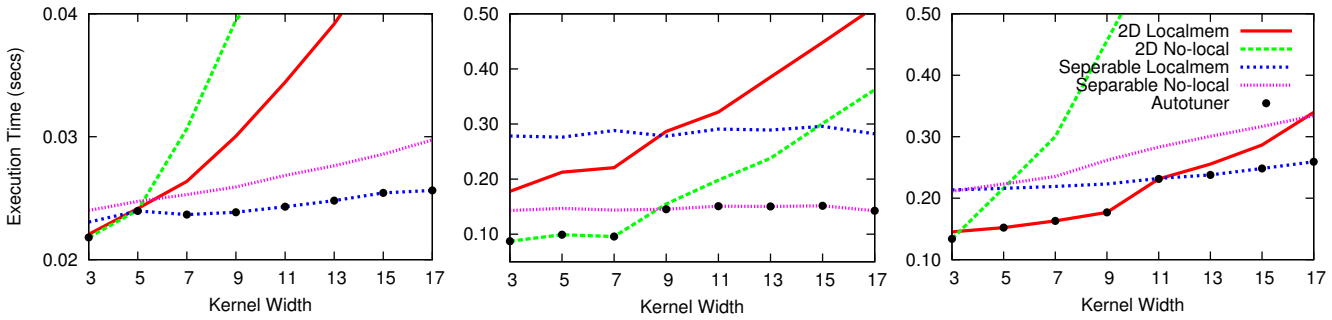
In evaluating our system across a range of applications and architectures, we show that algorithmic choices are required to get optimal performance across machines with different processor mixes, and the optimal algorithm is different in different architectures. Unlike prior work, we add the additional axis of algorithmic choice and global learning to our search space. We paint a complex picture of the search space where the best device(s) to use depends on the algorithm being run, the context it is being called from, and the underlying hardware being used. We show that the choice of how and when to use the GPU is complex with no simple answer. The best mapping of a program to devices depends not only on the execution times of different pieces of the program but also data location and existing tasks in the queues. Finally, we demonstrate that splitting work to run concurrently both on the GPU and the CPU can significantly outperform using either processor alone.

2. PetaBricks Language Background

The PetaBricks language provides a framework for the programmer to describe multiple ways of solving a problem while allowing the autotuner to determine which of those ways is best for the user's situation [3]. It provides both algorithmic flexibility (multiple algorithmic choices) as well as coarse-grained code generation flexibility (synthesized outer control flow).

At the highest level, the programmer can specify a *transform*, which takes some number of inputs (the `from` keyword) and produces some number of outputs (the `to` keyword). In this respect, the PetaBricks transform is like a function call in a procedural language. The major difference is that it allows the programmer to specify multiple pathways to convert the inputs to the outputs for each transform. Pathways are specified in a dataflow manner using a number of smaller building blocks called *rules*, which encode both the data dependencies of the rule and `C++`-like code that converts the rule's inputs to outputs.

Dependencies are specified by naming the inputs and outputs of each rule, but unlike in a traditionally dataflow programming model, more than one rule can be defined to output the same data. Thus, the input dependences of a rule can be satisfied by the output of one or more rules. The PetaBricks compiler and autotuner decide which rules to use to satisfy such dependencies by determining which are most computationally efficient for a given architecture and input. For example, on architectures with multiple processors, the autotuner may find that it is preferable to use rules that minimize



(a) Desktop (b) Server (c) Laptop

Figure 2. Execution time (lower is better) of different possible mappings of `SeparableConvolution` to OpenCL with varying kernel widths on three test systems when the input size is 3520×3520. The three test systems are described in Section 6.1. Our autotuner (described in Section 5) always discovers the best configuration for each system and width. Note that each mapping is optimal for at least one machine and kernel width. In handwritten OpenCL, these would correspond to four distinct programs the programmer would need to write.

the critical path of the transform, while on sequential architectures, rules with the lowest computational complexity may be a better choice.

2.1 Convolution in PetaBricks

Figure 1 shows `SeparableConvolution`, an example PetaBricks program. This program computes the convolution of a 2D matrix with a separable 2D kernel. The main transform (starting on line 1) maps from input matrix `In` to output matrix `Out` by convolving the input with the given `Kernel`. It does this using one of two rules:

- *Choice 1* maps from `In` to `Out` in a single pass, by directly applying the `Convolve2D` transform.
- *Choice 2* maps from `In` to `Out` in two passes, by first performing the `ConvolveRows` transform, storing its result in `buffer`, and then performing the `ConvolveColumns` transform on this intermediate result.

The three `Convolve*` transforms are each defined in terms of a single data parallel rule which, for each point in `Out`, computes a sum of the points in the corresponding `KWIDTH`-sized region of `In` weighted by the corresponding points in the `Kernel`. `ConvolveRows` and `ConvolveColumns` apply only in the horizontal and vertical directions, respectively, while `Convolve2D` applies both dimensions simultaneously, iterating over a 2D window of `In` for each output point. Based on the experimental data, the autotuner chooses which rules to run (a choice of algorithm), and how to map them onto the machine, including runtime scheduling strategy, parallelism, data placement, and the choice of processors on which to run.

2.2 The Choice Space for SeparableConvolution

The top-level `SeparableConvolution` transform compiles into two simple rules, each of which computes the entire `Out` matrix from the entire `In` matrix. Internally, these rules apply the `Convolve*` transforms which are defined elementwise, without sequential dependencies, and so can be executed in a data parallel fashion over the entire output space. As one choice, these data parallel rules can be compiled directly into equivalent OpenCL kernels, with each *work-item* computing a single cell of `Out`.

Because of its simple data parallel pattern, this example runs most efficiently when all `Convolve*` transform are executed entirely in the OpenCL runtime, but the ideal choices of *algorithm* and *mapping to the memory system* vary across machines, and for each machine across different kernel sizes. However, even though not optimal for this benchmark, the choice to run some fraction of the computation on the CPU is also available to the autotuner.

Intuitively, when blur size (kernel's width) is large, implementing the separable convolution as a pair of 1D passes over the data, first in the horizontal and then in the vertical direction, performs asymptotically less work than a single 2D convolution pass. Starting each block of work with an extra phase which prefetches a block of `In` into scratchpad memory shared by all processors in the block saves memory bandwidth compared to separately loading many overlapping inputs through the (slower) main memory hierarchy. But when the kernel size is small, the overhead (in bandwidth for intermediate results, and in kernel invocations) of multiple passes overtakes the computational complexity advantage of separable computation, making a single 2D pass faster, and the overhead of explicit prefetching into OpenCL local memory cannot be offset by the smaller number of redundant loads saved across a block of work. Where these tradeoffs dominate varies between machines. And the underlying machines vary further: on an OpenCL target where the shared memory maps to the same caches and buses used for all loads and stores, the explicit prefetch phase nearly always represents wasted work.

Figure 2 shows the running times of four distinct OpenCL choices (2D Convolution and Separable Convolution, each with and without local memory prefetching) generated by PetaBricks for `SeparableConvolution`, on three different machines (described in Section 6.1), over kernel sizes from 3-17. The ideal choice (the lowest line at a given point) varies among the four choices, depending on both the targeted machine and the kernel size. As expected, the execution times of the single-pass 2D algorithms increase faster than those of the two-pass separable algorithms as the kernel size increases, and the use of scratchpad memory helps little when the kernel size is small and worsens the performance on the CPU OpenCL runtime. But how exactly these effects interact, and the relative behavior of each choice, varies substantially across machines.

In short, simply mapping this program to the OpenCL runtime is not sufficient. *How* it is mapped (whether or not to prefetch shared inputs into scratchpad memory), and *which algorithms* to use, is a complex choice, highly dependent on the specific workload and target architecture.

