Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor

Henry Wong[1]*, Anne Bracy[2], Ethan Schuchman[2], Tor M. Aamodt[1], Jamison D. Collins[2], Perry H. Wang[2], Gautham Chinya[2], Ankur Khandelwal Groen[3], Hong Jiang[4], Hong Wang[2]

1: Dept. of Electrical and Computer Engineering, University of British Columbia
2: Microarchitecture Research Lab, Microprocessor Technology Labs, Intel Corporation
3: Digital Enterprise Group, Intel Corporation
4: Graphics Architecture, Mobility Groups, Intel Corporation

ABSTRACT

Moore's Law and the drive towards performance efficiency have led to the on-chip integration of general-purpose cores with special-purpose accelerators. Pangaea is a heterogeneous CMP design for non-rendering workloads that integrates IA32 CPU cores with non-IA32 GPU-class multi-cores, extending the current state-of-the-art CPU-GPU integration that physically "fuses" existing CPU and GPU designs. Pangaea introduces (1) a resource repartitioning of the GPU, where the hardware budget dedicated for 3D-specific graphics processing is used to build more general-purpose GPU cores, and (2) a 3-instruction extension to the IA32 ISA that supports tighter architectural integration and fine-grain shared memory collaborative multithreading between the IA32 CPU cores and the non-IA32 GPU cores. We implement Pangaea and the current CPU-GPU designs in fully-functional synthesizable RTL based on the production quality RTL of an IA32 CPU and an Intel GMA X4500 GPU. On a 65 nm ASIC process technology, the legacy graphics-specific fixed-function hardware has the area of 9 GPU cores and total power consumption of 5 GPU cores. With the ISA extensions, the latency from the time an IA32 core spawns a GPU thread to the time the thread begins execution is reduced from thousands of cycles to fewer than 30 cycles. Pangaea is synthesized on a FPGA-based prototype and runs off-the-shelf IA32 OSes. A set of general-purpose non-graphics workloads demonstrate speedups of up to 8.8×.

Categories and Subject Descriptors

C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Heterogeneous (hybrid) systems*

General Terms

Design, Performance

*Work performed while at Intel.

1. INTRODUCTION

As Moore's Law pushes for a more rapid pace of silicon development and even higher degree of on-die integration, the number of cores in future multi-core designs will continue to increase. As the microprocessor industry rapidly marches into the era of multi-core design, the future generation of multi-core processors will essentially become an integration platform with not only numerous cores, but also different types of cores varying in functionality, performance, power, and energy efficiency [9]. Fundamentally, ultra low EPI (Energy Per Instruction) cores are essential to scale multi-core processor designs to incorporate a large number of cores. One approach to improving EPI by an order of magnitude is through heterogeneous multi-core designs, which have a small number of large, general-purpose cores optimized for instruction-level parallelism (ILP) and many more special-purpose cores optimized for data-level parallelism (DLP) and thread-level parallelism (TLP). Such a multi-core processor offers opportunities for non-graphics application software and usage models [1, 25, 31, 32, 33, 38] to aggressively exploit the combination of ILP, DLP and TLP.

In this paper we present Pangaea, a synthesizable design of a heterogeneous chip multiprocessor (CMP) that integrates IA32 CPU cores with GPU multi-cores. Architected to support general-purpose parallel computation, Pangaea goes beyond the current state-of-the-art CPU-GPU integration that physically "fuses" an existing CPU design and an existing GPU design on the same die. In Pangaea, new enhancements are introduced to both the CPU and GPU to support tighter architectural integration, improved area and power efficiency, and scalable modular design. On the CPU side, a three-instruction extension to the IA32 ISA supports a fly-weight communication mechanism between the CPU and the GPU and a fine-grain shared memory collaborative multithreading environment between the IA32 CPU cores and the GPU multi-cores. This ISA enhancement allows an IA32 thread to directly spawn user-level threads to the GPU cores, bypassing most of the legacy graphics specific fixed-function hardware (*e.g.*, input assembler, vertex shader, rasterization, pixel shader, output merger [26]) found in a modern GPU design. This can achieve a two-order of magnitude reduction in thread spawning latency. On the GPU side, a state-of-the-art existing GPU design (Intel GMA X4500 [15]) is rearchitected to significantly reduce the fixed-function hardware, which is traditionally dedicated to support 3D-specific graphics processing. The legacy front-end is replaced with a small FIFO controller that can buffer

and dispatch GPU threads spawned by the IA32 CPU. The legacy back-end is replaced by sharing the memory hierarchy between the IA32 CPU and the GPU multi-cores. The removal of the legacy fixed-function hardware can result in area savings (on a 65 nm process) equivalent to nine additional GPU cores (of five hardware threads each) and power savings equivalent to five GPU cores.

This paper makes the following contributions:

- We describe the architecture support and microarchitecture reorganization of both CPU and GPU in Pangaea to achieve tighter architecture integration and power and area efficiency of a heterogeneous CMP design.
- We detail a fully functional synthesizable implementation of a Pangaea design, based on production quality RTL from an ILP optimized IA32 core and the GMA X4500 GPU.
- We present an in-depth analysis of architectural tradeoffs between the Pangaea design and a state-of-the-art design that physically fuses existing CPU and GPU on the same die.
- We report significant performance gains for a set of media and non-graphics parallel applications by employing Pangaea to harvest ILP, DLP and TLP, achieving speedups of up to 8.8×.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 provides a background on baseline GPU architecture. Section 4 introduces the architectural enhancements to the IA32 CPU and the microarchitectural reorganization of the X4500 GPU to support tighter architectural integration. Section 5 details the implementation of Pangaea and assesses the key architectural tradeoffs in terms of power and area savings compared to the state-of-the-art CPU-GPU design with physical fusion. Section 6 evaluates the performance of a set of general-purpose applications on a Pangaea hardware prototype on an FPGA-based emulator. Section 7 concludes.

2. RELATED WORK

We adopt the distinction between *asymmetric* and *heterogeneous* multi-core designs from related work [12, 38]. All cores in an *asymmetric* multi-core design are of the same ISA but differ microarchitecturally. In a *heterogeneous* multi-core design, some cores feature different ISAs in addition to microarchitectural differences. Prior work on multi-core architectures has demonstrated significant benefits for both power/performance and area/performance efficiency [3, 4, 8, 10, 12, 19, 20, 21, 27, 28]. However, those studies primarily focus on asymmetric rather than heterogenous multi-core design.

