Tradeoffs in Designing Accelerator Architectures for Visual Computing

Aqeel Mahesri Daniel Johnson
Neal Crago Sanjay J. Patel
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{*mahesri,djohns53,crago,sjp*}*@uiuc.edu*

Abstract

Visualization, interaction, and simulation (VIS) constitute a class of applications that is growing in importance. This class includes applications such as graphics rendering, video encoding, simulation, and computer vision. These applications are ideally suited for accelerators because of their parallelizability and demand for high throughput. We compile a benchmark suite, VISBench, to serve as a proxy for this application class.

We use VISBench to examine some important high level decisions for an accelerator architecture. We propose a highly parallel base architecture. We examine the need for synchronization and data communication. We also examine GPU-style SIMD execution and find that a MIMD architecture usually performs better.

Given these high level choices, we use VISBench to explore the microarchitectural design space. We analyze area versus performance tradeoffs in designing individual cores and the memory hierarchy. We find that a design made of small, simple cores achieves much higher throughput than a general purpose uniprocessor. Further, we find that a limited amount of support for ILP within each core aids overall performance. We find that fine-grained multithreading improves performance, but only up to a point. We find that word-level (SSE-style) SIMD provides a poor performance to area ratio. Finally, we find that sufficient memory and cache bandwidth is essential to performance.

1. Introduction

In this paper we examine architecture-level tradeoffs for applications broadly associated with visual computing. Visual computing deals with the processing, rendering, and modeling of visual information. It includes graphics rendering, as well as video processing, computer vision, imaging, tracking, and physics simulation. To support our exploration, we develop a benchmarking suite called VISBench * (Visual, Interactive, Simulation Benchmarks) to serve as an experimental proxy for the visual computing domain.

We propose a highly parallel, throughput-oriented meta-architecture for a visual computing accelerator. An accelerator is a co-processor that allows the CPU to off-load and accelerate compute intensive work. Our proposed accelerator is built around a large array of simple compute cores, connected by a minimal on-chip network to a shared cache and a high bandwidth memory system. In addition, we examine high-level issues such as synchronization and communication, and evaluate the performance potential of GPU-style SIMD execution of scalar threads. We determine that VISBench, unlike raster graphics, does need support for fine-grained synchronization, but that the applications are insensitive to its performance. We determine that for some applications, SIMD execution provides a major performance benefit, but other applications' control flow divergence results in an even greater performance loss.

We examine tradeoffs in designing the architectural implementation of our meta-architecture. Using VISBench, we explore the performance and area tradeoffs for a number of architecture-level design considerations, such as core architecture (1-wide vs 2-wide, in-order vs out-of-order), effects of word-level SIMD, multi-threading, and cache hierarchy. Our major conclusions are as follows: There exists enough low-hanging instruction-level parallelism to profitably support dual-issue compute cores. Given the significant area overhead of word-level SIMD execution, supporting these instructions is not worthwhile considering their low frequency in most VISbench apps. 2-way multi-threading is often worth the extra area overhead; 4-way is not.

Clearly, like most experimental studies, our results must be viewed in light of our assumptions; due to the wide variation in the overall design space, one can potentially arrive at different conclusions with different assumptions. Nonetheless, the results in this paper serve as a starting point for investigation of accelerator tradeoffs and relative trends are likely to hold even with different assumptions.

2 Visual Computing

We broadly define Visual Computing as application domains associated with the processing, rendering, and modeling of visual information. Applications in this domain represent a class of performance drivers for consumer and high-end computing. Higher baseline performance on such appli-

*Our VISBench should not be confused with NCSA VisBench, an unrelated application for analyzing remote CFD simulations

cations results in greater functionality and value to the end user. In many cases, the desire is to achieve interactive rates on certain applications (for example, video games require rendering and simulation rates of 30 frames per second and above).

Due to their visual nature, these applications tend to have considerable data-level parallelism. One can view visual computing applications as those that naturally map onto the GPU roadmap. As GPUs become more programmable, questions arise of what other applications can be mapped onto GPU architectures and how GPUs should be rearchitected to address the needs of the broader application base. We refer to this general acceleration architecture as an xPU (as do others [20]).

2.1 VISbench

To address the question of xPU architecture, we create an experimental benchmark suite consisting of a sampling of visual computing applications. We selected open-source applications that have some relevant deployment in commercial products.