The choices we make in automatically mapping to OpenCL are described further in the next section, while more detailed empirical evaluation and discussion of the best choices for this and other benchmarks on a diverse set of heterogeneous machines is given in Section 6.

```
transform SeparableConvolution
from In[w, h], Kernel[KWIDTH]
to Out[w−KWIDTH+1, h−KWIDTH+1]
{
  // Choice 1: single pass 2D convolution
  to(Out out) from(In in, Kernel kernel) {
    Convolve2D(out, in, kernel);
  }

  // Choice 2: two−pass separable convolution
  to(Out out) from(In in, Kernel kernel)
  using(buffer[w−KWIDTH+1, h]) {
    ConvolveRows(buffer, in, kernel);
    ConvolveColumns(out, buffer, kernel);
  }
}

transform Convolve2D
from In[w, h], Kernel[KWIDTH]
to Out[w−KWIDTH+1, h−KWIDTH+1]
{
  Out.cell(x,y)
  from(In.region(x, y, x+KWIDTH+1, y+KWIDTH+1) in, Kernel kernel)
  {
    ElementT sum = 0;
    for(int x = 0; x < KWIDTH; x++)
      for(int y = 0; y < KWIDTH; y++)
        sum += in.cell(x,y) * kernel.cell(x) * kernel.cell(y);
    return sum;
  }
}

transform ConvolveRows
from In[w, h], Kernel[KWIDTH]
to Out[w−KWIDTH+1, h]
{
  Out.cell(x,y)
  from(In.region(x, y, x+KWIDTH, y+1) in, Kernel kernel)
  {
    ElementT sum = 0;
    for(int i = 0; i < KWIDTH; i++)
      sum += in.cell(i,0) * kernel.cell(i);
    return sum;
  }
}

transform ConvolveColumns
from In[w, h], Kernel[KWIDTH]
to Out[w, h−KWIDTH+1]
{
  Out.cell(x,y)
  from(In.region(x, y, x+1, y+KWIDTH) in, Kernel kernel)
  {
    ElementT sum = 0;
    for(int i = 0; i < KWIDTH; i++)
      sum += in.cell(0,i) * kernel.cell(i);
    return sum;
  }
}
```

Figure 1. PetaBricks code for `SeparableConvolution`. The top-level transform computes `Out` from `In` and `Kernel` by *either* computing a single-pass 2D convolution, *or* computing two separate 1D convolutions and storing intermediate results in `buffer`. Our compiler maps this program to multiple different executables which perform different *algorithms* (separable vs. 2D blur), and map to the GPU in different ways.

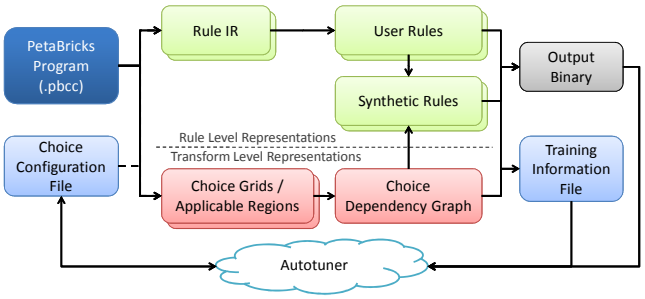


Figure 3. Flow for the compilation of a PetaBricks program with a *single* transform. (Additional transforms would cause the center part of the diagram to be duplicated.)

3. Compiling PetaBricks for Heterogeneous Machines

Figure 3 displays the general flow for the compilation of a PetaBricks transform. Compilation is split into two representations. The first representation operates at the rule level, and is similar to a traditional high level sequential intermediate representation. The second representation operates at the transform level, and is responsible for managing choices and for code synthesis.

The main transform level representation is the *choice dependency graph*, which is the primary way that choices are represented in PetaBricks. At a high level, the information contained in the choice dependency graph is similar to a familiar data dependency graph, however, the data is represented as an "inverse" of that graph: data dependencies are represented by vertices, while rules are represented by graph hyperedges. Additionally, data may be split into multiple vertices in the choice dependency graph if the transform contains rules that operate on just subregions of that data. The PetaBricks compiler uses this graph to manage code choices and to synthesize the outer control flow of the rules.

The final phase of compilation generates an *output binary* and a *training information file* containing static analysis information. These two outputs are used by the autotuner (described in Section 5), to search the space of possible algorithmic choices and other parameters. Autotuning creates a *choice configuration file*, which can either be used by the output binary to run directly or can be fed back into the compiler to allow additional optimizations.

3.1 OpenCL Kernel Generation

Since the PetaBricks language is more general (and supports calling arbitrary C/C++ code), only a subset of a PetaBricks program can be converted into OpenCL kernels. The process of generating additional choices for OpenCL execution is done early in the compilation process, before scheduling. The conversion process consists of three phases that are applied to each original user defined rule to inject equivalent synthetic OpenCL rules when possible.

In the first phase, a dependency analysis is performed to determine if the execution pattern of the rule fits into the OpenCL execution model. Sequential dependency patterns and data parallel dependency patterns can both be mapped to OpenCL kernels, but more complex parallel patterns, such as wavefront parallelism, can not be in our current implementation. It is possible that some sets of algorithmic choices will permit a mapping while others will not. The choice dependency graph is analyzed to determine if a rule can be mapped to OpenCL. The analysis looks at direction of the computed dependency for the strongly connected component associated with each of the rule's outputs. If there is no dependency, or the dependency is eliminated by selecting the rule choice under

consideration, then the outer dependency pattern of the rule can be mapped to OpenCL.

If a rule passes the first phase, it goes to the second phase where the body of the transform is converted into OpenCL code. This phase includes a number of syntactic conversions. It is also responsible for rewriting data accesses to use GPU global memory, detecting a number of language constructs, such as calls to external libraries, that can not be mapped to OpenCL, and disqualifying a rule from being converted to OpenCL. This phase can also detect some language constructs (such as inline native code), which can not be supported by OpenCL. The majority of these constructs are detected statically; however, there are a few more subtle requirements, sometimes OpenCL-implementation specific, which we detect by attempting to compile the resulting transform and rejecting synthetic OpenCL rules that do not compile.

The third phase attempts to optimize the *basic* version of OpenCL codes generated from the second phase by utilizing GPU local memory, known as OpenCL local memory or CUDA shared memory. For a subset of the mapped OpenCL rules, the *local memory* version can be generated. To do so, we analyze input data access pattern. A *bounding box* is a rectangular region of an input matrix that is used for computing an entry of the output matrix. If the size of the bounding box is a constant greater than one, then the local memory version of the GPU code is created; if the size of the bounding box is one, there is no need to copy the data into local memory because threads that share the same local memory never access the same data. The local memory version consists of two parts. First, all work-items on the GPU cooperate to load the data into local memory that will be accessed by the work-group they belong to. The second part is the actual computation derived from the basic version by replacing global memory accesses with local memory accesses. This local memory version is presented as a choice to the autotuner.

3.2 Data Movement Analysis

A second group of OpenCL analyses are performed at scheduling time in the compiler for determining how data is copied in and out of GPU memory. The PetaBricks compiler generates a unique schedule for each assignment of choices in a transform. These schedules represent parallelism as dependency trees that are given to the workstealing runtime. After each schedule is generated, it is analyzed for GPU data movement requirements. This analysis looks at preceding rule applications in the schedule and classifies applicable output regions generated on the GPU into three states:

- *must copy-out* regions are those that are immediately followed by a rule that executes on the CPU in the current schedule. For these regions, data is copied eagerly.
- *reused* regions are those immediately followed by another rule on the GPU. For these regions the data is left in GPU memory between rule applications.
- *may copy-out* are regions followed by dynamic control flow which we cannot analyze statically. For these regions, we employ a lazy copy-out strategy. A check is inserted before any code that may consume one of these regions to ensure that the required data is in CPU memory. If it is not, then the copy-out is performed when the data is requested.

Depending on the result of this analysis, tasks to prepare, copy-in, execute, and check copy-out status are inserted into the schedule by the compiler.

