Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors

Jayanth Gummaraju Laurent Morichetti Michael Houston
Ben Sander Benedict R. Gaster Bixia Zheng

Compute Research Group, Advanced Micro Devices Inc., Sunnyvale, California, USA
firstname.lastname@amd.com

ABSTRACT

Modern processors are evolving into hybrid, heterogeneous processors with both CPU and GPU cores used for general-purpose computation. Several languages such as Brook, CUDA, and more recently OpenCL are being developed to fully harness the potential of these processors. These languages typically involve the control code running on the CPU and the performance-critical, data-parallel kernel code running on the GPUs.

In this paper, we present Twin Peaks, a software platform for heterogeneous computing that executes code originally targeted for GPUs efficiently on CPUs as well. This permits a more balanced execution between the CPU and GPU, and enables portability of code between these architectures and to CPU-only environments. We propose several techniques in the runtime system to efficiently utilize the caches and functional units present in CPUs. Using OpenCL as a canonical language for heterogeneous computing, and running several experiments on real hardware, we show that our techniques enable GPGPU-style code to execute efficiently on multicore CPUs with minimal runtime overhead. These results also show that for maximum performance, it is beneficial for applications to utilize both CPUs and GPUs as accelerator targets.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming

General Terms: Design, Experimentation, Performance.

Keywords: GPGPU, Multicore, OpenCL, Programmability, Runtime.

1. INTRODUCTION

General-purpose computation on the GPU, commonly referred to as GPGPU ([1, 5, 17, 3]), is becoming increasingly popular as a way to accelerate applications that need large amounts of processing power (FLOPS) and/or memory bandwidth. Several applications in image and video processing, games, financial modeling, and scientific computing have been shown to benefit significantly from this style of computation [19].

The GPGPU model advocates an accelerator style of computation in which the CPU executes control code and offloads performance-critical data-parallel code to the GPU. Several frameworks have been developed to exploit GPUs as accelerators, including BrookGPU [5], Accelerator [23], CUDA [17], and more recently, OpenCL [3].

With the emergence of heterogeneous environments comprising closely integrated CPUs and GPUs, we advocate extending the GPGPU model to support both CPU and GPU cores as accelerator targets. Hardware vendors have already announced products with fused CPU and GPU cores for the next-generation laptop and desktop environments (e.g., AMD Fusion^{TM}, Intel Sandybridge, etc.). Software frameworks, like OpenCL, are likewise beginning to tackle supporting heterogeneous devices. In these environments, the gap between CPU and GPU in terms of raw FLOPS and memory bandwidth, as well as the cost of moving computation and data between them, is expected to be much smaller than today, making it necessary to utilize resources more efficiently on both processors to maximize performance.

This new model motivates the need for a software platform that can distribute performance-critical computation among the underlying compute resources, whether they are CPUs or GPUs. This permits a more balanced execution between the CPU and GPU, and enables portability of code between these architectures and to CPU-only environments. However, the diversity and significant architectural differences between CPUs and GPUs present several challenges to develop an efficient system.

In this paper, we present Twin Peaks, a software platform that enables applications originally targeted for GPUs to be executed efficiently on multicore CPUs. Our system maximizes the utilization of functional units in the CPU (e.g., SSE units of x86) by exploiting the data locality and data parallelism exposed by the GPGPU model through computation kernels. We develop our system using the vendor-independent OpenCL specification [3] as a canonical programming environment for heterogeneous computing. Our overall design consists of a compilation system, based on the LLVM [15] framework with a few extensions, and a runtime system that includes several novel mapping and optimization techniques for efficient execution on multicore CPUs.

We focus on bridging the gap between thousands of hardware threads on the GPU and a handful of hardware contexts available on multicore CPUs. We also target mapping the complex GPU memory system comprised of several ad-

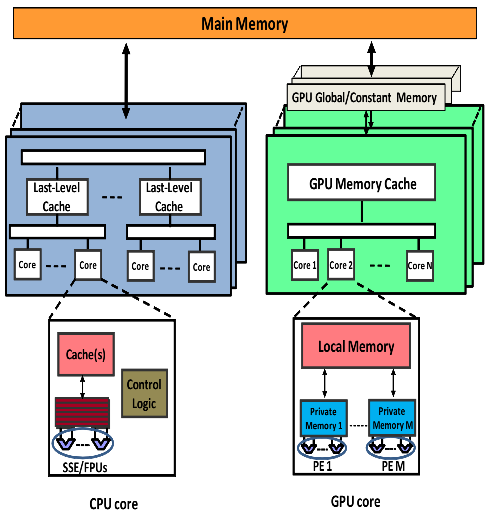


Figure 1: Heterogeneous Computing

dress spaces (e.g., private, local, etc.), to the CPU memory system, with special consideration to cache locality and reuse. We present several low-level optimizations targeted towards improving cache and TLB utilization, reducing the number of instructions and data-cache footprint using customized code while maintaining debug-ability. These optimizations are non-invasive to the computation kernel source code and performed entirely inside the runtime system.

We evaluate our system on real hardware (Intel Nehalem, AMD Istanbul, and Nvidia GeForce GTX 280/9600M processors) for several GPU-friendly applications and microbenchmarks. We demonstrate good performance on consumer multicore CPUs and efficient scaling on a large SMP machine. We show that when the datasets are small, kernels contain data-dependent control flow, or offload costs are high relative to computation time, CPUs can outperform GPUs. These results show that for maximum performance, it is beneficial for applications to utilize both CPUs and GPUs as accelerator targets. We also show that Twin Peaks has minimal overhead, and allows context switching between work-items in less than 10 ns for typical work-group sizes.

2. BACKGROUND AND MOTIVATION

The term *heterogeneous* computing is broadly used to refer to computation on processors/cores that have attributes different from one another (e.g., CPU cores with different degrees of multithreading [14], CPUs and FPGAs (e.g., Cray XD1), CPU cores and stream cores [11], etc.). In this paper, we exclusively focus on a system with CPUs and GPUs (e.g., AMD Fusion^{TM}, Intel Xeon with NVIDIA GPU, Intel Sandybridge, etc.).

Figure 1 shows a typical heterogeneous processor with several CPUs and GPUs. Each CPU and GPU, in turn, contains several computation cores, which could be part of the same die (e.g., AMD Fusion^{TM}) or on different dies (e.g, Intel Xeon with NVIDIA GPU). While the CPU cores have abundant control logic for executing general-purpose applications with instruction-level parallelism, GPUs are more tuned to execute graphics applications that are regular and have abundant data-parallelism. Figure 1 showcases these attributes with the CPU core containing few functional units (e.g., SSE units of x86 processors), and the GPU cores containing several of them at the expense of extensive control logic.