In VISBench (Visual, Interactive, Simulation Benchmark Suite) we cover classic visualization application areas, such as high-quality graphics rendering (Blender) and lighting (POVRay), and also video encoding (H.264), applications that are in commercial use today but which also have a continual need for improved throughput. We also cover emergent applications, such as interactive dynamics simulation (Open Dynamics Engine), computer vision (OpenCV) and high quality medical imaging (MRI), applications that are made possible by the widespread availability of low-cost, high-performance xPU architectures. VISBench applications are described below, and in further detail in [8].

For some VISBench applications, we start with sequential versions that implement parallelizable algorithms. For other applications, we start with versions parallelized with OpenMP or Pthreads. We remove any existing parallelization from the applications. We then parallelize the applications by placing annotations around the parallel loops. These annotations, used in the same manner as OpenMP pragmas, consist of dummy function calls to mark the beginning and end of parallel sections, and of individual threads. The annotations are detected by our simulation infrastructure and used to model the performance of the parallel implementation.

The space of visual computing is large and diverse. Naturally, our selection of open source benchmarks is only a sampling, and serves as a proxy for the commercial applications in this space. Moreover, we are considering applications that were not originally written with massively parallel accelerators in mind, and as such do not cover the whole set of architecture-specific optimizations proposed in the GPGPU community. Nonetheless, VISBench allows us to analyze a broad and relevant range of visual applications in order to provide insight into design choices for future accelerator architectures.

2.1.1 Scanline rendering: Blender renderer

Blender [12] is a free-software animation and 3D modeling program. We include Blender's internal 3D renderer as a VISBench application.

Blender implements solid scanline rendering, similar to rasterization algorithms used for real-time 3D graphics, though it is far more flexible and aimed at higher quality off-line rendering. Blender's rendering algorithm is broadly similar to algorithms such as REYES [15], used by RenderMan, which is commonly used for cinematic-quality rendering of movies, though the latter is more complex and offers far greater programmability.

For our benchmark input, we use a complex image of a hairball with a room as the background, rendered into a 640 $\times$ 480 image. Using the parallelized version of Blender, this image can be rendered by up to 76,800 threads, ranging in length from 500K to 2.4M instructions. The complete render takes roughly 60B instructions.

2.1.2 Ray tracing: POVRay

POVRay is a widely used ray tracer, and is also an FP benchmark in SPEC CPU2006. As input we use a scene containing a chessboard with a number of glass pieces. This input, which is included as part of the POVRay release, contains a large number of complex reflections and refractions. The scene is rendered at 384 $\times$ 384.

In VISBench, POVRay is decomposed hierarchically, with one thread for rendering each row, which in turn spawns off one thread for rendering each pixel. A full render produces 147,456 threads, ranging from 98K to 4M instructions.

2.1.3 Video encoding: H.264 motion estimation kernel

H.264, also known as MPEG-4 AVC, is a commonly used encoding for high-definition video. The most compute-intensive portion of the H.264 encoding process is motion estimation. VISBench includes a motion estimation kernel based on the sum of absolute differences (SAD) method.

The SAD kernel is embarrassingly parallel and can be parallelized in a number of ways. We use a standard CPU version of the kernel, written in C. We handle each macroblock in parallel, and within each macroblock all of the SAD comparisons for a particular row. One HD image has 3600 macroblocks, and a vertical search range of 33, so we have 118,800 threads all roughly 75K instructions long, for a total of 8.9B instructions.

Note that commercial versions of the H.264 encoder perform motion searches that are optimized and predictive, and perform less work than the full-search technique we model, with a slight loss of compression [37].

2.1.4 Dynamics simulation: ODE PhysicsBench

Open Dynamics Engine (ODE) is a free library for simulating articulated rigid body dynamics. PhysicsBench [38] is a suite of physical simulations built using the ODE library. A parallel implementation of PhysicsBench and ODE is described in [38].

The computation of ODE exhibits distinct phases: the minimally parallel *broad-phase*, the massively parallel *narrow-phase*, a sequential island creation phase, and massively parallel island processing and cloth processing phases.

Our test simulation from PhysicsBench contains 1608 objects, 933 of them as boxes in breakable walls. Collision detection generates 61 threads ranging from 390K to 2.3M instructions, the constraint solver generates 192 threads ranging from 127K to 282K instructions, and cloth simulation generates 7000 threads ranging from 12K to 525K instructions. One timestep takes roughly 900M instructions.