4. Runtime System

This section describes the runtime tasks, the *workstealing* model for CPU tasks, the *work-pushing* model for GPU tasks, and the integration of the two models. Figure 4 presents an overview of the different parts of the system that will be described in this section.

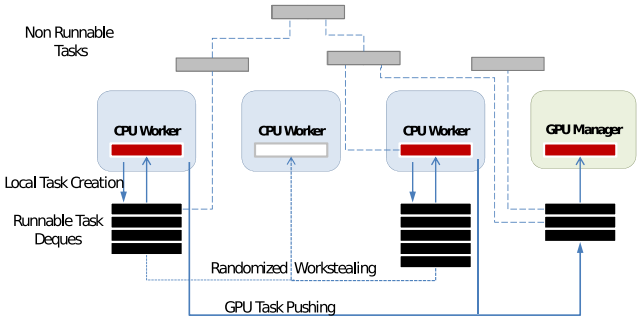


Figure 4. Overview of the runtime system. Worker threads use workstealing among themselves, and the GPU management thread only receives works that are pushed by the workers. Tasks exist in both a *runnable* and *non-runnable* state. Runnable tasks exist in deques of different threads, while non-runnable tasks are accessible only through dependency pointers in other tasks.

4.1 PetaBricks Workstealing Task Model

The PetaBricks runtime uses a workstealing mechanism similar to the one introduced in Cilk [11]. Instead of a stack, each thread has a THE-protocol [11] deque of tasks. Each thread continually pops tasks off the *top* of its deque and executes them, when tasks are created they are pushed onto the *top* of the local deque of the thread creating the task. When a thread is out of work, it choses a random victim thread and attempts to steal work off the *bottom* of that thread's deque. When starting a transform, much of the scheduling work is deferred in a continuation task that can be stolen by another thread.

Unlike Cilk, the PetaBricks task model supports arbitrary (non-cyclic) dependency graphs between tasks. To support this, dynamic dependency pointers are maintained from each incomplete task to those tasks that depend on it. When a task completes, it may return a continuation task to which all of its dependents are forwarded. Since task dependencies for dependent tasks are created in parallel to a task being executed, some care must be taken when managing tasks. Each task has a state, a count of dependencies, and a list of pointers to dependent tasks. A task can be in one of 5 states:

- *new task* is the initial state for a task. Dependencies may only be added to a task while it is in the new state, and those tasks that the new task depends on have to be tasks not already in the *complete* state. Adding a dependency to a task atomically increments the task's dependency count and adds the task to the appropriate dependents lists, following any needed continuation pointers. When dependency creation is finished, the task transitions to a *runnable task* if its dependency count is zero, otherwise to a *non-runnable task*.
- *non-runnable task* is the state for tasks whose dependency count is greater than zero. These tasks are waiting for other tasks to complete. Non-runnable tasks are stored only in the dependents lists of other tasks and are not stored in any central or thread-specific location. The task completion that eventually decrements the dependency count from one to zero is responsible for enqueuing the task in its thread-local deque.
- *runnable task* is the state for tasks that have zero dependencies and can be executed. These tasks are either being executed or are in exactly one thread-local deque of tasks. Runable tasks can be stolen. When executed, if the task returns a continuation task,

it transitions into the *continued* state, otherwise, it transitions to the *complete* state.

- *complete task* is the state for tasks that have already been executed and did not result in a continuation task. When a task becomes complete, it decrements the dependency count of each of its dependents and enqueues any dependent that becomes runnable. The dependents list is then cleared. Any subsequent attempt to depend on this task results in a no-op.
- *continued task* is the state for tasks that have been executed and returned a continuation. When a task enters the continued state a pointer is stored to the continuation task and the dependents list is transferred to the continuation task. Subsequent attempts to depend on this task instead depend on the continuation task (and possibly, recursively, a continuation of the continuation task).

4.2 GPU Management Thread and GPU Tasks

In this highly decentralized model, managing a GPU accelerator presents some challenges. First, for performance, it is important to overlap data transfer operations with computation. Second, one does not want to have many worker threads blocking and waiting for GPU operations to complete.

To address these challenges, we add a dedicated GPU management thread that is responsible for keeping track of all data that resides in GPU memory and schedule operations on the GPU. It operates using the same task representation as CPU threads', allowing dependencies between GPU and CPU tasks. A tasks is marked as either GPU or CPU task. We use workstealing scheme to manage CPU tasks, but work-pushing scheme to handle GPU tasks. CPU worker threads' deques can only contain CPU tasks, and the GPU management thread's FIFO queue can only contain GPU tasks.

Figure 5 depicts what happens when a task becomes runnable in different scenarios. We define the term *cause* as follow. Task A *cause*s task B to become runnable when task A satisfies the last dependency of task B; more precisely, task A decrements the dependency count of task B to zero. When a GPU task becomes runnable, it is pushed to the bottom of the GPU management thread's queue as shown in Figure 5(a). When a GPU task causes a CPU task to become runnable, the GPU management thread chooses a random CPU worker and pushes the task to the bottom of that thread's deque as shown in Figure 5(b). When a CPU task causes another CPU task to become runnable, the CPU worker that runs the former task pushes the newly runnable task to the top of its own local deque as shown in Figure 5(c).

There are four classes of GPU tasks that are run by the GPU management thread. For each execution of a GPU kernel, there are one *prepare* task, zero or more *copy-in* tasks, one *execute* task, and zero or more *copy-out completion* tasks.

- *Prepare* tasks allocate buffers on the GPU, and update metadata for GPU execution.
- *Copy-in* tasks copy the required input data to the GPU. One copy-in task is responsible for one input. This task performs a non-blocking write to a GPU buffer and becomes a *complete* immediately after the call, so the GPU manager thread does not have to wait and can execute the next task in its queue right away.
- *Execute* tasks initiate the asynchronous execution of the kernel, perform non-blocking reads from GPU buffers to must copy-out regions, and put may copy-out regions into *pending storage*.
- *Copy-out completion* tasks check the status of the non-blocking reads called by the execute task. If the status of a read is complete, the copy-out completion task corresponding to that output data transitions to the *complete* state. If the read is not complete, the GPU manager thread pushes the task back to the end of its queue.

There is no dependency between GPU tasks because the GPU management thread only runs one task at a time; the GPU tasks associated to one GPU kernel execution only need to be enqueued by following order: prepare, copy-in, execute, and copy-out completion, to ensure the correctness. However, CPU tasks may depend on GPU copy-out completion tasks.

4.3 Memory Management

GPU memory is allocated and managed by the GPU management thread. The GPU management thread keeps a table of information about data stored in the GPU. Each region stored on the GPU can either be a copy of a region of a matrix in main memory or an output buffer for newly computed data that must eventually be copied back to main memory. The GPU management thread is responsible for releasing buffers that become stale because the copy in main memory has been written to and for copying data back to main memory when the data is needed or when it has been flagged for eager copy-out.