Heterogeneous multi-core designs integrate cores of different ISAs and functionalities and can potentially lead to even further improvement in power/area/performance efficiency. IBM Cell's heterogeneous architecture [18] offers a mix of execution elements optimized for a spectrum of functions. Applications execute on this system, rather than a collection of individual cores, by partitioning the application and executing each component on the most appropriate execution element. The *exoskeleton sequencer* (EXO) architecture [38] presents heterogeneous cores as MIMD function units to the IA32 CPU and provides architectural support for shared virtual memory, ensuring efficient data sharing across the heterogeneous execution elements.

Recently, both AMD and Intel have made public announcements on their upcoming mainstream heterogeneous processor designs for the 2009-10 timeframe. These processors will be on-die integrations of the IA32 CPU and their respective GPUs, which are traditionally found on the chipset or in discrete GPU cards. The so-called *fusion* integration physically connects existing CPU and GPU designs and supports some level of cache sharing between them, while the designs themselves remain unchanged. Although the integrated GPU is intended to run the legacy graphics software stack, there has been growing interest in harvesting such heterogeneous multi-core processors to accelerate non-graphics applications. Furthermore, there have been extensive efforts to provide programming model abstractions and runtime support to ease the otherwise daunting task for programmers to use heterogeneous multi-cores [6, 31, 32, 33].

Although heterogeneous integration is key to Pangaea, Pangaea is different than fused designs in that it supports a tighter-coupled integration through lightweight user-level interrupts. Bracy *et al.* discuss these lightweight user-level interrupts and utilize existing coherency logic to provide simple, preemptive, low-latency communication between cores [5]. Many other microarchitectures also support preemptive communication [2, 7, 13, 14, 22, 24, 29, 35, 37].

3. BACKGROUND

This section provides some necessary background on GPU architecture and defines terminology that will be used in the following sections. Figure 1 depicts an architectural organization of a modern GPU. It consists of three major components (from left to right):

- **Front-end:** a graphics-specific pipeline ensemble of fixed-function units, each corresponding to a certain phase of the pixel and vertex processing primitives, *e.g.*, command streamer, vertex fetcher, vertex shader, clipper, strip/fan, windower/masker, roughly in correspondence to DirectX's input assembler, vertex shader, rasterization, pixel shader, and output merger [26], respectively. The front-end translates graphics commands into threads that can be run by the processing cores.
- **Processing multi-cores:** hereafter referred to as Execution Units (EU). This is where most GPU computations are performed. Each EU usually consists of multiple SMT hardware threads, each implementing a wide SIMD ISA. In the GMA X4500, each thread supports 8-wide SIMD operations.
- **Back-end:** consists of graphics-specific structures like render cache, etc., which are responsible for marshalling data results produced by the EUs back to the legacy graphics pipeline's data representation.

Non-graphics communities are understandably interested in harvesting the massive amount of thread level and data-level parallelism offered by the EU to accelerate general-purpose computation, for which the graphics specific hardware front-end and back-end are largely overhead. The GPU is managed by device drivers that run in a separate memory space from applications. Consequently, communication between an application and the GPU usually requires device driver involvement and explicit data copying. This results

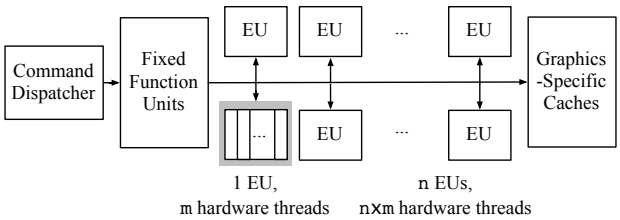

Figure 1: Organization of the Intel GMA X4500.

in additional latency overhead due to the software programming model.

Pangaea assumes the EXO execution model that supports user-level shared memory heterogeneous multithreading and an integrated programming environment such as *C for Heterogeneous Integration* (CHI) [38] that can produce a single fat binary consisting of multiple code sections of different instruction sets that target different cores. The focus of our study of the Pangaea design space is to investigate architectural improvements beyond the physical on-die fusion of existing CPUs and GPUs and to assess the power/area/performance efficiency using production quality RTL for both an IA32 CPU design and a modern multi-core multithreaded GPU design. The proposed architecture enhancements to both the CPU and GPU can enable much more efficient software management of parallel computation across heterogeneous cores. By minimizing resources dedicated solely to 3D-specific graphics processing, significant improvements in area and power efficiency can be achieved.

4. PANGAEA ARCHITECTURE

This section introduces Pangaea's architecture enhancements to the IA32 CPU and architectural reorganization of the X4500 GPU to support tighter architectural integration.

4.1 CPU-GPU Integration

Pangaea is a novel CPU-GPU integration architecture design that removes the legacy graphics front-end and back-end of the traditional GPU design to enhance general-purpose (non-graphics) computation. With architectural support for shared memory and a fly-weight user-level inter-core communication mechanism, Pangaea provides a tightly-coupled architectural integration of CPU and GPU EUs to more efficiently support collaborative heterogeneous multithreading between GPU threads and CPU threads.

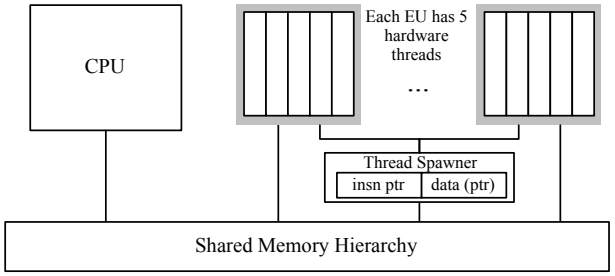

Figure 2: Pangaea: Integrated CPU-GPU without Legacy Graphics Front- and Back-End.

Figure 2 shows a high level diagram of the Pangaea architecture. Pangaea physically couples a set of EUs directly with each CPU via an agile thread spawning interface, but without the legacy graphics front-end and back-end. Each EU works as a TLP/DLP coprocessor to the CPU. This mechanism allows for a more power and area efficient design, which maximizes the utilization of the massively-parallel ALUs packed in the EUs.

The shared cache supports the collaborative multithreading relationship (peer-to-peer or producer-consumer) between the CPU and EUs. Both CPU and EU cores fetch their instructions and data from the shared memory. The common working sets between CPU threads and EU threads benefit from the shared cache. Enabling a coherent shared address space also make it easier to build a simple communication mechanism between the CPU and EU cores. The communication mechanism between the CPU and EU cores is introduced as an ISA extension.