2.1.5 High quality MRI

In magnetic resonance imaging (MRI), data from a magnetic resonance scan is reconstructed from a set of data points in the spatial frequency domain into a 3D image of the scan target. Conventionally, this is done using an FFT, which requires the data points to be on a Cartesian grid. However, due to the physics involved, a better image can be obtained using a non-Cartesian grid and performing a non-uniform Fourier transform, a process that can take many hours. A CUDA implementation of non-Cartesian MRI reconstruction is described in [32]. We base our study on the serial version.

The main computation in an image reconstruction from the non-Cartesian data consists of computing two vectors, Q, given by $Q(x_n) = \sum_{m=1}^{M} |\phi(k_m)|^2 e^{(j2\pi k_m x_n)}$, and $\mathrm{F}^H\mathrm{d}$, given by $\left[F^H d\right]_n = \sum_{m=1}^{M} \phi^*(k_m) d(k_m) e^{(j2\pi k_m x_n)}$. We use the $\mathrm{F}^H\mathrm{d}$ computation as a benchmark. We compute all $\mathrm{F}^H\mathrm{d}$ values in parallel, with each independent thread computing the value for different elements. A full MRI image requires a square convolution with several hundred thousand data points. In our study we simulate a 4096 by 4096 convolution which produces 4096 threads, each 240K instructions long.

2.1.6 Computer vision: OpenCV based face detection

OpenCV is an open-source library containing many basic algorithms used in computer vision. FacePerf [13] is a set of benchmarks for evaluating face recognition performance. As part of VISBench we use a modified version of the the OpenCV Haar-based face detector that is included in FacePerf.

The face detector can easily be parallelized, with subimages being processed in parallel. Pruning the search space results in some irregularity between the parallel tasks, but still allows for a large number of tasks. For this paper we run the face detector on a 2 megapixel image with over a dozen faces at varying angles and distances.

The face detector runs in repeated phases, each of which in turn has 3 parallel sub-phases: a short phase to set up the classifier, with 22 threads ranging from 1.3K to 95K instructions, and two phases running the classifier, each with roughly 450 threads ranging from 43K to 975K instructions.

2.2 Performance scaling

In visual computing applications, there is a correlation between the complexity of the application data set and the user experience. For instance, the quality of a rendered image can be increased by rendering more polygons, and the fidelity of a physical simulation can be improved by modeling more objects. In this sense, VISBench applications are appealing targets for vendors of high performance hardware, as these applications can take advantage of rapidly scaling computer performance.

Each of the VISBench applications contains short section(s) of serial code in addition to the time-intensive parallel portions. In all VISBench applications, we find that at least 86% of execution time on a serial machine is spent in parallel portions.

Furthermore, and perhaps more important, all of these parallel sections are either O(n^2) or quasi-O(n^2)* in the number of primitives being processed (such as pixels, polygons, objects, etc). That is, as the complexity of the visual simulation grows, as is the desire from generation to generation, the parallel workload grows rapidly. Hence, these workloads benefit from Gustafson's Law, which states that any sufficiently large problem can be efficiently parallelized. This important to note as it implies that an accelerator architecture can be scaled with Moore's Law through parallelism and still provide value to the same application.

3. Accelerator Meta-architecture

In this section, we examine the high-level architecture for a visual computing accelerator. We introduce a massively parallel co-processor that can speed up the compute-intensive portions of VISBench.

Given this basic architecture, we examine the data sharing and synchronization properties of VISBench applications and use them to motivate communication mechanisms that are different from a traditional multicore and yet also unlike GPUs. We move cache coherence functions from hardware to software as a means to improve area efficiency without incurring unreasonable overhead.

In addition, we examine control flow divergence and explore its implications for the accelerator's execution model. We compare multicore-style MIMD execution against GPU-style SIMD execution of scalar threads, and find the former to perform well over the broadest set of applications.

*O(n) with significant constants or worst case O(n^2)

3.1 Basic Architecture

Figure 1 shows the basic architecture of an xPU. The xPU consists of a compute array of a large number of cores, arranged in clusters, connected to a large shared cache. It also contains a controller core, a high bandwidth memory system, and a hardware assisted thread management system.

The xPU is connected to the rest of the system via a standard interface such as PCIe or HyperTransport, using DMAs to transfer data between the host and accelerator. The xPU contains a controller core to handle data transfers and initiate execution on the accelerator.