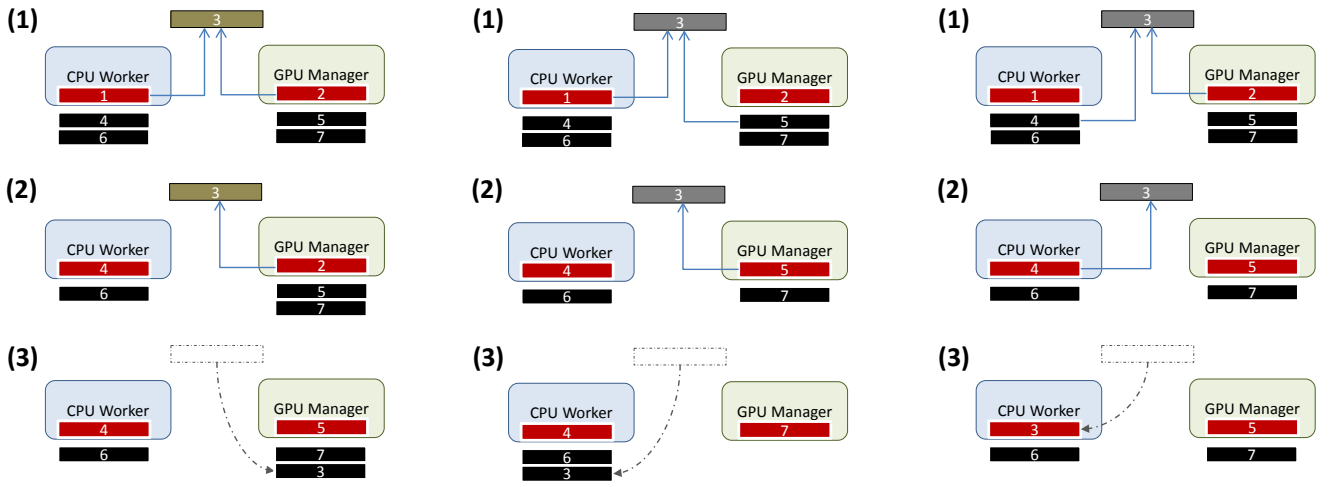
The memory management process includes various optimizations to minimize data transfer between the GPU and the CPU. Before we further explain our implementation, the term *matrix* and *region* should be clarified. A matrix is an input or an output of a transform, and is an n-dimensional dense array of elements. A region is a part of a matrix, defined by a start coordinate and size that is an input or an output of a rule.

Copy-in Management Before the GPU management thread executes a *copy-in* task, it will check whether the data is already on the GPU. It can already be on the GPU either if it was copied in for another task execution or if it was the output of another task on the GPU. If all data that will be copied in by the task is already on the GPU, then the GPU management thread will change the status of that *copy-in* task to *complete* without actually executing the task, otherwise it will perform the required copy.

Copy-out Management A single output matrix might be generated by multiple rules in a transform. For example, one task might generate the interior elements of a matrix and other tasks will generate the edges and corners. Instead of creating multiple small buffers on the GPU for these multiple rule outputs, it is often more efficient to create one big buffer for the entire matrix and have the individual tasks output to regions of this buffer. Creating one buffer will take less time than creating multiple buffers because there is an overhead associated with each read from GPU memory, and copying out multiple regions individually requires extra copying to consolidate the data back to the original buffer. It is also possible that some regions of a matrix will be generated on the CPU while others will be generated on the GPU. Our code generation and memory management needs to handle these more complex cases to keep track of which regions of a matrix have been written.

In order to apply this optimization, we use a static analysis for each possible schedule to determine which regions of the matrix are generated on the GPU. The GPU management thread will wait until all of the individual regions have been computed to change the state of the larger matrix.

CPU-GPU Work Balancing If a rule can be run on GPU, choices of how much data should be computed on the GPU are presented to our autotuner. At the extremes, the entire rule could computed on the GPU, or the entire rule could be computed on the CPU. We implement the CPU-GPU work balancing feature by having the autotuner pick a CPU/GPU ratio that defines how much of the data should be computed on each device. This ratio can be set in fixed 1/8th increments. The GPU-CPU ratio shared among all the



(a) A GPU task is always pushed to the bottom of the GPU management thread's queue.

(b) The GPU management thread pushes a CPU task to the bottom of a random CPU worker's deque.

(c) A CPU worker pushes a CPU task to the top of its own deque.

Figure 5. Three different cases when a task become runnable. The events progress from (1) to (3). Green tasks are GPU non-runnable tasks. Grey tasks are CPU non-runnable tasks.

rules in the same transform because they compute the same matrix but different regions. The system divides the matrix so that the first part is on GPU and the second part is on CPU.

5. Autotuner

This section describes the PetaBricks autotuner representation, the evolutionary autotuning algorithm, and how the synthesized OpenCL choices are represented to the tuner.

5.1 Autotuning Representation

Choices are represented in a configuration file that contains three types of structures. The first type is selectors which allow the autotuner to make algorithmic choices. Selectors can make different decisions when called with different input sizes dynamically. Using this mechanism, selectors can be used by the autotuner to construct poly-algorithms that dynamically switch algorithms at recursive call sites. Formally, a selector s consists of $\vec{C_s} = [c_{s,1}, \ldots, c_{s,m-1}] \cup \vec{A_s} = [\alpha_{s,1}, \ldots, \alpha_{s,m}]$ where $\vec{C_s}$ are input size cutoffs associated with algorithms $\vec{A_s}$. During program execution the runtime function $SELECT$ chooses an algorithm depending on the current input size by referencing the selector as follows:

$$SELECT(input, s) = \alpha_{s,i} \text{ s.t. } c_{s,i} > size(input) \geq c_{s,i-1}$$

where $c_{s,0} = 0$ and $c_{s,m} = \infty$. The components of $\vec{A_s}$ are indices into a discrete set of applicable algorithmic choices available to s, which we denote $Algorithms_s$.

In addition to these algorithmic choice selectors, the configuration file contains many other discrete tunable parameters. These tunable parameters include number of OpenCL work-items in the work-group, sequential/parallel cutoffs, and user defined parameters. Each tunable parameter is an integer with a positive bounded range.

5.2 Autotuning Algorithm

The autotuner uses an evolutionary algorithm to search through the available choice space. It maintains a population of candidate algorithms which it continually expands using a set of mutators and prunes by performance in order to allow the population to evolve more optimal algorithms. The input sizes used for testing during this process grow exponentially, which naturally exploits any optimal substructure inherent to most programs.

A key aspect of the autotuning algorithm is the mutation process. The mutation process is asexual, in that there is only a single parent algorithm for each newly created candidate. Additionally, newly created algorithms are only added to the population if they outperform the parent algorithm they were created from. Mutators are functions that create a new algorithm configuration by changing an existing configuration. The set of mutator functions is different for each program, and is generated fully automatically with the static analysis information extracted by the PetaBicks compiler.

There are different types of mutation operators that operate on specific parts of the program configuration. *Selector manipulation mutators* either add, remove, or change a level in a specific selector represented in the choice configuration file. *Synthetic function manipulation mutators* apply a change to a range of the underlying parameters based on the current input size being tested. Finally, *tunable manipulation mutators* randomly change a tunable value. Values that are compared to an input size, such as cutoffs, are scaled randomly by a value taken from a lognormal distribution. This means that small changes are more likely than large changes and a value is equally likely be halved as it is to be doubled. Values choosing from a set of choices, such as an algorithm, are chosen uniform randomly when mutated.

5.3 GPU Choice Representation to Autotuner

The compiler exposes four classes of GPU choices to the autotuner. First, there is the decision of if and when to use the GPU. This is encoded as an algorithmic choice in the selectors constructed by the autotuner. The autotuner can construct selectors that use the GPU for some input sizes and not others. The autotuner can also construct poly-algorithms that run some parts of the computation on the GPU and other parts on the CPU.

The second type of choice is memory mapping from PetaBricks

code to OpenCL. The choice indicates whether or not to use the local memory of the device when possible. This choice exists only if the OpenCL kernel with local memory version of a rule is generated. This is also mapped to an algorithmic choice using a selector constructed by to autotuner.

The third type is the number of work-items in the work-groups (or so called local work size) of each OpenCL kernel, since assigning the right local work size is a common optimization for GPU computing. These parameters are mapped to tunable values that the autotuner can explore independently of algorithmic choices.

The final one is GPU-CPU workload ratio of each transform, which defines what percentage of a region should be computed on the GPU. To limit the search space size, the possible ratios restricted to multiples of $1/8$.