In Panagea, the EUs appear as additional function units to which threads can be dispatched from the CPU. The CPU is responsible for both assigning and monitoring the GPU's work. The CPU can receive results from the GPU as soon as they are ready and schedule new threads to the GPU as soon as EU cores become idle. Inter-processor interrupts (IPIs) have often been leveraged for cross-core communication, but they introduce performance overheads that are not appropriate in the intended fine grained multithreaded environment of Pangaea. Instead of using IPIs, Pangaea leverages simple and fast *user-level interrupts* (ULIs) which are discussed in the next section. A fast mechanism is desirable as the EU threads are short lived and each EU thread processes only a small amount of data. The CPU spawns a large number of threads to increase the resource utilization of the EUs which are optimized for DLP and TLP.

Sections 4.2 and 4.3 describe the IA32 ISA extension that supports a user-level communication mechanism between the CPU and EUs. Section 5 presents an analysis of the power and area efficiency of Pangaea versus the fusion design.

4.2 ISA Extension for User-level Interrupts

Pangaea introduces a three-instruction IA32 ISA extension that supports communication between heterogeneous cores. The three instructions are **EMONITOR**, **ERETURN**, and **SIGNAL**. The communication mechanism is as follows.

A *scenario* is a particular machine event that may occur (or *fire*) on any core. Example scenarios include an invalidation of a particular address, an exception on an EU, or termination of a thread on an EU. **EMONITOR** allows application software to register interest in a particular scenario and to specify a user-defined software handler to be invoked (via user-level interrupt (ULI)) when the scenario fires. This scenario-to-handler mapping is stored in a new form of user-level architecture register called a *channel*. Multiple channels allow multiple scenarios to be monitored simultaneously.

When the scenario fires, the microcode handler disables future ULIs, flushes the pipeline, pushes the current interrupted instruction pointer onto the stack, looks up the instruction pointer for the user-defined handler associated with the channel, and redirects program flow to that address. The change in program control flow is similar to what happens when an interrupt is delivered. The key difference is that the ULI is handled completely in user mode with minimal state being saved/restored when the user-level interrupt handler is invoked.

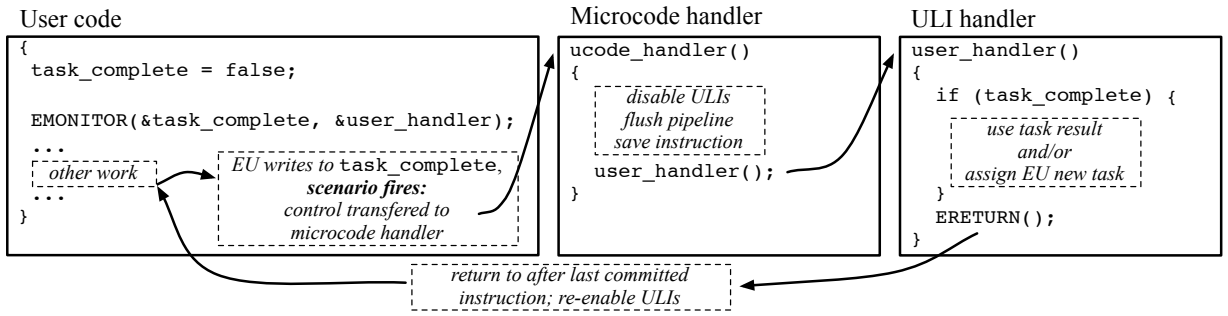

Figure 3: Example of User-Level Interrupt (ULI).

ERETURN is the final instruction of the user-defined handler. It pops the stack and returns the processor to the interrupted instruction while re-enabling ULIs.

Figure 3 shows an example of using ULIs. On the left and right is code provided by software. In the middle is the microcode handler. Software activates a channel by executing the EMONITOR instruction, registering its interest in invalidations to the `task_complete` variable and providing the handler that should be called when the invalidation occurs. In this example—one of many possible usage models—the user code spawns a task to the EU and then performs other work. When the EU completes its task, it writes to the variable `task_complete` which is being monitored and the scenario fires. The microcode handler invokes the user-defined interrupt handler. The user's handler can use the result of the EUs immediately and/or assign the EU another task. The user's handler ends with ERETURN. The program then returns to the instruction just after the last committed instruction prior to the interrupt and the user code continues its work. Other usage models might have the EU's task completion affect the user code's behavior upon returning from the interrupt.

To spawn a thread to the EU, the CPU stores the task (including an instruction pointer to the task itself and a data pointer to the possible task input) at an address monitored by the *Thread Spawner*, shown in Figure 2. The Thread Spawner is directly associated with the thread dispatcher hardware on the EUs. The CPU then executes the **SIGNAL** instruction—the third ISA extension—to establish the signaling interface between the CPU and EU.

As in related work [12], the SIGNAL instruction is a special store to shared memory that the CPU uses to spawn EU threads. Using SIGNAL, the EUs can be programmed to monitor and snoop a range of shared addresses similar to SSE3's MONITOR instruction [17]. Upon observing the invalidation caused by the CPU's SIGNAL, the Thread Spawner loads the task information from the cache-line payload. The Thread Spawner then enqueues the EU thread into the hardware FIFO in the EU's thread dispatcher, which binds a ready thread to a hardware thread core (EU), and then monitors the completion of the thread's execution.

Upon recognizing the completion of a thread, the Thread Spawner performs a final store (here, writing to `task_complete`) that results in the scenario firing, as shown in Figure 3. The CPU thread can schedule and dispatch more EU threads in response (not shown).

Because the thread spawning and signaling interface between the CPU and EUs leverages simple loads and stores, it can be built as efficiently as regular cache coherence with very low on-chip latencies.

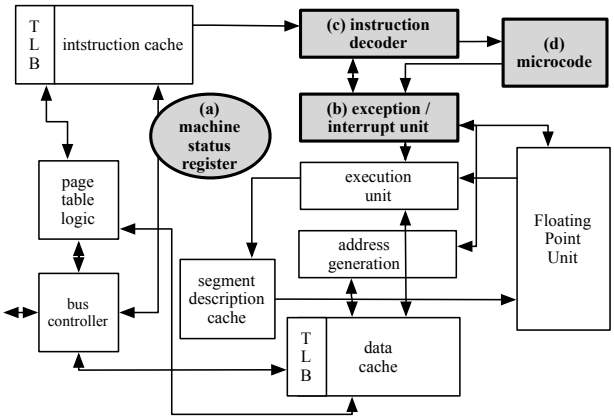

Figure 4: IA32 CPU Block Diagram. Shaded blocks indicate modifications to support ULI.