The compute fabric of the xPU consists of an array of mini-cores. Each core consists of an execution pipeline with integer and floating point execution units, register files, and small instruction and data caches. These cores are arranged into clusters which share a common link to the global interconnect, to allow cores to share bandwidth.

The on-chip interconnect connects the compute array to the global cache (gcache). The gcache is a large, multi-banked cache accessible to all the cores in the array. This arrangement allows for a very large gcache bandwidth, which in effect multiplies the memory bandwidth.

The gcache is connected to a high-bandwidth interface to off-chip memory. Several banks of gcache are connected to a single memory controller, which controls a single high-bandwidth DRAM channel, striping the memory space across the DRAM channels.

3.2 Synchronization and communication

Traditional multicore architectures provide extensive support for fine-grained synchronization via atomic primitives upon which to implement such higher level constructs as locks and semaphores. Multicore architectures also provide extensive support for data communication through a cache-coherent shared memory. Supercomputers, on the other hand, provide message passing interfaces. These interfaces allow for simple and highly scalable synchronization mechanisms, but complicates the programmer's task with regard to sharing data. Finally, GPUs provide a minimum of support for communication, requiring data to go off chip to memory and then be read back in. In the xPU, we envision providing a compromise between these approaches that reflects the synchronization and data communication patterns in VISBench applications.

To evaluate these choices, we take a look at the data communication patterns within the VISBench applications. These can be identified by examining the interaction between loads and stores from different threads. Figure 2 shows the different possible patterns of data communication among parallel threads. Figure 3 shows the frequency of non-private loads and stores.

From Figure 3, we see frequent loads to read-shared data. In a message passing architecture we deal with read-shared data either by keeping it on a "home node" and sending it to a requester, or by replicating it on multiple nodes. However, the former strategy is a poor choice because of the frequency of access to such data. Moreover, the limited amount of cache and bandwidth available per core on a single chip limits the appeal of the latter strategy. On the other hand, shared memory systems can deal easily with read-shared data, and can even take advantage of the sharing to improve utilization of a shared cache.

When the only data communication between threads occurs across barriers, a bulk synchronous model is sufficient. However, from Figure 3, we also see the presence of data that needs to be communicated within barriers, particularly in ODE. Nonetheless, while fine-grained synchronization needs to be supported, it is not performance-critical. Even in ODE, shared writes within barriers occurs infrequently, about once every 600 instructions.

Cache coherence allows parallel processing elements to communicate data through memory without incurring a performance overhead in the absence of a collision. This is particularly useful if the application has many writes to and reads from write-shared data, but where writes to the same location rarely collide. From Figure 3 we see that shared writes are infrequent, and while shared loads are common, the vast majority are separated from their corresponding writer by a synchronization barrier. This suggests that fast, automatic cache coherence in hardware may be unnecessary. Instead, we can keep values coherent by flushing output data from caches at barriers.

Putting the above observations together, we envision a memory model consisting of a shared address space but without hardware cache coherence. Instead, we assume a software-enforced coherence mechanism. The basic mechanism for keeping shared data consistent is to exclude such values from local caches. Shared reads and writes are performed with special global load and store operations, which always go out to the global cache or memory. In addition, input data must be read in globally at the beginning of each thread while output data must be flushed out from local caches at the conclusion of each thread (though input and output values can be locally cached during the running of the thread). This model requires the programmer to identify the input, output, and shared data and use global operations and flushes when appropriate, but on VISBench applications it allows us to achieve high performance with reduced hardware complexity.

3.3 MIMD execution

SIMD execution is a common route to achieving high performance in architectures ranging from traditional vector-processing supercomputers to modern GPUs. This sort of global SIMD, different from the 128-bit SIMD found in most modern CPU architectures, can be very effective at obtaining high performance on numerical applications. On the other hand, it may provide poor performance on control-