In a big picture, every transform provides 12 levels of algorithmic choices for 12 different ranges of input sizes. Note that all levels of the same transform have the same number of algorithmic choices. For example, in SeparableConvolution configuration, each level of each `Convolve*` transform has three possible choices: using CPU backend, using OpenCL backend with global memory only, and using OpenCL backend with local memory optimization. Each `Convolve*` has two OpenCL kernels, so each of them has its own tunable parameters for local work size and GPU-CPU workload ratio. Apart from OpenCL related parameters, the configuration also includes other parameters such as split size for CPU work-stealing model, number of execution threads on CPU, and a sequential/parallel cutoff.

5.4 Challenges with Runtime Kernel Compilation

The PetaBricks autotuning approach runs large numbers of tests on small input sizes in order to quickly explore the choice space and seed exploration for larger input sizes. With only the CPU, these tests on small input sizes are very cheap and greatly improve convergence time. However, with our OpenCL backend these small tests take a much larger amount of time, because OpenCL kernels are compiled dynamically at runtime. These kernel compiles represent a fixed startup cost, often on the order of a few seconds, that dominate execution time for small input sizes. This factor increases autotuning time.

To address this problem we use two techniques. First, we cache the intermediate representation (IR) used by the OpenCL compiler. The first time we run a program, we store the OpenCL runtime-specific IR for each compiled kernel with the hash of the source for that kernel. This IR is reused in subsequent executions in order to skip the parsing and optimization phases of kernel compilation. Second, we adjusted the parameters of the autotuner so that it runs fewer tests at small input sizes. This involved both skipping extremely small input sizes entirely and running fewer tests on the smaller input sizes we do use. The result of these optimizations reduced typical training times from many days for some benchmarks to an average of 5.2 hours. This training time is still heavily influenced by architecture-specific JITing, which OpenCL does not allow to be cached. Full binary caching, as allowed by other languages such as CUDA, would further reduce training times.

6. Experimental Results

Our experimental results explore the extent to which different heterogeneous systems require different configurations to obtain optimal performance. To this end, our experiments test how configurations tuned for one heterogeneous system perform when run on a different system. We examine these differences both by testing relative performance and by examining the configurations and algorithmic choices of these tuned configurations.

Name	# Possible Configs	Generated OpenCL Kernels	Mean Autotuning Time	Testing Input Size
Black-Sholes	10^{130}	1	3.09 hours	500000
Poisson2D SOR	10^{1358}	25	15.37 hours	2048^2
SeparableConv.	10^{1358}	9	3.82 hours	3520^2
Sort	10^{920}	7	3.56 hours	2^{20}
Strassen	10^{1509}	9	3.05 hours	1024^2
SVD	10^{2435}	8	1.79 hours	256^2
Tridiagonal Solver	10^{1040}	8	5.56 hours	1024^2

Figure 8. Properties of the benchmarks. The number of configurations are calculated from the parameters described in Section 5.3.

6.1 Methodology

Figure 9 lists the representative test systems used in our experiments and assigns code names that will be used in the remainder of the section. We chose these machines to represent three diverse systems a program may run on. Desktop represents a high-end gaming system with a powerful processor and a graphics card. Server represents a large throughput-oriented multicore system one might find in a data center. It does not have a graphics card, we instead use a CPU OpenCL runtime that creates optimized parallel SSE code from OpenCL kernels. Laptop represents a laptop (it is actually a Mac Mini), with a low-power mobile processor and a graphics card. Together, our test systems cover graphics cards from both AMD and NVIDIA, use three different OpenCL runtimes, have two different operating systems, and range in cores from 2 to 32.

In our experiments we first create three program configurations by autotuning: *Desktop Config* is the configuration tuned on Desktop; *Server Config* is the configuration tuned on Server; and *Laptop Config* is the configuration tuned on Laptop.

Next, we run each of these three configurations on each of our three machines. Since we are migrating configurations between machines with different numbers of processors, for fairness, we remove the thread count parameter from the search space and set the number of threads equal to the number of processors on the current machine being tested. (On Server, the number of threads is set to 16 which provides better performance on every benchmark.)

Finally, for some benchmarks, we also include baseline results for comparison. We do this either by writing a PetaBricks program configuration by hand, by running OpenCL programs included as sample code in the NVIDIA OpenCL SDK, or by running CUDPP applications. We use the label *Hand-coded OpenCL* to indicate the performance of a standalone OpenCL program not using our system. These baselines are described in detail for each benchmark.

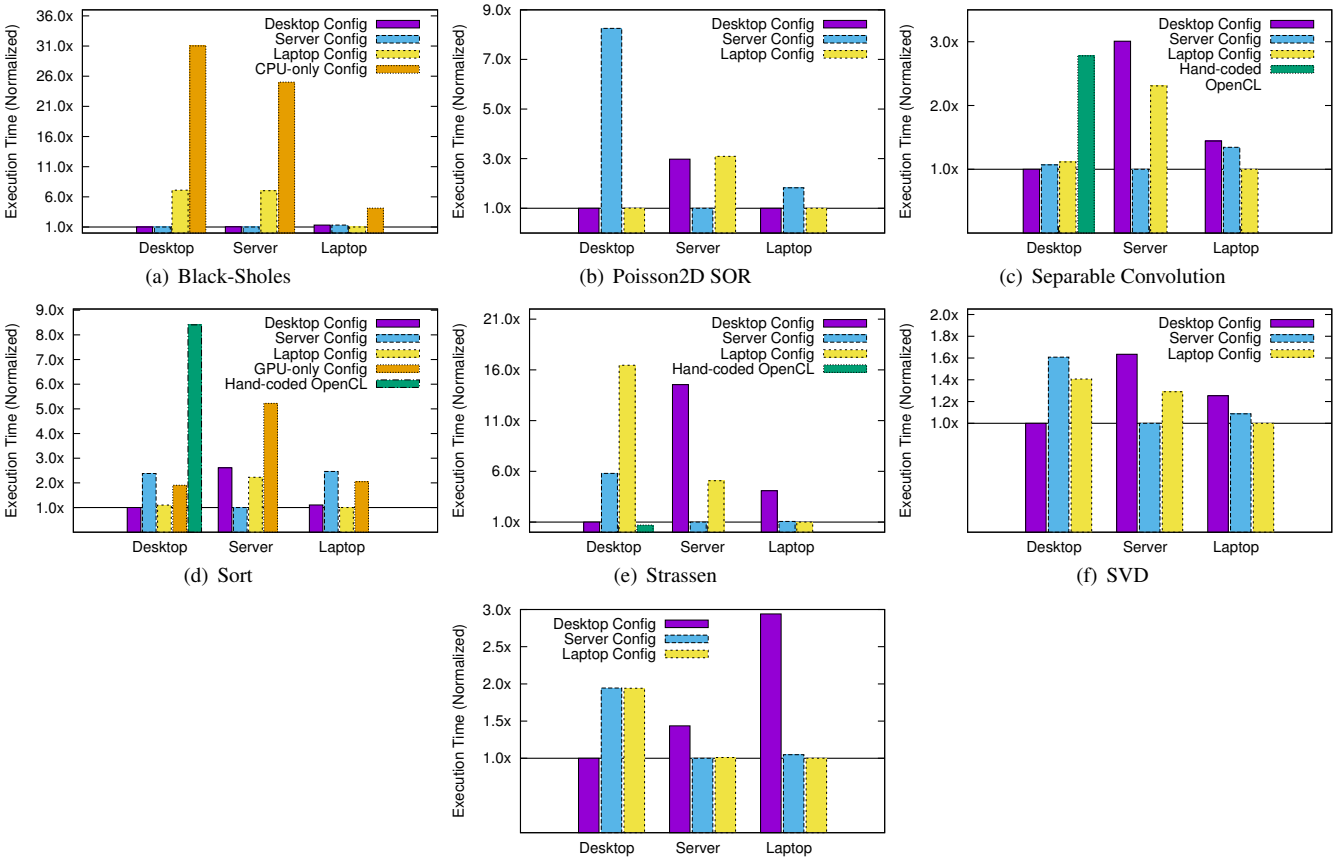
Figure 8 lists properties of each of our benchmarks from the PetaBricks benchmark suite. The configuration space is large, ranging from 10^{130} to 10^{2435}. Our system automatically creates up to 25 OpenCL kernels per benchmark. Average autotuning time was 5.2 hours. This training time is larger as a result of overheads from OpenCL kernel compilation, which dominate tuning time for many benchmarks. Individual benchmarks are described in more detail in the following section.