A similar fly-weight signaling mechanism is also used in hardware to implement the exoskeleton *proxy execution* mechanism [38]. In Pangaea the IA32 CPU handles exceptions and faults incurred on the GMA X4500 cores for *address translation remapping* and *collaborative exception handling* using proxy execution. These mechanisms are essential to support a shared virtual address space between the IA32 CPU and the GMA X4500 cores.

Figure 4 shows the microarchitecture block diagram of the IA32 core used for this study. The darkened units were modified to support ULIs. First, new registers are introduced to support multiple channels (shown in Figure 4(a)). Each channel holds a mapping between a user handler's starting address and the associated ULI scenario. A register is used to hold a blocking bit which specifies if ULIs are temporarily disabled. Since the channel registers store application specific state, these registers need to be saved and restored across OS thread context switches along with any active EU thread context. Existing IA32 XSAVE/XRSTOR instruction support can be modified to save and restore additional state across context switches [16]. These registers can be read and written under the control of microcode.

The exception/interrupt unit (shown in Figure 4(b)) handles all interrupts and faults, and determines whether instructions should be read from the instruction decoder or the microcode. This unit is modified to recognize ULI scenarios. A new class of interrupt request, **ULI-YIELD**, triggers at the firing of a scenario and requests a microcode

control-flow transfer to the ULI microcode handler. This interrupt is handled in the integer pipeline. All state logic associated with the ULI-YIELD, determining when an ULI-YIELD should be taken, and saving pending ULI-YIELD events is found here. Because the ULI-YIELD request has the lowest priority of all interrupt events, ULIs do not interfere with traditional interrupt handling. Once the ULI-YIELD has priority, the exception/interrupt unit flushes the pipeline and jumps to the ULI microcode handler. If multiple channels are implemented, when multiple instances of ULI-YIELD interrupts simultaneously occur, lower indexed channels have higher priority over higher indexed channels.

The instruction decoder (shown in Figure 4(c)) is responsible for decoding instructions and providing information needed for the rest of the CPU to execute the instruction. The decoder is modified to add entry points for the new IA32 instructions EMONITOR, ERETURN and SIGNAL. These changes map the CPU instructions to the corresponding microcode flows in the microcode. The microcode (shown in Figure 4(d)) is modified to contain the ULI microcode handler and the microcode flows for EMONITOR, ERETURN and SIGNAL. The ULI microcode handler flow saves the current instruction pointer by pushing it onto the current stack, sets the blocking bit to prevent taking recursive ULI events, and then transfers control to the user-level ULI handler. The EMONITOR microcode flow registers a scenario and the user handler instruction pointer in the ULI channel register. The ERETURN microcode flow pops the saved instruction pointer off the stack, clears the blocking bit and finally transfers control to the main user-code where it starts re-executing the interrupted instruction.

In Pangaea, we introduce a ULI scenario, **ADDR-INVAL**, which architecturally represents an invalidation event incurred on a range of addresses, which resembles the behavior of a user-level version of the MONITOR/MWAIT instruction in SSE3. Unlike MWAIT [17], when the IA32 CPU in Pangaea snoops a store to the monitored address range, the CPU will activate the ULI microcode handler and transfer program control to the user-level ULI handler. To implement a producer-consumer workload using a traditional polling model, the producer regularly reads a designated semaphore address, checking for a value indicating that the consumer has completed its task. With the ADDR-INVAL ULI, the producer sets up a ULI channel to monitor future asynchronous updates to a semaphore and then proceeds to work on other tasks in parallel while the hardware performs the monitoring. When a consumer writes to the semaphore indicating task completion, this triggers the ADDR-INVAL ULI scenario and the producer is informed of this asynchronously. This ULI scenario is used for the signaling between the IA32 CPU cores, the thread spawner, and the GMA X4500 EUs by leveraging the existing cache coherence protocol support, which is much more efficient than traditional IPI mechanisms that are sent via the interrupt controller. The address range that needs to be monitored is set up using the SIGNAL instruction which directly communicates with the thread spawner.

4.3 User-level Interrupt Handler

Certain precautions need to be taken in designing and writing a user-level interrupt handler as it runs in the context of the monitoring software thread. The monitoring software thread is the thread that executes the EMONITOR instruction and monitors the execution of the EU threads. The monitoring software thread runs on the IA32 CPU concurrently with the EU threads that run on the GPU. The user-level interrupts are delivered in the context of the monitoring thread without operating system intervention and they pre-empt the execution of the monitoring thread. Due to the pre-emptive nature of the user-level interrupt the user-defined interrupt handler should avoid attempting to acquire locks or invoke system calls that acquire locks as the monitoring thread may be executing in the middle of a critical section when it is pre-empted to execute the user-level interrupt handler. If the user-level interrupt handler attempts to acquire the same lock that has already been acquired then a deadlock results. An ideal user-level interrupt handler does not need to be complex or invoke system calls as the user-level interrupt handler is responsible for dispatching a new set of threads to the EU or resolving exception conditions for the EU threads to make forward progress. The user-level interrupt handler usually sets flags that are checked by the monitoring thread when exception conditions have to be resolved. An example of this is shown in Figure 3.

The user-level interrupt serves as a notification mechanism of a exception that needs to be resolved for the EU threads to make forward progress or to inform the monitoring thread about the termination of a group of EU threads. The monitoring thread can resolve the exception condition and then resume the EU thread at a later point in time. The interrupt mechanism is optional and the monitoring thread can always use the polling mechanism to poll on the status of the EU threads by reading the channel registers which contain the scenarios that are being monitored as well as the current status of the scenario. The monitoring thread may attempt to just poll the channel registers when there is no more concurrent work to do or there is a need for a barrier synchronization between the monitoring thread and the EU threads.