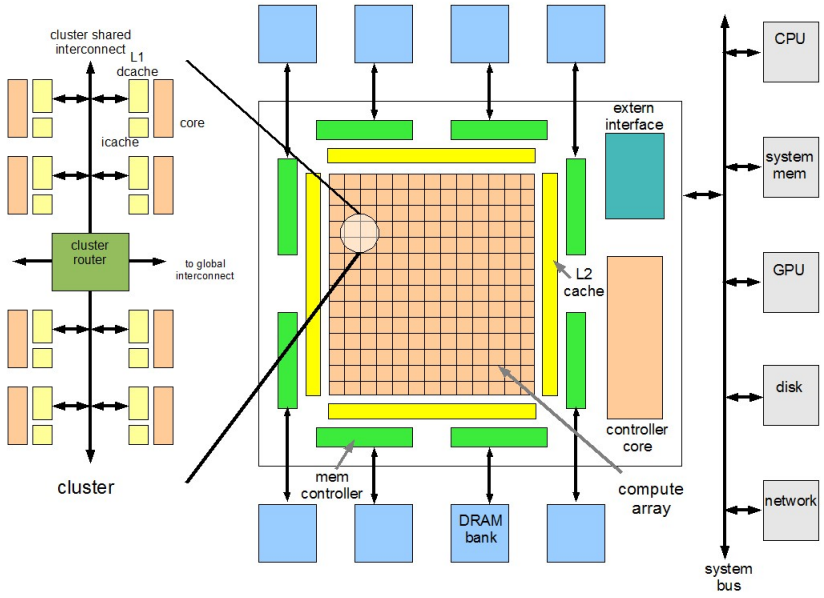

Figure 1. Block diagram of accelerator

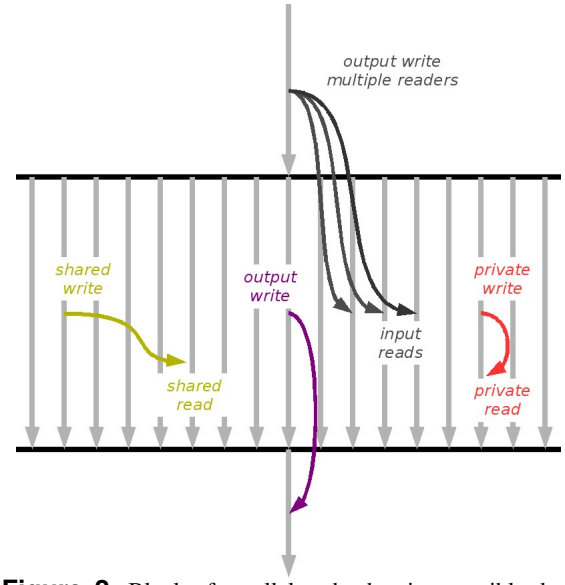

Figure 2. Block of parallel code showing possible data sharing.

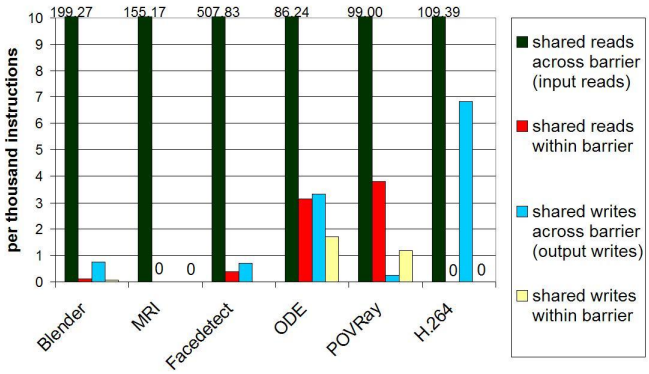

Figure 3. Frequency of shared memory accesses

intensive code.

The compute fabric of the xPU can either be designed using discrete cores which execute in a MIMD fashion, or it can be designed with clustered scalar pipelines that execute in SIMD lockstep in a manner similar to GPUs. Dense numerical applications are able to take advantage of SIMD because of the very regular pattern of control flow. In the case of visual applications, we could replace the thread execution model (which is notably MIMD) with large scale SIMD if the control flow were largely the same across different threads. On the other hand, if control flow varies from thread to thread, SIMD hardware will suffer a performance loss as the parallel threads would be forced to serialize.

To determine relative performance of SIMD and MIMD configurations, we compare the instruction throughput per pipeline of a SIMD machine with that of a MIMD machine, assuming an idealized memory system. Scalar threads from each benchmark are grouped together into warps. As long as all the threads in a warp are following the same control flow path, the SIMD machine executes the warp in lockstep. When threads diverge, the threads executing down one path are serialized relative to the threads executing down the other. When the threads in each of these subwarps reconverge, the warp again executes in lockstep. To minimize the divergence within warps, we group together threads which operate on consecutive data elements. We use the immediate postdominator of the divergence point as the reconvergence point, which has been found to be near optimal on real pro-