6.2 Benchmark Results and Analysis

Figure 6 summarizes the auto tuned configurations for each benchmark and Figure 7 shows the performance of each of these configurations on each system. Detailed analysis for each benchmark is provided in this section, and a more high level overview is provided in the next section.

	Desktop Config	Server Config	Laptop Config
Black-Sholes	100% on GPU	100% on OpenCL	Concurrently 25% on CPU and 75% on GPU
Poisson2D SOR	Split on CPU followed by compute on GPU	Split some parts on OpenCL followed by compute on CPU	Split on CPU followed by compute on GPU
SeparableConv.	1D kernel+local memory on GPU	1D kernel on OpenCL	2D kernel+local memory on GPU
Sort	Polyalgorithm: above 174762 2MS (PM), then QS until 64294, then 4MS until 341, then IS on CPU[1]	Polyalgorithm: above 7622 4MS, then 2MS until 2730, then IS on CPU[1]	Polyalgorithm: above 76830 4MS (PM), then 2MS until 8801 (above 34266 PM), then MS4 until 226, then IS on CPU[1]
Strassen	Data parallel on GPU	8-way parallel recursive decomposition on CPU, call LAPACK when $< 682 \times 682$	Directly call LAPACK on CPU
SVD	First phase: task parallism between CPU/GPU; matrix multiply: 8-way parallel recursive decomposition on CPU, call LAPACK when $< 42 \times 42$	First phase: all on CPU; matrix multiply: 8-way parallel recursive decomposition on CPU, call LAPACK when $< 170 \times 170$	First phase: all on CPU; matrix multiply: 4-way parallel recursive decomposition on CPU, call LAPACK when $< 85 \times 85$
Tridiagonal Solver	Cyclic reduction on GPU	Direct solve on CPU	Direct solve on CPU

Figure 6. Summary of the different autotuned configurations for each benchmark, focusing on the primary differences between the configurations. [1]For sort we use the abbreviations: IS = insertion sort, 2MS = 2-way mergesort, 4MS = 4-way mergesort, QS = quicksort, PM = with parallel merge.



(a) Black-Sholes (b) Poisson2D SOR (c) Separable Convolution

(d) Sort (e) Strassen (f) SVD

(g) Tridiagonal Solver

Figure 7. Benchmark performance when varying the machine and the program configuration. Execution time on each machine is normalized to the natively autotuned configuration. Lower is better. 7(c), 7(d), and 7(e) include *Hand-coded OpenCL* as a baseline taken from the NVIDIA SDK sample code. This baseline uses NVIDIA-specific constructs and only runs on our Desktop system. These hand-coded OpenCL baselines implement 1D separable convolution, radix sort, and matrix multiply respectively. As additional baselines, 7(b) includes a *CPU-only Config* which uses a configuration autotuned with OpenCL choices disabled and 7(d) includes *GPU-only Config* which uses PetaBricks bitonic sort on the GPU.

Codename	CPU(s)	Cores	GPU	OS	OpenCL Runtime
Desktop	Core i7 920 @2.67GHz	4	NVIDIA Tesla C2070	Debian 5.0 GNU/Linux	CUDA Toolkit 4.2[1]
Server	4× Xeon X7550 @2GHz	32	None	Debian 5.0 GNU/Linux	AMD Accelerated Parallel Processing SDK 2.5[2]
Laptop	Core i5 2520M @2.5GHz	2	AMD Radeon HD 6630M	Mac OS X Lion (10.7.2)	Xcode 4.2[1]

Figure 9. Properties of the representative test systems and the code names used to identify them in results. The OpenCL runtimes marked [1] targets the GPUs of the machines, while [2] targets the CPU by generating optimized SSE code.

Black-Sholes The Black-Scholes benchmark results (Figure 7(a)) show that on some machines it is fastest to divide the data and compute part of the output on the CPU and another part concurrently on the GPU. Black-Scholes implements a mathematical model of a financial market to solve for the price of European options. Each entry of the output matrix can be computed from the input matrix by applying the Black-Scholes formula.

The autotuned Black-Scholes configurations on the Desktop and Server both perform all computation using the GPU/OpenCL backend, while the configuration on the Laptop divides the work such that 25% of the computation is performed on the CPU, and 75% is performed on the GPU. This 25/75 split provides a 1.3x speedup over using only the GPU on the Laptop, while such a split produces a 7x slowdown on the other two systems.

The reason why these configurations perform this way is the OpenCL performance for Black-Sholes is an order of magnitude better than the CPU performance on the Desktop and Server. However, on the laptop the relative performance of the GPU is only about 4x the performance of the CPU. This can be seen in the *CPU-only Config* bar in Figure 7(a), which is a baseline configuration that does not use the GPU/OpenCL backend. On the laptop, where the relative performance of the two processors is close, exploiting heterogeneous parallelism between the CPU and the GPU results in performance gains.

Poisson2D SOR The Poisson2D SOR benchmark (Figure 7(b)) shows that the choice of which backend is best for each phase of the computation can change between machines. This benchmark solves Poisson's equation using Red-Black Successive Over-Relaxation (SOR). Before main iteration, the algorithm splits the input matrix into separate buffers of red and black cells for cache efficiency.

In the Desktop and Laptop tuned configuration, this splitting is performed on the CPU and then actual iterations are performed using the OpenCL GPU backed. In the Server configuration, nearly the opposite happens; the OpenCL backend is used for splitting a large area of the matrix, and the CPU backend is used for the main iterations. The reason for this switch is the very different performance profiles of the OpenCL backends. The Desktop and Laptop utilize actual GPUs while the Server OpenCL backend shares the CPU.

Separable Convolution The Separable Convolution benchmark (Figure 7(c)) highights the effect of algorithmic choices on the GPU. Three configurations, all using only OpenCL for computation, provide very different performance on each machine. This benchmark is used as a driving example in Section 2.1, which describes the choices benchmark in more detail. At width 7, shown here, Desktop performs best using 1D separable convolution on the GPU with the GPU local memory. Laptop, with the less powerful GPU, performs best using local memory but with the single-pass 2D convolution algorithm, because the overhead of synchronization and creating an extra buffer to store the intermediate results between the two passes dominates the computational savings of the separable algorithm. The best configuration on Server uses the OpenCL version of separable convolution, but without local memory prefetching, since the CPUs' caches perform best here without explicit prefetches.

The results also show 2.3x better performance than OpenCL baseline implementation taken from the OpenCL samples in the NVIDIA SDK (Figure 7(c)). Our implementation differs from this sample code in that in our generated code each work-item computes exactly one entry of the output, while in the sample code each work-item computes multiple entries of the output. This optimization in the sample code not only increases code complexity, but also results in worse performance than our PetaBricks implementation on the Tesla C2070. Performance on these machines is complex and unpredictable, making hard-coded choices often lose to our automatically inferred results.

Sort The Sort benchmark (Figure 7(d)) shows that for some tasks it makes sense to run on the CPU. The benchmark includes 7 sorting algorithms: merge sort, parallel merge sort, quick sort, insertion sort, selection sort, radix sort, and bitonic sort; in addition, merge sort and parallel merge sort have choices of dividing a region into two or four subregions. The configuration defines a poly-algorithm that combines these sort building blocks together into a hybrid sorting algorithm

None of the tuned configurations choose to use OpenCL in the main sorting routine (although some helper functions, such as copy, are mapped to OpenCL). The choices in CPU code are complex, and result in up to a 2.6x difference in performance between autotuned configurations. Each configuration is a poly-algorithm that dynamically changes techniques at recursive call sites. Desktop uses 2-way merge sort with parallel merge at the top level, switches to quick sort when sizes of sub-arrays are smaller, switches to 4-way merge sort with sequential merge when sizes are even smaller, and finally ends with insertion sort as a base case when size is less than 341. Server uses 4-way merge sort with sequential merge, switches to 2-way merge sort when sizes are smaller, and finally switches to insertion sort when size is less than 2730. Laptop alternatively switches between 4-way and 2-way merge sort, uses parallel merge until size is less than 34266, and switches to insertion sort when size is less than 226.