The GPU portion of Figure 1 shows the abstract processor model of the GPU. The figure shows a number of cores N (10s) each containing M (100s) processing elements (PEs). The PEs contain several functional units including special purpose units such as inverse-square root and sine/cosine. Each PE, in turn, is directly connected to a *private memory* (maps to GPU register file) and a group of PEs share a *local memory*. All the cores share a global data cache, which is directly connected to off-chip *global* and *constant memories*.

This abstract processor model directly translates into the execution model for the kernels. Each GPU thread, or *work-item*, executes on one PE and the private data needed by the work-item is directly accessed from its private memory (up to MxNxP work-items in flight (Figure 1)). Sets of these work-items that constitute a *work-group* share data during execution through the local memory attached to the core. The work-items operate on data input from the global memory via the data cache, and output the processed data back to the global memory via the data cache. Constant memory contains data that remain constant throughout the execution of the kernel and is also cached.

Synchronization between work-items is limited to those belonging to the same work-group. Although this can be seen as a limitation, it is imperative to have this restriction to be able to effectively manage thousands of threads without incurring significant hardware overhead associated with implementing larger degree of synchronization.

In essence, the overall execution of a computation kernel is as follows: A control thread executing on the CPU dispatches the kernel to a compute device (GPU, for example). The compute device breaks the kernel execution into a number of N-dimensional work-groups equal to that specified by the application. The work-groups are then dynamically scheduled on the cores. Once all the work-groups finish execution, the control thread is notified.

3. TWIN PEAKS OVERVIEW

In this section, we describe the overall system framework to program, compile, and execute applications in a heterogeneous environment with CPUs and GPUs using Twin Peaks. The programming and compiler systems mostly reuse existing technologies with small extensions. We focus on the runtime system and present the overall design for high-performance mapping.

3.1 Programming Environment

Without loss of generality, we use the programming environment prescribed by the OpenCL [3] specification to design and implement our system. The OpenCL specification utilizes a subset of ISO C99 with some extensions for parallelism. This facilitates porting of existing applications and reuse of existing toolchains for compilation, debugging, etc. OpenCL is vendor-independent and ratified by several hardware vendors, which enables us to compare Twin Peaks against other implementations.

The OpenCL API provides an abstraction to the programmer using command queues to create and dispatch work to the underlying compute devices (Table 1).The API, in its current form, is a fairly low-level software layer to target CPUs and GPUs. We expect that once this layer is more commonly used and more processor classes are well supported, this API will be targeted by higher layers (e.g., domain-specific languages, C/C++ parallel libraries, etc.) to facilitate easier programming.

The kernels written for execution on the compute device are C functions, with some restrictions and extensions. Here are some of the important characteristics:

- Work-items uniquely determined by their N-dimensional ids (`global id`: unique work-item id; `local id`: id within a work-group; `group id`: id of the work-group)
- Uses qualifiers to designate kernels (`__kernel`) and address spaces: `__global, __local, __constant,` and `__private`
- Provides functions to synchronize between work-items (e.g., `barrier()`, `atom_add()`, etc.)
- Places restrictions on pointer usage to remove aliasing and facilitate compiler optimizations
- Supports extended data types for scalar (e.g., `half`) and vector operations (e.g., `float16`)

Although the OpenCL specification provides several other APIs (e.g., for creating contexts, querying devices, OpenGL interoperability, etc.) and built-in functions (e.g., math functions), we focus on these aspects because they constitute the core attributes needed to test our system. Whereas the runtime system implements the OpenCL API, the compiler is responsible for compiling the application kernels for the specified device (Figure 2), as we discuss next.

3.2 Compilation System

The compiler transforms application kernels into device-specific binaries. It also provides a path for source-to-source transformation to standard C99. Figure 2(a) showcases the compiler framework. The compiler uses an industry-standard C front-end along with the LLVM [15] framework with extensions to support OpenCL. The front-end is extended to support additional data-types (e.g., `half, float8,` etc.), additional keywords (`__kernel, __global,` etc.) and built-in functions (e.g., `global_id()`, `barrier()`, etc.). Several additional syntactic and semantic checks are performed to ensure the kernels meet the OpenCL specification.

The front-end creates a LLVM-IR, carrying over OpenCL specific information as metadata structures. For example, for efficient debug support of kernels, we create metadata structures to hold the debug information generated by the front-end and insert a pass to translate this into LLVM debug nodes, which includes the line numbers and source code mapping. The LLVM-IR is then optimized using standard optimization passes provided in LLVM. In general, the set of optimizations that benefit OpenCL kernels executing on the CPU is the same set that applies for sequential code (e.g., loop unrolling/peeling, constant propagation, etc.). However, it is often possible, due to the parallel semantics of the application kernels, to aggressively apply these optimizations without the limitations of dependency analysis enforced by languages with aliases. LLVM-AS is then used to generate x86 binary if the target is CPU. If it is GPU, LLVM-IR is transformed to GPU-specific IR, then a GPU compiler is used to generate the binary.

3.3 Runtime System

The runtime system is responsible for scheduling the work in command queues to the underlying devices, respecting the dependencies among them. Commands are enqueued using the `clEnqueue` API shown in Table 1. The commands broadly fall into three categories: 1) Kernel commands (e.g., `clEnqueueNDRangeKernel()`), which execute the application kernels on the compute device; 2) Memory commands (e.g., `clEnqueueReadBuffer()`), which transfer data between the host memory and compute device memory; and, 3) Event commands (e.g., `clEnqueueWaitForEvents()`), which specify dependencies between the kernel and memory commands.

Figure 2(b) illustrates the overall functionality of the runtime system. The application may create multiple command queues (e.g., some in libraries, for different components of the application, etc.). These queues are muxed into one queue per device type (e.g., CPU device queue). The figure shows command queues 1 and 4 merged into one CPU device queue. The device queue then schedules work onto the multiple compute cores.

On the GPU, commands are dispatched to a hardware sequencer for scheduling. The dotted line in the figure for the GPU indicates that we do not discuss the mapping to GPU in this paper, since the focus is on executing on multicore CPUs.

On the CPU, each kernel command is broken down into multiple smaller commands that are dispatched to the cores. The event commands are used to determine which command gets executed next. The cores then schedule the commands on their compute units (e.g., SSE units). Since the host memory and device memory are the same on the CPU, the memory commands just assign the appropriate source and destination pointers for data transfers, unless explicitly requested otherwise by the application.

4. MAPPING TO MULTICORE CPUS USING TWIN PEAKS

In this section, we present techniques to bridge the gap between an execution model with thousands of hardware threads on GPUs and a handful of hardware contexts available on multicore CPUs. We showcase mapping a complex memory system comprising of several address spaces (e.g., private, local, etc.) to the CPU memory system, with special consideration to cache locality and reuse. We also present low-level optimizations to minimize overheads.