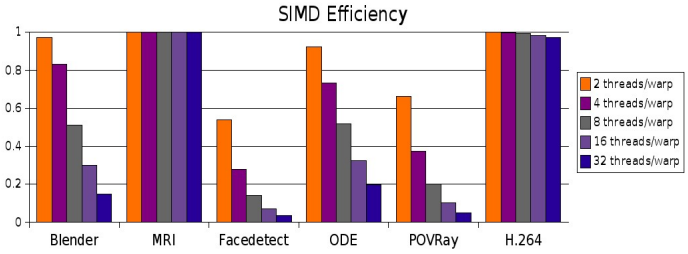

Figure 4. SIMD efficiency versus warp size

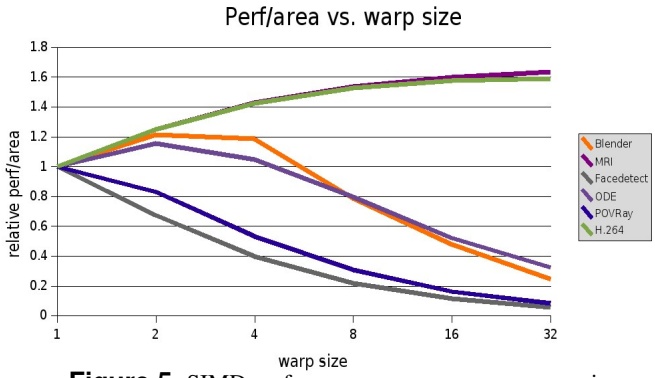

Figure 5. SIMD performance per area versus warp size

grams [18]. We use a control flow stack to handle multiple levels of control flow divergence, so that we can handle divergence and reconvergence even within subwarps.

Figure 4 plots the relative IPC (per pipeline, relative to MIMD) for varying warp sizes. We can see that for some applications (H.264 and MRI), the SIMD efficiency is very high, even for large warp sizes. Other applications, however, have a declining SIMD efficiency as the warp size grows.

While SIMD has a cost in IPC, MIMD has a cost in chip area. MIMD requires all the control flow logic of the core to be replicated for each pipeline, whereas SIMD only requires one set of control logic per cluster. However, SIMD still requires the register files, functional units, caches and cache ports, and data bits of the pipeline latches to be replicated. We find that roughly 40% of the pipeline area does not need to be replicated. Hence, a 2-way SIMD cluster has 1.6 times the area of a scalar pipeline, while a 4-way has 2.8 times the area.

Figure 5 shows the ratio between area and IPC per cluster for varying warp sizes. The benchmarks in VISBench separate clearly into three groups. The first group is the one that has nearly perfect SIMD efficiency (i.e. H.264 and MRI). The second group (Blender and ODE), shows a modest performance benefit for small warp sizes, but then exponentially decreasing performance as the warp size grows large. The third group (POVRay and Facedetect) shows performance loss with any level of SIMD, and exponential performance loss with large warp sizes. Note that we are assuming perfect memory, so this result is an upper bound on the SIMD efficiency.

One thing we do not consider in this study is algorithmic changes to improve SIMD efficiency. Blender and H.264 required a moderate amount of hand tuning in order to achieve their level of SIMD performance, but were not altered at the algorithm level. With additional programmer effort, one may be able to reclaim much more of the performance loss from SIMD, but this optimization comes at a substantial cost in development time. For instance, work has shown that the SIMD efficiency of ray tracing can be improved, though even with major algorithmic changes it remains well below 100% on complex scenes [33].

This result indicates that SIMD, while a substantial constant factor win for some applications, is a much larger performance loss for others. It also illustrates one of the limitations of GPUs as they expand into more general purpose application domains. For numerical applications and the traditional applications for GPUs, SIMD is the right design choice. However, the performance potential of SIMD architectures is limited as the space of applications expands.

For the remainder this paper, we assume a MIMD architecture.

4 Microarchitectural Evaluation of Architectures for Visual Computing

In the previous section, we examined some of the high-level aspects of an accelerator architecture. In this section, we examine the architecture with a detailed analysis to determine the performance effects of specific microarchitectural decisions. Essentially, we ask the following question: if we were designing a chip with a specific area budget, how should we architect that area to maximize performance. We assume a large area budget (400mm^2 in a 65nm process technology), and examine how to architect the compute cores and caches.

To perform this evaluation, we have developed a performance measurement methodology based on simulation. In addition we have developed a model to compute the area cost of a variety of microarchitectural features by mapping out the required hardware in detail and then determining the area cost of each component.