For comparison, we wrote a configuration by hand that uses bitonic sort in OpenCL using our system (*GPU-only Config* in Figure 7(d)). This configuration is between 1.9 and 5.2x slower than the native autotuned configuration. Interestingly, this configuration does beat the Server configuration on both of the other machines that have GPUs. This means that, if one had the Server configuration, using the GPU instead would give a speedup, but not as much of a speedup as re-tuning on the CPU.

As a second baseline, we include the radix sort sample program from the NVIDIA SDK (*Hand-coded OpenCL* in Figure 7(d)). This OpenCL implementation of Sort performs 8.4x worse than our autotuned configuration and 4.4x worse than our bitonic sort configuration. The poor performance of both of these GPU Sort baselines, relative to our autotuned Sort, speak to the difficulty writing a high performance Sort on the GPU. Researchers have developed faster methods for GPU sorting, however, these methods require both an autotuning system and heroic programmer effort [**?**], and their performance generally does not account for overhead in copying data to and from the GPU, which our benchmarks do.

Strassen The Strassen benchmark (Figure 7(e)) shows that choosing the right configuration for each architecture results in very large performance gains. The Strassen benchmark performs a dense matrix-matrix multiplication. The choices include: transposing any combination of the inputs; four different recursive decompositions, including Strassen's algorithm; various blocking methods; naive matrix multiplication; and calling the LAPACK external library.

Figure 7(e) shows the performance of our Strassen benchmark with different configurations. Laptop configuration gives a 16.5x slowdown on Desktop. OpenCL is used in the Desktop configuration, and C++/Fortran (through a call to LAPACK) is used in the Server and Laptop configurations. The large computation to communication ratio in matrix multiplication stresses the difference in relative CPU/GPU computational power between Desktop, with a high performance graphics card, and Laptop, with a mobile graphics card. This results in the Desktop GPU producing a speedup compared to its CPU, and the Laptop GPU producing a slowdown.

Altough Server and Laptop both use CPUs, their optimal configurations are different. On Server, the best algorithm is to recursively decompose the matrices in 8-way parallel fashion until the regions are smaller than a certain size, call LAPACK on the small regions, and finally combine the data. On Laptop, the best algorithm is to make a direct call to LAPACK without any decomposition.

As a baseline, we include the matrix multiplication OpenCL sample from the NVIDIA SDK (*Hand-coded OpenCL* in Figure 7(e)). This baseline runs 1.4x faster than our autotuned configuration on Desktop. The reason for this difference is the hand-coded OpenCL uses a number complex manual local memory optimizations that accumulate partially computed outputs in local memory shared between work-items. We have not implemented a similar optimization in our system; however, it would be possible to automatically perform a similar optimization.

Singular Value Decomposition (SVD) The results for SVD (Figure 7(f)) are particularly interesting because on some systems the autotuner constructs a poly-algorithm with task parallel divisions between the GPU and the CPU, and the complexity of the benchmark provides a large number of choices in the search space. This benchmark approximates a matrix through a factorization that consumes less space. SVD is a variable accuracy benchmark where many of the choices available to the autotuner, such as how many eigenvalues to use, impact the quality of of the approximation. The autotuner must produce an algorithm which meets a given accuracy target. These variable accuracy features are described in more detail in [4].

On Desktop, the autotuned configuration divides the work by using the GPU to compute one matrix and the CPU to concurrently compute another matrix. Since the computations of the two matrices are very similar, and the CPU and the GPU have relatively similar performance for these computations on Desktop, overall throughput is increased by dividing the work in this way.

This benchmark also demonstrates that the best configurations of a sub-program might be different when the sub-program is a part of different applications even running on the same machine. The SVD benchmark uses the Strassen code to perform matrix multiplication. However SVD uses matrix multiply on sub-regions of multiple larger arrays (resulting in different data locality) and possibly making multiple calls concurrently. Due to the different data-accessing patterns, the cache behaviors are not the same on the CPU, the bank conflicts on GPU memory vary, and the interactions between subsystems change. This makes the best configurations differ for Strassen inside SVD and in isolation. While the best matrix multiplication configuration on Desktop for Strassen in isolation always uses the GPU, the best one for this benchmark is 8-way parallel recursive decomposition and then calling LAPACK. The autotuned configurations on Server and Laptop for this benchmark are also different from Strassen in isolation.

Tridiagonal Solver The Tridiagonal Solver benchmark (Figure 7(g)) shows that often algorithmic changes are required to utilize the GPU. The benchmark solves a system of a equations where the matrix contains non-zero elements only on cells neighboring and on the diagonal and includes a variety of techniques including polyalgorithms. We implement a subset of the algorithmic choices described in [9, 30].

Similar to the Strassen benchmark, the GPU is only used on the Desktop machine; however, in order to utilize the GPU, an algorithmic change is required. Cyclic reduction is the best algorithm for Desktop when using the GPU. If a machine does not use OpenCL, it is better to run the sequential algorithm as demonstrated used on Server and Laptop.

Our best configuration on Desktop is 3.5x slower than CUDPP[30] on input size 512. Some of this slowdown is a result of OpenCL being generally slower than CUDA when using the NVIDIA toolchain. Additionally, our automatically generated OpenCL kernel is not as highly optimized as CUDPP's kernel, which guarantees the efficient use of shared memory without bank conflicts.

6.3 Results Summary

In all of our benchmarks, algorithmic choices—which traditional languages and compilers do not expose—play a large role in performance on heterogeneous systems since the best algorithms vary not only between machines but also between processors within a machine. Ultimately, the complex and interdependent space of best mapping choices seen in these benchmarks would be very difficult to predict from first principles, alone, but our empirical exploration effectively and automatically accounts for many interacting effects on actual machine performance, in each program on each system.

Taken together, these seven benchmarks demonstrate even more of the complexity in mapping programs to heterogeneous machines than any one benchmark alone.

- *Strassen* shows that choosing the right configuration for each specific architecture can provide a huge performance improvement. In this benchmark, choosing to use a data parallel accelerator can yield large speedups (16.5x) on some machines, and large slowdowns (4.1x) on others, for the same problem, depending on the exact characteristics of all heterogeneous processors in the system.
- *Poisson 2D SOR* further supports the previous argument by showing that the optimal placement of computations across the processors in one system is almost the opposite of another.
- *Tridiagonal Solver* demonstrates that not only where computations run, but also which algorithms to use on that particular resource, can dramatically affect performance.
- *Separable Convolution* shows that, even when a program is best run entirely on the data-parallel compute resources, the best algorithm to use and the best strategy for mapping to the complex heterogeneous memory hierarchy vary widely both across machines and across program parameters (kernel size).
- *Sort*, on the other hand, shows that even parallel problems may not always be best solved by data parallel accelerators, and that complex machine-specific algorithmic choice is still essential to performance on a smaller array of conventional processors.
- *SVD* shows that the best performance can require mapping por-

tions of a program across all different processor types available, and together with Strassen, it shows that the best configurations of the same sub-program in different applications vary on the same system.

- Finally, while SVD confirms that sometimes it is best to run different portions of a program on different subsystems concurrently, *Black-Scholes* illustrates that sometimes it is best to run the same portion of the program but different regions of data across multiple heterogeneous subsystems simultaneously; therefore, considering the workload balance between processor types is important to achieve the optimal performance on a heterogeneous system.