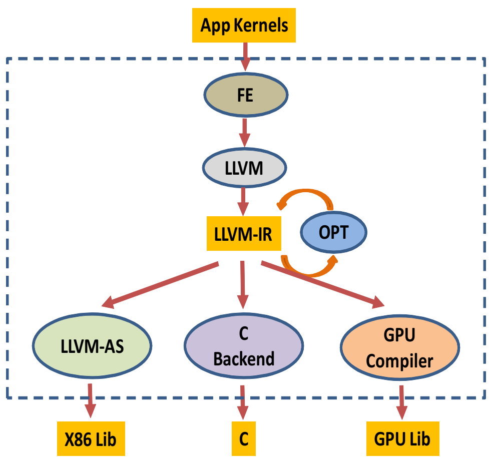
4.1 Mapping Computation Kernels

Computation kernels, as described in Section 2, are data-parallel, with thousands of threads simultaneously in flight. These threads need to be mapped to CPUs with far fewer hardware contexts. The OS is responsible for scheduling and executing system-level OS threads on these hardware contexts. A naïve mapping, in which each GPU thread is mapped to an OS thread (e.g., using pthreads) involves the OS managing thousands of active threads. This not only degrades performance significantly, assuming the OS can support so many threads, but also affects debugging since monitoring more than a handful of active threads is a difficult problem.

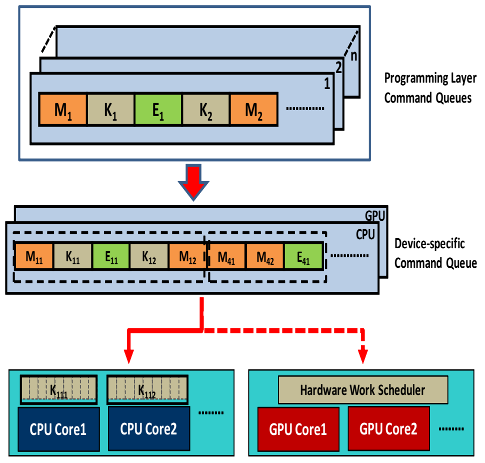
Figure 3(a) shows our mapping of GPU threads/work-items onto CPU cores using light-weight threading (LWT). The basic idea is to combine fine-grain GPU threads consti-

OpenCL API function	Description
clCreateCommandQueue()	Create a command queue for a specific device (e.g., CPU, GPU)
clCreateProgramWithSource()	Create a program object using the source code of the application kernels
clBuildProgram()	Compile and link to create a program executable from the program source or binary
clCreateKernel()	Create a kernel object from the program object
clCreateBuffer()	Create a buffer object for use via OpenCL kernels
clSetKernelArg(), clEnqueueNDRangeKernel()	Set the kernel arguments and enqueue the kernel in a command queue
clEnqueueReadBuffer(), clEnqueueWriteBuffer()	Enqueue a command in a command queue to read from a buffer object to host memory or write to the buffer object from host memory
clEnqueueWaitForEvents()	Wait for the specified events to complete

Table 1: Key OpenCL API methods implemented in Twin Peaks



(a) Compilation system. FE: Front-End; OPT: Optimizer; LLVM-AS: LLVM Assembler


n
2
1
M_1 K_1 E_1 K_2 M_2
Programming Layer Command Queues
GPU
CPU
M_{11} K_{11} E_{11} K_{12} M_{12} M_{41} M_{42} E_{41}
Device-specific Command Queue
K_{111}
K_{112}
CPU Core1
CPU Core2
Hardware Work Scheduler
GPU Core1
GPU Core2

(b) Runtime system. $X_{i,j,k}$:: X: kernel (K), memory (M), or event (E) command, i: command queue; j: device; k: core.

Figure 2: Twin Peaks Design

tuting a work-group into a single OS-thread that executes on a single CPU core. This mapping not only significantly alleviates the context switching overhead, but also removes the penalty of switching between user and kernel modes, since all the work-items are managed in user space.

Figure 3(a) shows two scenarios for kernels: 1) has no synchronization primitives, and 2) has synchronization primitives. In scenario 1, the work-items in a work-group are combined into a single CPU thread that executes the kernel by processing the first work-item to completion, then the second work-item to completion, and so on until all the work-items in the work-group are processed. Once the thread finishes the execution of a work-group, the same thread executes the next work-group, and so on. The runtime instantiates a number of threads equal to the number of CPU cores and pins them to the cores. These threads execute the work-groups dynamically assigned to them by the runtime.

Scenario 2 is handled similarly, except that the synchronization (indicated by *barrier*()) inside the kernel implies that work-items cannot be processed to completion without interruption. All the work-items need to be processed until the synchronization point, before executing the code beyond this point. We accomplish this by executing the first work-item until the synchronization point, then saving its state and jumping to the execution of the second work-item (*setjmp*()), and so on until all the work-items in the work-group are processed. The control then jumps back to the stored state of the first work-item (*longjmp*()) and executes until the next synchronization or work-item completion point, jumps to the stored state of the second work-item and so on. In Section 4.3, we present low-level optimizations to the traditionally expensive *setjmp*()/*longjmp*() routines, and show how they can be implemented with little overhead ($\sim$ 10 ns per work-item).

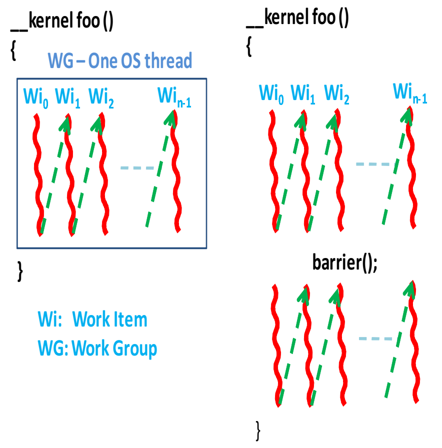
To facilitate debugging, we make no changes to the source code of the kernels. Instead, we hide the execution of fine-grain multithreading inside the runtime system. For example, the *setjmp*() and *longjmp*() calls are embedded inside the synchronization routines and start/exit of kernels. This helps to easily map errors back to the original kernel code.