The user-level interrupt handler is also responsible for saving and restoring the register state that is not saved/restored by the microcode handler. Since the user-level interrupt handler runs in the context of the monitoring thread it is safe to assume that the code segment or stack segment registers do not change after the monitoring thread executes the EMONITOR instruction as segmentation is not normally used for virtual memory management in modern operating systems. The only exception to this assumption is when the monitoring thread is running in compatibility 32-bit mode under a wrapper on a 64-bit operating system. A change in the code and stack segment occurs during transition from compatibility 32-bit mode to 64-bit mode in user space. The microcode handler is modified to suppress any user-level interrupts to be delivered when the code segment values do not match what was recorded when the EMONITOR instruction is executed. The delivery of the user-level interrupt is frozen for the duration of execution in 64-bit user mode. The EU threads that do not need to report any exceptions or terminate can continue to execute even when the monitoring thread is executing in 64-bit user mode. When the monitoring thread returns from executing in 64-bit mode back to 32-bit mode the microcode detects the pending user-level interrupt and invokes the user-level interrupt handler. This simple mechanism is sufficient to allow 32-bit applications to continue to work when migrated to run on a 64-bit operating system that runs the application in compatibility mode.

Parameter	Configuration
IA32 CPU	2-issue, in-order, 4-wide SIMD capabilities, optimistically giving 4x speedup over non-SIMD
CPU-only L1 Caches	8KB 2-cycle access write-back data cache, 8KB Instruction cache, 2-way set associative
EUs	2 EUs, 5 hardware threads each, 8-wide SIMD ISA, 4-wide SIMD execution unit, 0 latency thread switch, 64 256-bit registers per thread. Same clock speed as CPU
EU-only Instruction Cache	4KB shared instruction cache, 4-way set associative
Shared L2 Cache	256KB shared with EU for EU instructions and data, 32-bits/clock bandwidth, configurable access latency by EU (2 to >100 cycles)

Table 1: One Pangaea Prototype Configuration that fits one Xilinx Virtex-5.

The user-level interrupt mechanism provides a simple, fast and efficient core-to-core communication mechanism without having to introduce new interrupts that need device driver management or major changes to the interrupt controller.

5. PANGAEA IMPLEMENTATION

To assess its power/area/performance efficiency, we implement a synthesizable design of Pangaea using production quality RTL for both an IA32 CPU design and a modern multi-core multithreaded GPU design. This section describes the Pangaea implementation and prototyping on an FPGA. We also discuss the power/area efficiency analysis. Section 6 presents a performance evaluation of Pangaea using a set of non-graphics parallel workloads.

5.1 Pangaea's Synthesizable RTL Design

We build a prototype of the proposed Pangaea architecture by implementing synthesizable RTL of a fully functional single-chip heterogeneous CMP consisting of an IA32 CPU and GMA X4500 multi-cores (*i.e.*, EUs). The CPU used in our prototype (shown in Figure 4) is a production two-issue in-order IA32 processor equivalent to a Pentium with a 4-wide SSE enhancement. The EU is derived from the RTL for the full GMA X4500 production GPU. We configure our RTL to have two EUs, each supporting five hardware threads. While the baseline design is the physical fusion of the existing CPU and full GPU, in Pangaea much of the front-end and back-end of the GPU have been removed, keeping only the EUs and necessary supporting hardware. By attaching the EU onto the memory hierarchy of the CPU (sharing of the last-level cache), we no longer need to duplicate the hardware required for accessing and caching memory on the GPU. This prototype design provides means to adjust various configuration parameters, including capacities and access latencies for the memory hierarchy, number of EUs and number of hardware threads per EU. The RTL can be synthesized to either ASIC or FPGA targets.

Table 1 shows one particular design that can be synthesized to a Xilinx Virtex-5 XC5VLX330 FPGA using Synplify Pro 9.1 and Xilinx ISE 9.2.03i. Table 2 shows the resource usage as reported by Synplify Pro for our FPGA prototype. The IA32 core is larger than one EU, taking up approximately 24% of the 207,360 available FPGA 6-LUTs. As the table shows, the EU subsystem with 2 EUs is less than double the area IA32 CPU in our prototype. The impact from the modifications to the CPU to support ULIs (not shown) is negligible—on the order of 50 LUTs. The logic added to support the thread spawner (not shown) is only 2% of a single EU.

	LUTs	Registers	Block RAMs	DSP48 blocks
IA32 CPU	50621	24518	118	24
EU Subsystem	84547	36170	67	64
Other	1604	591	91	2

Table 2: Virtex-5 FPGA Resource Usage for the Pangaea configuration in Table 1.

The prototype can fit just two EU cores and occupies 66% of the 6-LUTs available on the Virtex-5 LX330. Larger configurations consisting of multiple EUs have been evaluated in RTL simulation. For parallelizable workloads evaluated in this paper (see Section 6), we expect throughput performance to scale roughly with the number of EUs. The critical timing path within the EU allows us to clock the Pangaea prototype system at a maximum of 17 MHz without any special tuning. Similar to [23], the FPGA system on chip is mounted on an adapter that sits in a standard Intel Pentium motherboard with 256MB DRAM. Because of the critical path in our FPGA prototype, we underclocked the motherboard to 17 MHz, down from the original 50 MHz. Note that by underclocking the entire board, the relative speeds between all parts of the system remain unchanged, including processor, RAM and cache. The main advantage of an FPGA prototype compared to RTL simulation is the ability to execute orders of magnitude faster. Even at 17 MHz, the FPGA emulation speed is quicker than fast IA32 platform functional simulators such as SoftSDV [36]. This allows our prototype to run off-the-shelf operating system software, including Windows XP and Linux, and execute fat binaries of heterogeneous multithreaded programs produced by frameworks similar to EXOCHI [38].

5.2 Area Efficiency Analysis

To assess the area efficiency of Pangaea versus the baseline fusion design, we use the area data collected from the ASIC synthesis of the baseline GMA X4500 RTL code. This ASIC

synthesis result corresponds to a processor built on a 65 nm process. The left column of Table 3 shows the area distribution of a fusion-styled design with two EUs, including both legacy graphics front- and back-ends. The total area used for graphics-specific legacy hardware (the front- and back-ends) is 81%—the equivalent of over nine EUs. Even if this cost were amortized across more EUs, the overhead remains significant. With 32 EUs, for example, the front- and back-ends still occupy 23% of the chip area.

	2-EU GPU	2-EU Pangaea
Processing	17%	94%
Thread Dispatch	1%	5%
Front-End	34%	–
Memory Interface	1%	–
Back-End	47%	–
Interfacing Logic	–	1%

Table 3: Area distribution of two-EU systems.