7. Related Work

A number of offline empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains. ATLAS [27] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [10] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [14] for sparse matrix computations, SPIRAL [23] for digital signal processing, and OSKI [26] for sparse matrix kernels.

The area of iterative compilation contains many projects that use different machine learning techniques to optimize lower level compiler optimizations [1, 2, 12, 22]. These projects change both the order that compiler passes are applied and the types of passes that are applied. However, these projects do not explore the type of algorithmic choices that the PetaBricks language exposes, and these systems create only compile time, not runtime choices.

The use of autotuning techniques is even more commonplace when optimizing GPGPU programs. Autotuning is typically applied in a program or domain-specific fashion. Such systems use autotuners to construct poly-algorithms for solving large tridiagonal systems [9], for tuning GPU sorting algorithms [**?**], autotuning 3D FFT with a focus on padding and bandwidth optimizations [21], autotuning sparse matrix-vector multiply by building performance models [8], and to tune dense linear algebra with a mix of model-driven and empirical techniques [25]. These techniques are often specific to a problem or class of problems.

Besides problem-specific techniques, there is a high-level directive-based GPU programming that uses HMPP workbench to automatically generate CUDA/OpenCL code, and auto-tunes on the optimization space on the generated GPU kernels [13]. However, this technique and the previous ones only point towards autotuning as a necessity in order to get the best performance on modern GPUs; they do not address how to utilize all available resources together to achieve the best performance on a heterogeneous system.

Several methods to efficiently distribute workload between different devices have been studied. StarPU applies work-stealing framework to balance work among subsystems [5]. However, StarPU requires the programmer to write separate CPU and GPU code, and relies entirely on dynamic work-stealing guided by automatic hints to distribute tasks. CnC-HC automatically generates CPU, GPU, and FPGA code and uses a work-stealing scheduler to distribute work among different devices guided by manual hints only [24]. Qilin, automatically generates code and uses *adaptive mapping* for performance tuning [20]. During the training run, Qilin executes the program on different input sizes on CPUs and GPUs separately, and uses the result to determine workload partitions between the CPUs and the GPUs. However, the mapping of these systems may not be ideal. Our system automates the entire process, both translating kernels to different targets, and empirically determining where they should run in our hybrid work-stealing/work-pushing runtime. For real applications, the ideal mapping is globally non-linearly inter-dependent with all other choices, and our global optimization captures this during autotuning. Our results demonstrate the importance of global learning.

CGCM [15] uses a technique for automatic management of GPU/CPU memory communication. This technique is similar to our analysis for determining when lazy or eager copy-outs are used. Their technique uses a conservative reachability analysis to determine where to insert calls into a runtime that dynamically tracks and maps data to different memories. They focus on a number of optimizations to this technique to reduce runtime overhead. While CGCM manages data movement automatically, it requires some programmer help when managing parallelism.

A number of other programming languages attempt to make programming for GPGPU devices easier. CUDA-lite [29], automates some of the parallelization and memory management tasks in writing CUDA. JCUDA [28] alleviates many of the difficulties in using the GPU from Java code. There have also been efforts to map subsets of C to the GPU [6, 19]. These techniques put affine data access restrictions on the program. There have been other efforts to map OpenMP to the GPU [17]. While these efforts make running code on the GPU easier, they will produce the same code for each heterogeneous system and do not allow algorithmic choice, or empirically infer the best mapping for each machine.

Researchers have also studied the relative throughput of the CPU and the GPU and discovered contradictory findings with some studies showing 100x performance differences and others just 2.5x [18]. Our work sheds more light on this debate, showing that both claims can be correct. The best device to use is highly dependant both on the architecture and algorithmic choices and cannot be determined by simple relative performance numbers. We also show cases where the best throughput is obtained by using both the CPU and the GPU in parallel.

8. Conclusions

Programmers and compilers collaborate to map problems into precise machine instructions. There are many choices in how to map a given problem onto a particular machine. The programmer typically makes the higher-level choices (e.g. which algorithms to use), while a compiler will make the lower-level choices in how to map to a specific machine (e.g. register allocation, instruction selection, loop transformations). Both the programmer's and the compiler's decisions are often based on heuristic models of machine performance. Simplistic models of machine performance are assumed when the programmer performs O analysis or cache blocking, or when the compiler uses heuristics like minimizing instructions executed or stack spills.

It is increasingly difficult for a single model to accurately represent the complex heterogeneous machines in widespread use today. Further, on these machines, high-level algorithmic and scheduling choices, and low-level machine mapping choices, interact in complex ways to influence actual machine performance. We have found, and our results show, that simultaneous empirical exploration of algorithmic and machine mapping choices can effectively compile an array of programs to efficient execution on modern heterogeneous parallel machines, including both multicore CPUs and GPUs. The best choices for each heterogeneous machine are often complex, and map portions of different algorithms in multiple ways across all processors and memories in the machine. Our results further show that this full choice space is important to consider, since the best algorithms and mapping strategies on one heterogeneous system are often not the same as on another.

Models are still useful in some situations. The search space

of all possible choices in algorithm and machine mapping is enormous, and many individual choices have optimal substructure, so reducing the empirical search space with model-driven choices can be both essential and effective, even when autotuning. Most of all, models aid human comprehension, which can be essential to creating new algorithms and optimizations, and new architectures on which to run them. But compilers should not be wedded to models and heuristics alone when faced with programs and machines of ever-increasing complexity.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback, David Garcia for many suggestions on OpenCL related problems on the Khronos forum, and NVIDIA for donating a graphics card used to conduct experiments. This work is partially supported by DOE award DE-SC0005288, DOD DARPA award HR0011-10-9-0009 and NSF award CCF-0632997.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Symposium on Code Generation and Optimization*, 2006.

[2] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA, 2004.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[4] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Symposium on Code Generation and Optimization*, Chamonix, France, Apr 2011.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2), 2011.

[6] Muthu Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-CUDA code generation for affine programs. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011. Springer Berlin / Heidelberg, 2010.

[7] J. Charles, P. Jassi, N.S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel Core i7 Turbo Boost feature. In *Symposium on Workload Characterization*, Oct 2009.

[8] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2010.

[9] Andrew Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Parallel and Distributed Processing Symposium*. IEEE, May 2011.

[10] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), February 2005.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Programming language design and implementation*, New York, NY, USA, 1998.

[12] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and Francois Bodin. MILEPOST GCC: machine learning based research compiler. In *GCC Developers' Summit*, Jul 2008.

[13] Scott Grauer-Gray, Lifan Xu, Robert Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing Conference*. IEEE, May 2012.

[14] Eun-jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Computational Science*. Springer, 2001.

[15] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *Programming language design and implementation*, New York, NY, USA, 2011.

[16] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11), November 2005.

[17] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44, February 2009.

[18] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *international symposium on Computer architecture*, New York, NY, USA, 2010.

[19] Allen Leung, Nicolas Vasilache, Benoit Meister, Muthu Baskaran, David Wohlford, Cedric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Workshop on General-Purpose Computation on Graphics Processing Units*, New York, NY, USA, 2010.

[20] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *International Symposium on Microarchitecture*, New York, NY, USA, 2009.

[21] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-d FFT library for CUDA GPUs. In *High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2009.

[22] Eunjung Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Symposium on Code Generation and Optimization*, April 2011.

[23] Markus Puschel, Jose M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robbert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE*, volume 93. IEEE, Feb 2005.

[24] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. In *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, New York, NY, USA, 2012.

[25] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing*, November 2008.

[26] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Scientific Discovery through Advanced Computing Conference*, San Francisco, CA, USA, June 2005.

[27] Richard Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 1998.

[28] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704. Springer Berlin / Heidelberg, 2009.

[29] Sain zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *Workshops on Languages and Compilers for Parallel Computing*. Springer, 2008.

[30] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Symposium on Principles and Practice of Parallel Programming*, January 2010.