4.2 Mapping Memory Spaces

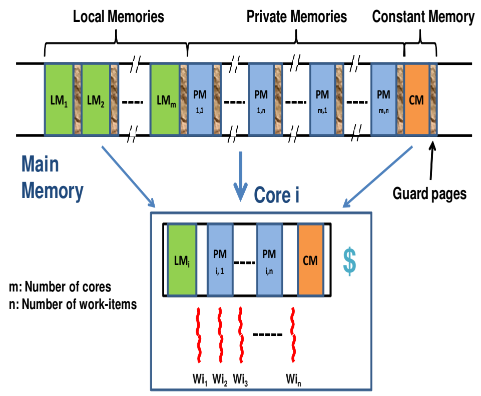
Twin Peaks maps the various memory spaces viz. global memory, local memories, private memories, and constant memory onto the CPU memory/cache hierarchy. It's sufficient to map the global memory to the CPU main memory. For high performance, the remaining memories must be mapped to the CPU cache hierarchy as well. This translates to assigning memory regions in the CPU main memory space such that cache locality is maximized and the different memory regions co-reside in the cache hierarchy.

4.2.1 Mapping Local Memories

As described in Section 2, local memory is the memory shared between the work-items belonging to the same work-group. Since a work-group executes on only one core (using the LWT approach (Section 4.1)), the corresponding local memory gets brought into its private cache. Although



(a) Mapping threads using LWT



(b) Mapping memory spaces

Figure 3: Conceptual Mapping to CPUs

the size of local memory on GPUs is 16-32 KB, the actual amount of local memory used is dependent on the application's requirement, which is typically much smaller, and usually fits comfortably in the core's lower level caches (e.g., L1-D). By not distributing the work-group across cores, we significantly reduce communication by maintaining locality and largely avoiding cache coherence issues.

Since only one work-group can execute on a core at any given time, only one local memory is active on the core. Therefore, we allocate one local memory per core and reuse it for all the work-groups that execute on that core. This approach reduces off-chip memory bandwidth usage, efficiently utilizes caches, and saves expensive `malloc()` calls. In addition, to retain the local memory in the cache, x86 non-temporal instructions (e.g., `movntq`, `prefetchnta`) are used for other data accesses, which helps to effectively lock the local memory in the cache.

Figure 3(b) shows the details of mapping the local memory. We allocate a contiguous region of global memory to emulate the local memory. The local memory addresses accessed by the computation kernel are translated into offsets within this allocated region. The size of the local memory allocated is $m * local\_memory\_size$ per work-group, where m is the number of cores. The figure also shows guard pages [13] between all memories. These are allocated by our runtime system to help identify buffer overruns. Buffer overrun is one of the most common forms of memory corruption. This problem is exacerbated while executing computation kernels because many types of memories coexist simultaneously. Any access to a memory location inside the guard pages triggers the runtime to throw an exception.

4.2.2 *Mapping Private Memories*

Private memory refers to the memory that is private to each work-item (Section 2). This contains work-item context data (e.g., local/global/group ids/sizes, etc.) and kernel stack variables.

Private memories are actively used by one work-item at a time within a core. If the kernel has no synchronization primitives, we allocate only one private memory per core. For kernels having synchronization primitives, we allocate one set of private memories (corresponding to all the work-items in a work-group) per core and reuse them for all work-groups that get scheduled on the same core (Figure 3(b)). This approach has similar memory bandwidth and cache benefits discussed earlier for local memory(Section 4.2.1).

The total size of the private memory allocated in this scenario is $m * n * private\_memory\_size$, where m is the number of cores and n is the number of work-items per work-group. For typical work-group sizes (e.g., 128), the private memory could be very large and may not be able to fit in the private caches. In Section 4.3, we present several optimizations to efficiently pack the private memories in the cache hierarchy.

4.2.3 *Mapping Constant Memory*

Constant memory contains data that is constant throughout the execution of the kernel (Section 2). The data is read-only and cannot be modified during the execution of the kernel.

Since the data is not work-group dependent we allocate just one buffer in the global memory to map the constant memory (Figure 3(b)). The constant data accessed inside the kernel gets translated into offsets inside this global buffer. Because all the cores access this data, they get replicated in all the private caches (as seen in Figure 3(b)). In addition, these pages are marked read-only by the runtime system to catch potential bugs related to modifying constant data.

4.3 Low-level Optimizations

Most of the overheads in Twin Peaks stem from creating, storing, and switching between work-item contexts. We present several low-level optimizations to address these overheads. These optimizations adhere to the constraint of being non-invasive to the kernel source code. We divide the optimizations into four categories: Context creation, Cache-related, TLB-related, and *setjmp*()/*longjmp*() code optimizations.

4.3.1 *Optimizing Work-item Context Creation*

Creating work-item contexts involve setting the local and global IDs/sizes, group ID, number of dimensions, etc. These values need to be computed once per work-item. In addition, setting up the link area, pushing the arguments into registers, and calling the kernel are also performed once per work-item. These overheads could be significant, especially for kernels that do not have much computation (e.g., matrix transpose, etc.), and can be amortized by reusing work-item context information from earlier work-groups.

Once work-item contexts are created for a work-group, the subsequent work-groups can derive their work-item context information from earlier work-groups, rather than recreating it. This can be accomplished by using what we call *hot work-item stacks*. Since we reuse the work-item stacks for all the work-groups executing on the same core (Section 4.2.2), we could change only the requisite fields each time a new work-group executes, thus only partially updating the stack. For example, the local/global sizes, number of dimensions, etc. do not change; only fields such as global id, group id, etc. would need to be updated for each new work-group execution. The work-item stacks are thus *hot*, with most data already filled.

4.3.2 Cache-related Optimizations

Figure 4(a) shows the placement of the work-item stack data in the CPU cache for a baseline implementation prior to any optimizations. For illustration, we use a 32 KB 4-way set-associative L1-D cache. In main memory, the work-item stacks are allocated as contiguous chunks of memory in which each work-item stack is 8 KB in size. When a work-item executes on a core, the work-item data, described in the previous section, is brought into the cache, occupying one or more cache lines. Since the data in the work-items are aligned to the 8 KB boundary, they map to the same *set* of the cache as shown in Figure 4(a). This leads to new work-items kicking out the data belonging to the old work-items due to conflict misses. Most of the cache is not used leading to poor performance. This effect is exacerbated as the number of work-items increases; and hence, this implementation does not scale efficiently.

Figure 4(b) shows how we optimize the cache utilization. The data in the work-item stack are staggered so they end up in different sets of the L1-D cache. We accomplish this by offsetting stacks differently for each work-item. As shown in the figure, once we stagger the work-item stacks, the data spreads more evenly and utilizes the cache more efficiently.