The right column of Table 3 depicts the distribution of chip area of the Pangaea configuration shown in Table 1. Unlike the two EU GPU in a fusion design, a two EU Pangaea design has much higher area efficiency. A majority (94%) of the area is used for computation. The extra hardware added to implement the thread spawner and its interface to the interconnection fabric is minimal, amounting to 0.8% of the two-EU system, and easily becomes negligible in a system with more EUs. This significantly reduced overhead allows us to efficiently use EUs as building blocks for DLP/TLP and couple them with the IA32 cores in a heterogeneous multi-core system.

5.3 Power Efficiency Analysis

Table 4 shows the total power consumption distribution for a two-EU GPU including both dynamic power and leakage power. Like our area analysis, we use power data based on ASIC synthesis. Most noticeable is that the legacy graphics front-end contributes a lower proportion of power relative to its area. This is mainly due to extensive use of clock-gating that results in reduced dynamic power consumed by the front-end, since only the fixed-functions in the front-end that relate to the current task are switched on. We estimate that removing the legacy graphics-specific hardware would result in the equivalent of five EUs of power savings.

Processing	29%
Thread Dispatch	0.5%
Front-End	14%
Memory Interface	0.5%
Back-End	57%

Table 4: Power distribution of a two-EU GPU.

Because of the reduced front-end power, the power overhead for keeping the front-end and back-end in the design is lower than the area overhead. Despite that, the power overhead is still significant for a large number of EUs per GPU, and prohibitive for a small number of EUs. For a two-EU Pangaea (not shown), the power increase due to the thread spawner and related interfacing hardware is negligible compared to the amount of power saved by removing the legacy graphics specific front- and back-ends of the two-EU GPU.

5.4 Thread Spawn Latency

Table 5 compares the latency of spawning a thread in fusion CPU-GPU integration versus Pangaea. The thread spawn latencies are collected from RTL simulations of the two configurations. The latencies reported are for the hardware only. For the baseline GPGPU case, thread spawn latency is measured from the time the GPU's command streamer hardware fetches a graphics primitive from the command buffer until the first EU thread performing the desired computation requests is scheduled on an EU core and performs the first instruction fetch. For the Pangaea case, we measure the time from when the IA32 CPU writes the thread spawn command to the address monitored by the thread spawner set up by the SIGNAL instruction, until the thread spawner dispatches the thread to an EU core and the first instruction is fetched. The latency in the GPGPU case is approximate, as the amount of time spent in the 3D pipeline varies somewhat depending on the graphics primitive performed.

GPGPU		**Pangaea**	
3D pipeline	$\sim$ 1500	Bus interface	11
Thread Dispatch	15	Thread Dispatch	15
Total	$\sim$ 1515	Total	26

Table 5: Thread Spawn Latency in cycles.

Unlike the Pangaea case, the measurement for the GPGPU case is optimistic since (1) the latency numbers apply only when the various caches dedicated to the front-end all hit, and (2) the measurement does not take into account of the overhead incurred by the CPU to prepare command primitives. In the GPGPU case, the CPU needs to do a significant amount of work before the GPU hardware can begin processing. For example, when the GPGPU parallel computation is expressed in a shader language, the CPU needs to first convert the device independent shader byte code into native graphics primitives, place the appropriate commands into the command buffer, and notify the GPU that there is new data in the command buffer. Since CPU and GPU operate in separate address spaces, the CPU would also need to go through the device driver interface to copy the code and data into non-cacheable memory the GPU can access. This process is usually inefficient due to the involvement of privilege level ring transitions and data movement between cacheable and non-cacheable memory regions. In effect, the 1515 cycle latency for GPGPU assumes 0-cycles of CPU work. In contrast, the Pangaea case simply involves a user-level 32-bit store containing the instruction pointer of the EU thread to be spawned to the EU core.

Much of the latency for the GPGPU case comes from needing to map the computation to the 3D graphics processing pipeline. Most of the work performed in the 3D pipeline is not relevant to the computation, but is necessary if the problem were formulated as a 3D computation. By bypassing the front-end of the 3D pipeline, we have successfully reduced the thread spawning latency. With spawning latency reduction of two orders of magnitude, Pangaea can enable more versatile exploration of ILP, DLP and fine grain TLP through tightly-coupled execution on the heterogeneous multi-cores. In Section 6, we will study a set of workloads with varying degrees of ILP, DLP and TLP.

Kernel	Description	EU-kernel code size	Data Size	Threads	Icount/ thread
Linear filter 1,2	computes average of pixel and 8 neighbors	2.5 KB	**1:** 640x480 24-bit image	6,480	159
			2: 2000x2000 24-bit image	83,500	159
Sepia Tone 1,2	modifies RGB values of each pixel	4.0 KB	**1:** 640x480 24-bit image	4,800	247
			2: 2000x2000 24-bit image	62,500	247
Film Grain Technology (FGT)	applies artificial film grain filter from H.264 standard	6.6 KB	1024x768 image	96	15,200
Bicubic Scaling	scales YUV image using bicubic filtering	6.1 KB	360x240 → 720 x 480	2,700	691
k-means	k-means clustering of uniformly distributed data	1.5 KB	k=8, 100,000x8	200,000	94
SVM	kernel from SVM-based face classifier	3.6 KB	704x480 image	1,320	11248

Table 6: Benchmark Suites

6. PERFORMANCE EVALUATION

This section evaluates the performance of Pangaea. Our benchmarks are run on the FPGA prototype with the configuration described in Table 1, under Linux, compiled using a production IA32 C/C++ compiler that supports heterogeneous OpenMP with the CHI runtime [38]. For the benchmarks, we select four product quality media processing kernels and 2 informatics kernels that are representative of highly parallel compute-intensive workloads rich in ILP, DLP and TLP. These benchmarks have been optimized to run on the IA32 CPU alone (with 4-way SIMD) as the baseline, and on Pangaea to use both the IA32 CPU and the GMA X4500 EUs in parallel whenever applicable, including leveraging the new IA32 ISA extension to support user-level interrupts. Table 6 gives a brief description of the benchmarks. While FGT and SVM have relatively few threads of coarser granularity, the rest have many more threads of fine granularity.

Figure 5 shows the speedups of Pangaea relative to a CPU only case. Despite each EU being slightly smaller in area than the CPU, running highly parallel workloads on Pangaea rather than the IA32 CPU alone results in significant performance improvements, ranging from 1.9 to 8.8× improvement on a two-EU Pangaea system.