To stagger the work-item stacks, we define a set of contiguous, equal size regions in the virtual address space. Each region contains one stack (e.g., of size 8 KB) at some offset. The choice of the *region size* and the *offset* inside the region together determine where exactly in the cache the data is mapped. It is important to choose these parameters carefully to ensure maximal utilization of multiple levels of the cache. In our implementation, we decided to choose these values such that for both L1 and L2 caches, an entire *way* of the cache is filled before filling the next way, eliminating conflict misses. To accomplish this, we use the following equations to determine region size and stack offset values:

1. $region\ size = 2 * max(L1Size/L1Assoc, L2Size/L2Assoc)$
2. $offset = max(CacheLineSize, 2^{ceil(log_2(WIDataSize))})$

The first equation shows how the region size is related to both L1 and L2 cache parameters. For small work-group sizes, the data fits in the L1 cache, so the L1 cache size and associativity determine performance. As the work-group size increases, the work-item stack data overflows into the L2 cache, so the L2 cache parameters must also be taken into account. By choosing *region size* to be the maximum of the one-way cache sizes of L1 and L2, we ensure that in both caches one-way is filled before filling the next way. The factor of two at the beginning is needed to fit the last work-item stack.

The second equation shows how the offset relates to the work-item data size. Typically, we would expect offset to equal work-item data size ($WIDataSize$). The equation additionally ensures that the offset is a multiple of the cache-line size that fits the work-item data. Furthermore, the equations ensure that both the region size and offset are powers of 2. This is useful for computing the addresses of the work-item data because all the multiplications can be converted into shifts for better performance.

The simple model works well for most cases. In practice, however, other factors can affect the placement of data in the work-item stacks. For example, as the kernel work-item state increases, more adjacent cache lines begin to fill. Hence, we need to change the offset depending on the amount of data in the stack. In our implementation, we carefully chose a generic value for WIDataSize (256 bytes) that works reasonably well for several applications. Furthermore, as the number of input/output data-arrays to the kernel increases, the work-item stacks can potentially get kicked out of the cache due to capacity misses. By tagging the data arrays as having little temporal locality (e.g., using non-temporal instructions) we can help preserve the work-item stacks in the data cache.

4.3.3 TLB-related Optimizations

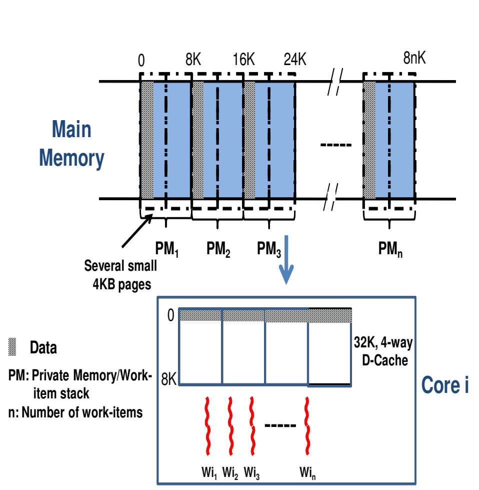
The analysis in Section 4.3.2 assumes that the mapping of each virtual address to its corresponding location in the cache is known by the user-level runtime system. However, this mapping is not known to the runtime system for caches, like the L2 cache, that are physically addressed rather than virtually addressed. In that case, the cache location of each virtual address is determined by OS page coloring policies. As a result, mapping to the cache may be sub-optimal, causing unexpected conflict misses that could hurt performance. To overcome this problem, as shown in Figure 4(a), we use huge pages (2-4 MB) to map the memory regions, ensuring that regions contiguous in virtual address space are also contiguous in the physical address space, and thus map to the L2 cache as expected.

Using huge pages has the additional advantage of imposing less pressure on D-TLB (typically 32-64 entries), avoiding potentially expensive page walks and cache pollution from page table entries. This reduces the required number of D-TLB entries significantly and is particularly useful for large work-groups, as shown in Section 5.4.

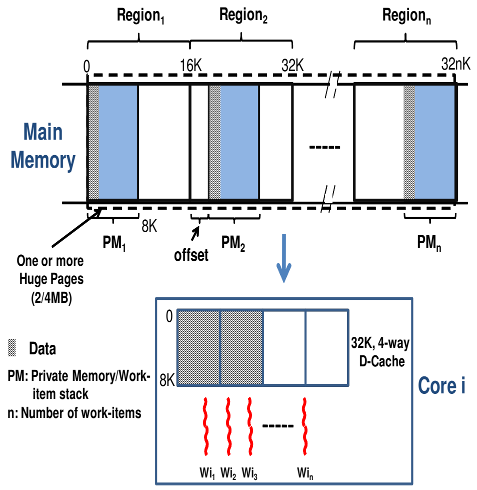
4.3.4 setjmp()/longjmp() Code Optimizations

The runtime uses *setjmp*() and *longjmp*() routines to save and restore register state when switching between work-item contexts during `barrier` execution. However, their standard library implementations contain code for a generic scenario (e.g., to respond to signals, etc.), which is extraneous for our purposes and lead to a high overhead.

We implemented our own light-weight *setjmp*()/*longjmp*() routines, which only save and restore a few registers. On executing *setjmp*() we only save the callee-save registers (e.g., rbx, r12-r15 in 64-bit x86), stack registers (e.g., rsp, rbp in 64-bit x86), and return address in the work-item stack, and restore them when *longjmp*() gets executed. This significantly reduces overhead, as shown later in Section 5.4.



(a) Baseline implementation



(b) Optimized implementation

Figure 4: Implementation of work-item stacks.

For the custom *longjmp*(), we took special care to optimize the code interaction with the hardware branch predictor. Modern micro-processors use a call/return stack for predicting the target of RET instructions: When a CALL instruction is decoded, the processor saves the address of the instruction following the CALL on a hardware stack structure. When a RET instruction is decoded, the processor pops the head of call/return stack as the predicted target for the RET.

Because the *longjmp*() routine jumps directly to a saved target and never returns to the caller, there is a danger that the call/return stack will become misaligned and subsequent RET instructions will be mispredicted. The standard library implementation of *setjmp*()/*longjmp*() suffers from this behavior. After significant experimentation, we settled on the following approach:

- Use an indirect JMP instruction inside *longjmp*() to jump to the saved target. This JMP can be easily predicted because nearly all of the *setjmp*() calls come from the same location, so the *longjmp*() is nearly always returning to the same point.
- Inline the *longjmp*() function. This eliminates the dangling call to *longjmp*() so the call/return stack does not become misaligned when *longjmp*() never returns.

Additionally, we optimized *longjmp*() so the compiler does not needlessly attempt to save or restore the callee-save registers. We desired a straight-line set of code to restore the callee-save registers from the saved environment and jump to the saved target.

5. EVALUATION

5.1 Experimental Set-up

Figure 5 showcases the overall framework to execute applications in Twin Peaks. The functions of the application that need to be accelerated are re-written in OpenCL (`App Kernel Files`) and the code calling these functions is changed to use the OpenCL API (`App Driver Files`). The rest of the application remains unchanged (`Other App Files`).

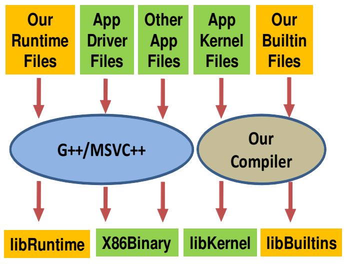


Figure 5: Overall Framework

The kernel files are then compiled using our compiler to generate x86 libraries (`libKernel`), whereas the other application files are compiled using native compilers (e.g., g++, MSVC++) to generate x86 binaries. Our runtime files are also compiled using the native compilers to generate a runtime library (`libRuntime`).

Here are the system configurations used: A 3.2 GHz Intel Xeon W5580 Nehalem processor (Quad-core) with 8 GB of DDR3 memory; a quad-socket 2.1 GHz AMD Istanbul system (totaling 24 cores) with 32 GB of DDR2 memory; an Apple MacBook Pro with a 2.8 GHz Intel Core2 Duo and a Nvidia 9400M (not used for this paper) and a Nvidia 9600M GT; an Intel Core2 Duo at 2.8 GHz with a high-end GPU, Nvidia GeForce GTX 280. The Nvidia driver we used was 190.89[1].

5.2 Applications and Characteristics

We adapt several OpenCL applications available in the AMD and Nvidia OpenCL SDK as examples. We have chosen nbody, boxfilter, matrix multiply, matrix transpose, BlackScholes, and Mandelbrot. The applications have varied characteristics. Some have high arithmetic intensity, like nbody and BlackScholes. Others, like matrix transpose, are largely bound by memory bandwidth. There is also a wide range of usage of local memory and `barrier` in the applications.

[1]For Blackscholes, we used 195.36 driver since the performance was an order of magnitude slower with the 190.89 driver.

For example, matrix multiply is implemented by having the work-group load in a tile of the matrices, issue a `barrier`, perform a sub-matrix multiply, followed by another `barrier`, all in a loop. The number of `barrier`s executed is directly related to the tile size and thus, work-group size. Other applications, like BlackScholes and Mandelbrot, have no communication between work items and the kernel is basically just loads and stores with a lot of math in between. An interesting trait of Mandelbrot is that the code has a lot of control flow relative to the other applications being tested.

Where possible, we performed simple manual vectorization of the kernels by making use of OpenCL float4/int4 (128-bit types) on the CPU to allow for both SSE ALU operation and SSE load/store. In general, this provided about a two-fold performance increase on the CPU, but was neutral or detrimental to the Nvidia GPU, so the GPU results below utilize scalar versions of the code whereas the CPU utilizes the simple vectorized versions. Beyond simple manual vectorization, no specific performance tuning was done on the applications for either the CPU or GPU, but it should be noted we are running GPU-friendly applications.

5.3 CPU vs. GPU

Figures 6(a)-(f) compare the kernel-only times between the Intel Nehalem, the Nvidia GeForce GTX 280, and the Nvidia 9600M for our applications. *Kernel-only time* is application execution time with all our optimizations enabled, neglecting initial data transfer to the accelerator and the return of the final results. All experiments on the CPU fully exercise our runtime system. Results without our optimizations are omitted due to lack of space, but generally show large slowdowns (at least 10X).

For the majority of applications, the GTX280 is notably faster than the CPU for large dataset sizes. Applications like BlackScholes, which are "poster-child" applications for the GPU, are much faster on mid-range GPUs than on a high-end CPU (4.3X). What is more interesting is the closeness of the CPU and Nvidia GeForce 9600M performance on some applications (e.g., nbody, Mandelbrot, matrix multiply). While part of this is likely related to the maturity of the OpenCL implementation, it shows that in systems with a GPU and CPU it can be better to use the CPU over the GPU, GPU over the CPU, or use both together as accelerators.

The application characteristics directly affect which applications perform well on each hardware. For example, matrix transpose (Figure 6(a)) is heavily memory bandwidth-limited and so across all datasets GPUs, with their faster memory hardware, outperform the CPU (e.g., for 8Kx1K matrix transpose GTX 280 is 10X faster than CPU). A second issue is the use of `barrier` inside this kernel. While `barrier` has negligible overhead on GPUs due to hardware support, the overhead is significantly higher on CPUs despite using highly optimized `barrier` implementation (Section 5.4) in software.

For applications with data-dependent control flow, CPUs typically perform fairly well compared to GPUs. This can be seen in Mandelbrot (Figure 6(e)) where the GPU performance degrades substantially compared to CPU performance for data sets that have more unpredictable control flow (scales: 2.9 and 1.8). For these datasets, CPU outperforms the mid-range GPUs by more than 70%.

The analysis so far has focused on kernel execution time alone. However, this neglects the cost of dispatching work to the GPU and receiving results. Even ignoring round-trip times and just concentrating on kernel-only times, for small datasets the performance of the high-end GPU and CPU is comparable (e.g., for nbody with 1024 elements, the three processors are only about 2% apart). When including round-trip times (i.e., including the time for moving data to the device, setting arguments, and all of the required API calls), the advantage of the GPU over the CPU is diminished and for image processing applications such as boxfilter (Figure 6(g)), the CPU can achieve better performance than a high-end GPU even up to medium-sized problems (e.g., for 512x512 images).

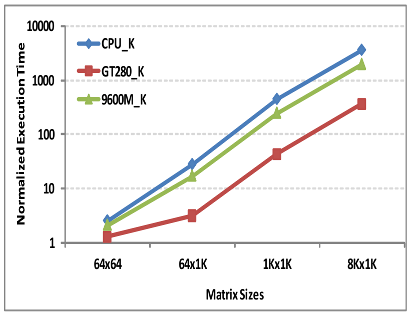
Despite our optimizations for the CPU, there is a lag in performance compared to high-end GPUs. This is because high-end GPUs typically have an order of magnitude higher FLOPS and memory bandwidths compared to CPUs. They also have specialized units to perform synchronization (e.g., barrier units, etc.). These characteristics make the GPUs very effective, especially in running data parallel applications with abundant parallelism and ALU operations. However, as GPGPU applications and GPU hardware become more general in the future, the performance differences between the two could change and a detailed evaluation of this is left for future work.

5.4 Overhead Evaluation

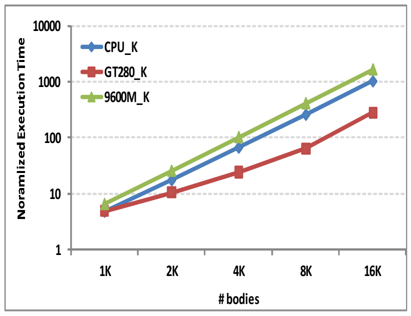
We next quantify the benefits due to our low-level optimizations described in Section 4.3. For evaluation, we use a simple kernel containing a `barrier` primitive, which forces a context switch between work-items. We compare against a baseline implementation using standard *setjmp*()/*longjmp* library routines, without any of our cache- or TLB-related optimizations. Our experiments were performed on the Nehalem system (Section 5.1) with 32 KB L1-D cache. The *region size* was set to 16 KB and *offset* to 128 bytes.