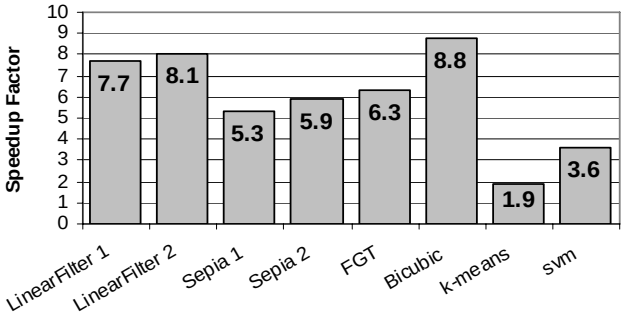

Figure 5: Pangaea speedup vs. CPU w/ SSE alone.

The first four benchmarks are implementations of several key image and video processing algorithms. They operate on image frames and tend to be highly parallelizable, because an input image can usually be divided into independent macro-blocks (*e.g.*, 8 by 8 pixels in dimension) which can be processed independently. Consequently, many parallel threads can be created, each corresponding to a macroblock. Each thread can be further optimized to exploit 8-wide SIMD operations. Between threads, spatial or temporal locality can also be exploited. For example, in some video processing algorithms, adjacent macro-blocks along x-, or y- or the diagonal dimension may have overlapping stripe or mini-blocks. It is advantageous to schedule the corresponding threads back-to-back so that the overlapped data fetched by the first thread can be reused by the second thread. With architectural support for fly-weight thread spawning and inter-core signaling, Pangaea can efficiently support agile user-level thread scheduling. With these optimizations, the benchmarks show impressive speedups. Linear filter computes the average pixel values of a pixel with its 8 neighbors. Sepia tone modifies each pixel's RGB values, dependent only on the same pixel's original RGB values. FGT applies an artificial film grain filter. Bicubic performs a bicubic-filtered image scaling.

Although similar to Sepia tone, Linear filter sees a larger speedup mainly because Linear filter makes references to neighboring pixels, which the CPU cannot store entirely in architectural registers, requiring cache accesses. When executed on the EU, an entire block of pixels can be stored and manipulated in the EU's large register file. The other two benchmarks are classic machine learning informatics benchmarks that focus on either clustering (k-means) or segregating (SVM) classes of high dimensional data. K-means clustering finds k clusters in a set of points by finding the set of points closest to randomly-generated centroids, then iteratively moving the centroid to be the mean of the set of points that belongs to it. This benchmark is partially parallelized, and cooperatively executes on both the CPU and EU simultaneously. Finding which cluster each point belongs to is parallel and runs on the EU, and computing the mean is performed serially, on the CPU. The serial portion is the bottleneck in this benchmark, resulting in a small 1.9× speedup. The transition between parallel and serial sections of the computations is made more efficient through the fly-weight thread spawning and signaling between the CPU and the EU. The Support Vector Machine (SVM) kernel performs the dot product of blocks of pixels with an array of constant values. Unlike k-means, there is no significant serial portion to the code, and a speedup of 3.6× is achieved.

While it may seem that achieving almost a 9× speedup with only twice the number of functional units is unrealistic, multiple architectural features combine to allow the EUs to operate much more efficiently than the CPU's SIMD unit and result in larger than expected speedup. As discussed in Section 3, Pangaea utilizes not only DLP but also TLP. When multiple threads exist, multithreading signifi-

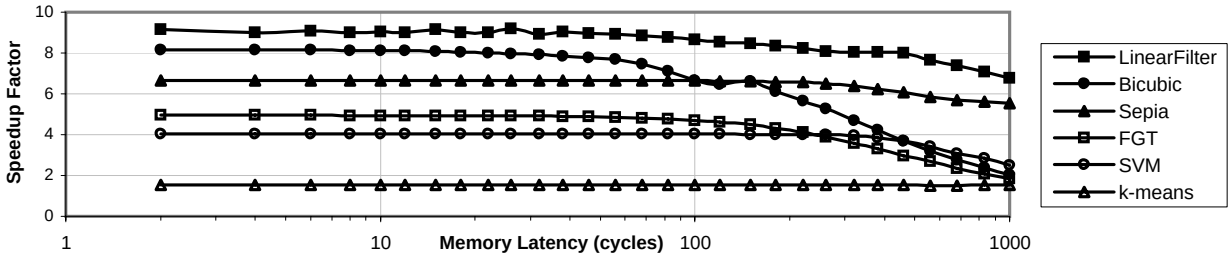

Figure 6: Tolerance of Pangaea to Different Memory Access Latencies.

cantly increases utilization of the EU's functional units (*e.g.* 92% on the EUs vs 65% on the CPU in Linear filter). Additional performance improvement is attributable to the EUs' ISA. The EU's SIMD-8 instructions allow a large reduction in the instruction count for these data parallel workloads. Furthermore, the EU's large register file minimizes spilling of registers to memory (57% of CPU instructions in bicubic reference memory, whereas only 7.4% of the EU instructions are loads and stores). Bicubic also heavily uses the multiply accumulate instruction and the low latency accumulator registers (55% of EU instructions), which the CPU does not support, giving this benchmark a particular advantage on the EUs.

To further explore the performance aspects of Pangaea, we assess its sensitivity to the latency of the shared memory hierarchy. Here we vary the latency it takes the EU hardware thread to access shared memory from 2 to 1000 cycles. Figure 6 shows the results of this experiment. This experiment sheds light on the impact of not only different access times for the shared L2, but also different shared memory configurations. While a latency of between 50 and 100 cycles might simulate a shared last level cache, latencies exceeding 100 cycles can indicate the performance impact of configuration where CPU and EUs share no caches at all. Although the "performance knee" varies for each benchmark, performance is insensitive to access latency up to approximately 60 cycles for all benchmarks. Once access time exceeds 100-200 cycles, performance improvement slowly diminishes, but even at 1000 cycles, speedups are still anywhere from 1.9× to 5.9×. Bicubic and FGT are the most sensitive to access latency due to the fact that the EU's instruction cache is only 4KB, and each of these kernels is over 6KB in size (see Table 6). Consequently, higher memory latency affects not only data accesses, but also the instruction supply. K-means shows the least sensitivity to memory latency. This is because the serial portion of the algorithm (the part run on the CPU) continues to be the performance bottleneck.

The results of this sensitivity study indicate that a variety of shared cache configurations and access times will produce similar speedups. The performance of the Pangaea architecture does not depend entirely on sharing the closest level cache; the choice of which level of memory hierarchy to share can be traded off with margins for ease or efficiency of implementation without noticeably degrading performance.