Figure 6(h) shows the context switch overhead from saving and restoring work-item stack data for varying numbers of work-items per work-group. The results show that using all of our optimizations (S+C+T) provides huge speedups compared to the base implementation (up to 7.1X). The work-item context switch overhead is less than 10 ns ($\sim$ 30 cycles) for most configurations, which is several orders of magnitude faster than typical context switch implementations (e.g., using *pthreads* is on the order of micro-seconds).

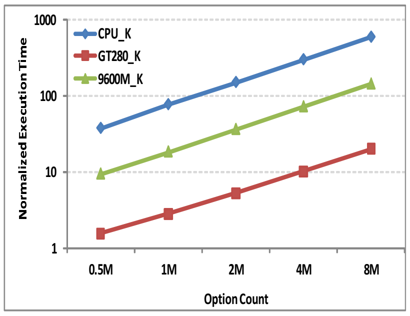
Our custom *setjmp*()/*longjmp*() code provides on average 2.3X improvement over the baseline implementation across different work-group sizes. This is a result of fewer instructions in the customized code, smaller D-cache footprint, and better interaction with hardware branch predictor, all of which are independent of the work-group size. Using cache-related optimizations in addition to *setjmp*()/*longjmp*() code optimizations eliminates conflict misses in the L1-D cache and is especially effective for medium-sized work-group sizes (e.g., 5.6X for work-group size of 128). This is because the baseline implementation has conflict misses even in L2 cache, whereas our efficient packing enables work-item stacks to continue to fit in the L1-D cache up to work-group size of 128. For large work-groups (512-1024), the cache optimizations continue to help, but provide diminishing returns due to L1-D capacity misses and TLB misses. Our TLB-related optimizations using huge pages further improve performance



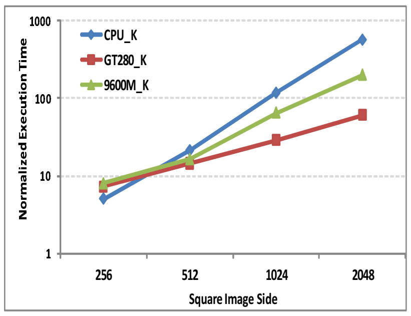
(a) Transpose



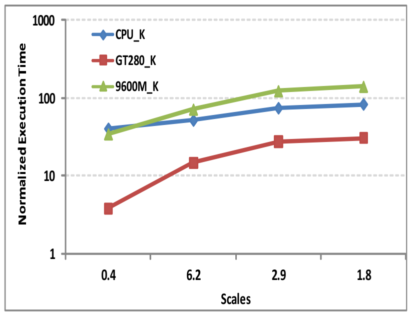
(b) nbody



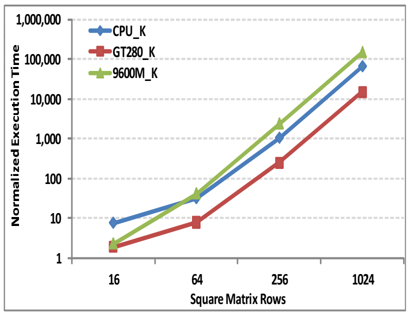
(c) BlackScholes



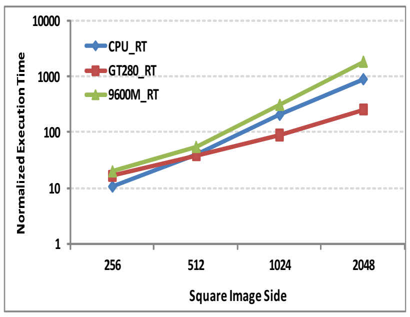
(d) BoxFilter



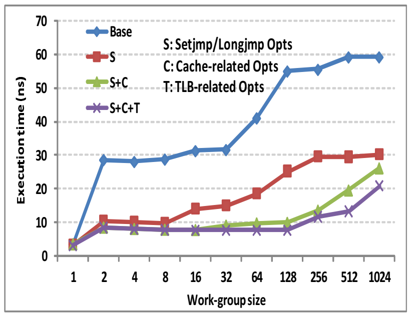
(e) Mandelbrot



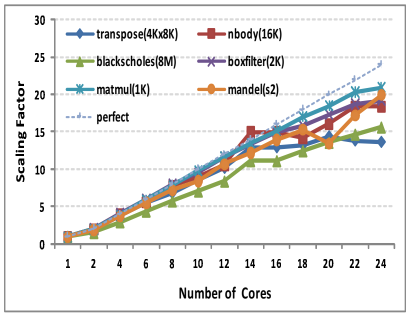
(f) MatrixMultiply



(g) Round Trip (BoxFilter)



(h) Overhead Evaluation



(i) Scaling

Figure 6: Twin Peaks Evaluation

(up to 7.1X) by reducing the number of TLB entries and reducing cache pollution due to page table walks.

The remaining overheads in Twin Peaks stem from dispatching the kernel to the cores and notifying the kernel completion. Our current implementation takes about 25 microseconds to enqueue the command in the device queue followed by about 6 microseconds to notify the host of completion. Tuning these paths is left for future work.

5.5 Scalability

Figure 6(i) shows the scaling behavior of the applications using Twin Peaks on a four-socket AMD Istanbul machine totaling 24 cores. As expected, applications with high arithmetic intensity like matrix multiply and nbody scale quite well, although not completely linearly. Other applications like matrix transpose top out quickly on scaling because of memory bandwidth limitations. Applications like nbody and Mandelbrot show more skewing in some parts of the curve because of NUMA effects on the machine. Our current implementation does not attempt to do any NUMA-aware allocation beyond the default policy from the OS.

6. RELATED WORK AND DISCUSSION

Several frameworks exist for exploiting the many cores of CPUs (e.g., traditionally OpenMP [18] and pthreads, and more recently Intel TBB [2], Streamware [9], etc.) and similarly, several frameworks have been proposed to exploit the GPUs for general-purpose compute (e.g., BrookGPU [5], CUDA [17], OpenCL [3], etc.). In this paper, by mapping kernels originally written for GPUs on multiple cores of the CPU, we advocate a unified programming model for heterogeneous computation. The underlying software system automatically handles distributing the computation kernels between the CPUs and GPUs.