7. CONCLUSION AND FUTURE WORK

In this paper, we present Pangaea, a heterogeneous multi-core design, including its architecture, an implementation in synthesizable RTL and an in-depth evaluation of power, area, performance efficiency and tradeoffs. We demonstrate the potential to significantly improve power/area/performance efficiency for heterogeneous multi-core designs, should they be targeted for a general-purpose heterogeneous multithreading model beyond legacy graphics. As long as Moore's Law continues at its current pace, the level of integration in mainstream microprocessors will continue to increase in terms of quantity and diversity of heterogeneous building blocks, so will the need to achieve higher power/area efficiency. It is advantageous to represent these heterogeneous building blocks as additional architectural resources to the general-purpose CPU. Such tighter architectural integration will allow ease of programming and the use of these new building blocks without requiring drastic changes in the software ecosystem (*e.g.*, the OS). In turn, the software ecosystem will continue to innovate and harvest the parallelism offered by the hardware more efficiently. Even for graphics, leading researchers [11, 34] are actively investigating opportunities beyond today's brute-force, unidirectional rendering pipeline. They have proposed programmable graphics and interactive rendering techniques to design adaptive, demand-driven renderers that can efficiently and easily leverage all processors in heterogeneous parallel systems and tightly couple the distinct capabilities of the ILP-optimized CPU and DLP/TLP-optimized GPU multi cores to generate far richer and more realistic imagery. Like the famed wheel of reincarnation [30], an efficient heterogeneous multi-core design like Pangaea potentially offers opportunities to significantly accelerate parallel applications like interactive rendering. We continue to actively investigate these opportunities in our on-going exploration.

Acknowledgments

We would like to thank Prasoonkumar Surti, Chris Zou, Lisa Pearce, Xintian Wu, and Ed Grochowski for the productive collaboration throughout the Pangaea project. We also appreciate the support from John Shen, Shekhar Borkar, Joe Schutz, Tom Piazza, Jim Held, Ketan Paranjape, Shiv Kaushik, Bryant Bigbee, Ajay Bhatt, Doug Carmean, Per Hammarlund, and Dion Rodgers. In addition, we would like to thank the anonymous reviewers whose valuable feedback has helped the authors greatly improve the quality of this paper. Henry Wong and Tor Aamodt are partly supported by the Natural Sciences and Engineering Research Council of Canada.

8. REFERENCES

[1] *GPGPU: General Purpose Computation using Graphics Hardware*. http://www.gpgpu.org.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. 17th International Symposium on Computer Architecture*, pages 104 – 114, May 1990.

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proc. 32nd International Symposium on Computer Architecture*, 2005.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. 32nd International Symposium on Computer Architecture*, pages 506–517, Jun. 2005.

[5] A. Bracy, K. Doshi, and Q. Jacobson. Disintermediated Active Communication. *IEEE Computer Architecture Letters*, 5(2), 2006.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM Transactions on Graphics*, volume 23, pages 777–786, 2004.

[7] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a Message-Driven Processor. In *Proc. 14th International Symposium on Computer Architecture*, pages 189 – 196, 1987.

[8] S. Ghiasi. Aide de Camp: Asymmetric Multi-core Design for Dynamic Thermal Management. Technical Report TR-01-43, 2003.

[9] E. Grochowski and M. Annavaram. Energy per Instruction Trends in Intel Microprocessors. *Technology@Intel Magazine*, March 2006.

[10] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of Both Latency and Throughput. In *Proc. IEEE International Conference on Computer Design*, 2004.

[11] E. Haines. An Introductory Tour of Interactive Rendering. *IEEE Computer Graphics and Applications*, 26(1), 2006.

[12] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple Instruction Stream Processor. In *Proc. 33rd International Symposium on Computer Architecture*, 2006.

[13] D. S. Henry and C. F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, 1992.

[14] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proc. 23rd International Symposium on Computer Architecture*, pages 244–255, May 1996.

[15] Intel. *G45 Express Chipset*. http://www.intel.com/Assets/PDF/prodbrief/319946.pdf.

[16] Intel. *IA Programmers Reference Manual 2008*. http://www.intel.com/products/processor/manuals/index.htm.

[17] Intel. *Use MONITOR and MWAIT Streaming SIMD Extensions 3 Instructions*. http://softwarecommunity.intel.com/Wiki.

[18] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[19] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. 36th International Symposium on Microarchitecture*, Dec. 2003.

[20] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. 31st International Symposium on Computer Architecture*, Jun. 2004.

[21] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *Proc. 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[22] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, 1994.

[23] S.-L. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh. An FPGA-based Pentium in a Complete Desktop System. In *International Symposium on Field-Programmable Gate Arrays*, pages 53–59, 2007.

[24] O. Maquelin, G. R. Gao, H. H. J. Hum, K. B. Theobald, and X.-M. Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proc. 23rd International Symposium on Computer Architecture*, pages 179–188, 1996.

[25] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance Evaluation of GPUs Using the RapidMind Development Platform. In *Proc. 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[26] Microsoft. *A Roadmap for DirectX*. http://msdn.microsoft.com/en-us/library/bb756949.aspx.

[27] T. Morad, U. Weiser, and A. Kolodny. ACCMP - Asymmetric Cluster Chip-Multiprocessing. Technical Report 488, CCIT, 2004.

[28] T. Morad, U. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE Computer Architecture Letters*, 5(1), 2006.

[29] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proc. 23rd International Symposium on Computer Architecture*, 1996.

[30] T. H. Myer and I. E. Sutherland. On the Design of Display Processors. *Communications of ACM*, 11(6):410–414, 1968.

[31] Nvidia. *Compute Unified Device Architecture (CUDA)*. http://developer.nvidia.com/object/cuda.html.

[32] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[33] Peakstream Inc. *The PeakStream Platform: High Productivity Software Development for Multi-core Processors*, 2006.

[34] M. Pharr, A. Lefohn, C. Kolb, P. Lalonde, T. Foley, and G. Berry. Programmable graphics: the future of interactive rendering. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–6, 2008.

[35] C. A. Thekkath and H. M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, 1994.

[36] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture. *Intel Technology Journal*, (Q4):14, 1999.

[37] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. 19th International Symposium on Computer Architecture*, pages 430–440, May 1992.

[38] 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 *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, 2007.