Over the last few years, a couple of platforms have been proposed for heterogeneous computing in a CPU-Accelerator environment similar to Twin Peaks (e.g., EXOCHI [25], x86-Larrabee [20], x86-Cell [8], etc.). However, these platforms mainly target efficiently using the accelerators (e.g., Cell [11], Larrabee [21], etc.) and use the CPU cores only for serial execution. As opposed to our approach in which CPUs and GPUs are considered loosely coupled entities, EXOCHI [25] tightly couples the heterogeneous system by combining its task (*shred*) scheduler with the OS scheduler and extending the hardware to support shared virtual memory. x86-Larrabee [20], on the other hand, eliminates explicit memory management and serialization between x86 and Larrabee cores by using a shared address space, which is contended by both types of cores. Our work differs from these prior systems by targeting both CPU and GPU as accelerators, using low-level compilers to target their respective ISAs.

Recently, compiler techniques have been proposed to effectively execute CUDA kernels on CPUs (mCUDA [22]). The kernels are modified to execute on work-groups rather than work-items and the work-group state is stored in local arrays. The `barrier`s are eliminated by treating them as fission points for the work-item loops. This potentially leads to low `barrier` overhead. Our approach differs by not relying on advanced compiler techniques and leaving the kernel source code unchanged by moving the work-item scheduling to inside the runtime system. This enables use of the same code on both CPU and GPU, and reusing existing off-the-shelf compiler and debug tools with minor modifications. Because our optimizations are inside the runtime system and not in a static compiler, we can tailor our optimizations to the underlying processor configuration and factor in other dynamic information (e.g., cache-conscious data placement, etc.). Our approach also preserves the spatial locality for the private data of each work-item unlike the mCUDA approach. A detailed evaluation of these trade-offs is left for future work.

Our evaluation demonstrates that it is often beneficial to use CPU along with the GPU for maximal performance. Recent studies [16] show that when kernels are heavily tuned for each architecture, the CPU can achieve a significant fraction or even exceed the performance of the GPU. Thus, our approach of using both devices together would improve performance significantly. Our results show that these benefits hold for non-specialized kernels that preserve programmability important for code development and maintenance.

Using user-level threads to exploit fine-grain parallelism is not a novel concept [4, 24]. This technique has been used extensively in several parallel programming (e.g., [18],[4]) and networking frameworks (e.g., [24]) to avoid expensive context switches between threads. Our contribution is to apply these concepts in a heterogeneous computing framework taking advantage of the data-parallelism and simple control flow in computation kernels.

Several cache coloring, clustering [7], and data placement algorithms (for stack variables, global data, etc.) [6] are closely related to our approach of efficiently packing data in caches to eliminate conflict misses. Our contribution is to apply these ideas in a GPGPU framework, and develop a systematic methodology to efficiently pack work-item stacks in the CPU cache hierarchy.

setjmp()/longjmp() have been conventionally used for exception handling in C [12]. We use these constructs in a GPGPU framework to save and restore work-item stack data. Several compiler optimizations have been proposed for fast *setjmp()/longjmp()* execution [10]. For example, liveness analysis can be performed by the compiler to save and restore only registers that are live. Furthermore, the compiler can be forced to use only a subset of registers to minimize the amount of state saved and restored. These optimizations are complementary to the low-level code optimizations we propose and can be used to further reduce the *setjmp()/longjmp()* execution time in both GPGPU and other environments.

7. CONCLUSIONS AND FUTURE WORK

With the emergence of heterogeneous processors, in the near future both CPUs and GPUs should be treated as accelerator targets for data-parallel algorithms. To this end, we proposed, designed, and implemented Twin Peaks, a system that efficiently runs the same code on both multicore CPUs and GPUs. We showed that applications written originally for GPUs can be efficiently re-targeted for CPUs by mapping the thousands of GPU threads onto the CPU hardware contexts using LWTs, and mapping the GPU memories onto the CPU cache/memory hierarchy. We presented techniques to perform the mapping of kernels onto multicore CPUs without modifications to the kernel source code.

We designed and implemented Twin Peaks using the core set of vendor-independent OpenCL specification, and evaluated it comparing our CPU implementation against a GPU implementation, using GPUs with different target markets. Our results show that it is useful to have support for the entire heterogeneous system due to application characteristics that may favor one architecture over the other, as well as providing more aggregate computational power. Our studies also showed that our system scales well with CPU cores and has minimal overhead.

In the future, we plan to investigate automatically load balancing computation kernels between CPUs and GPUs using application characteristics and run-time profiling information. We also plan to extend our software platform to perform emulation studies exploring architecture trade-offs for future GPUs and heterogeneous platform designs.

8. ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to the anonymous reviewers for their insightful comments. Many thanks to our colleagues in the OpenCL team at AMD for their help in setting up the infrastructure and answering questions throughout this work. Thanks also goes to Phil Rogers, Roy Ju, Suki Samra, Nuwan Jayasena, and Lee Howes of AMD for their valuable feedback on the drafts of our paper. Finally, special thanks to Yoshio Turner of HP Labs for his continuous help in substantially improving the quality of our presentation.

9. REFERENCES

[1] GPGPU. www.gpgpu.org.
[2] Intel TBB. www.threadingbuildingblocks.org.
[3] OpenCL. www.khronos.org/opencl/.
[4] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. on Computer Systems*, 10(1), 1992.
[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[6] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of ASPLOS*, 1998.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of PLDI*, 1999.

[8] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the Cell broadband engine processor. In *Proceedings of Computing Frontiers*, 2008.

[9] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: Programming general-purpose multicore processors using streams. In *Proceedings of ASPLOS XIII*, 2008.

[10] G. Hoflehner, K. Kirkegaard, R. Skinner, D. Lavery, Y.-F. Lee, and W. Li. Compiler Optimizations for Transaction Processing Workloads on Itanium®Linux Systems. In *Proceedings of International Symposium on Microarchitecture*, 2004.

[11] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *Proceedings of HPCA*, 2005.

[12] International Standards Organization. *ISO/IEC 98899 - Programming Language C*, 1999.

[13] N. Joukov, A. Kashyap, G. Sivathanu, and E. Zadok. KeFence: An Electric Fence for Kernel Buffers. In *StorageSS*, 2005.

[14] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[15] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code generation and Optimization*, 2004.

[16] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of ISCA*, 2010.

[17] NVIDIA Corporation. CUDA Programming Guide 2.0, June 2008.

[18] OpenMP Architecture Review Board. *OpenMP Application Program Interface 3.0*, 2007.

[19] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5), May 2008.

[20] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. In *PLDI*, 2009.

[21] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH*, 2008.

[22] J. Stratton, S. S. Stone, and W. W. Hwu. M-CUDA: An efficient implementation of CUDA kernels on multicores. *Int'l Workshop on Languages and Compilers for Parallel Computing*, 2008.

[23] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proceedings of ASPLOS*, 2006.

[24] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Netw.*, 1(5), 1993.

[25] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of PLDI*, 2007.