4.1 Area Modeling Methodology

Modeling area is a challenge for architecture researchers. The most accurate way to model the area of a core is to design the core in RTL and synthesize, place, route, and optimize for speed, power, and area constraints. Unfortunately, this method requires developing RTL for all of the various design possiblities, a time consuming endeavor, and is not ideal for a high level design space exploration.

A much simpler method used by previous researchers is to measure components of real designs using die photos or published values [22]. [30] extends this method by using analytical formulas. However, this method is restricted to

the design assumptions made by the vendor of the baseline design.

We use a hybrid approach that allows us to evaluate a larger design space. We divide the components of the area cost into three groups: SRAMs, functional units, and pipeline logic. For most SRAMs, we compute area estimates using CACTI-6 [31]; however, CACTI is designed to evaluate large SRAM arrays, and so for some of the smallest SRAMs we count the number of bit cells and gates needed to implement the SRAM and multiply by published figures for the sizes of these structures [5].

For ALU and FPU components, we use published synthesis results ([34], [25]). When synthesis results are not available, we obtain area estimates by designing the structures at the logic gate level and counting the number of gates, flops, and SRAM bits required in the implementation.

Finally, for remaining pipeline logic (e.g. muxes, PC selection logic) we implement the components at the logic gate level and again count the number of gates, flops, and SRAM bits.

We compose these components to model three different basic pipeline configurations: A 1-wide in-order pipeline, a 2-wide in-order pipeline, and a 2-wide out-of-order pipeline. These pipeline configurations represent the sort of architectures that are often considered for many-core design because of their small area and high ratio of performance to area.

The 1-wide configuration consists of a standard 5-stage fully-bypassed, in-order pipeline with a static BTFN predictor. The 2-wide in-order pipeline consists of 6 stages with a bimodal predictor. A second simple ALU is added to the execution stage (adder and logical operations, no shifter or multiplier). The out-of-order pipeline consists of a 6-stage, 2-wide, out-of-order pipeline with a bimodal predictor. The pipeline is modeled roughly after the P6 microarchitecture. Execution resources remain the same as the 2-wide in-order. The ROB contains 24 entries, the scheduler 12 entries. We assume all of these cores run at 1GHz.

For each basic configuration, we examine the effects of additional performance enhancing features such as word-level SIMD and multithreading. We model the addition of 2-way and 4-way SIMD, 2-way and 4-way multithreading, and special purpose functional units for sine and cosine. The additional cost of word-level SIMD (a la SSE) is modeled by increasing the number of execution units available as well as accounting for the increase in pipeline registers. No additional register files are added for SIMD, but read and write ports are scaled appropriately. The incremental cost for multithreading is modeled by replicating portions of the front-end stages (fetch, decode, rename), the architectural register files, and the RAT.

Table 1 shows area by major hardware function for each of the basic configurations. Frontend includes fetch and decode as well as scheduling and renaming. Other includes remaining logic and pipeline registers.

We make the simplifying assumption in this study that adding cores does not affect the incremental area consumed by the interconnect. Because we want to focus on the core and cache architecture in this study, we want to minimize the impact of the interconnect. Hence, we model the bandwidth limitations at the global cache level and assume that we can build a network, perhaps somewhat overprovisioned, that is capable of maximizing the utilization of the global cache. Because we hold the global cache bandwidth fixed, we would not need to significanly increase the area consumed by the interconnect in order to supply maximal gcache utilization to a larger number of cores.

Area, of course, is affected by more than just architecture. Physical design itself involves a large design space exploration, and a given microarchitecture can be implemented by widely varying layouts. Assuming that we use synthesis to generate the layout, sources of variation include wiring overhead due to place and route (since optimal place and route is NP-hard), choice of libraries, choice of logic gates, and padding due to DFM. We used a 33% overhead to encompass all of these, but in real designs the overhead can vary dramatically. In later sections we will assume a confidence interval for our area numbers of $\pm 20\%$. This number was the empirical variation in area for layouts generated by the Synopsys Design Compiler in [26].

Other potential sources of inaccuracy in our model include mismatches between the architectural and logic level design, uncertainty in the results from tools such as CACTI, and uncertainty in area numbers for our components. We cross-checked our area results against existing designs such as the MIPS 74K [2] (1.7mm^2 in 65nm) and Tensilica 108Mini (.143mm^2) and 570T (.349mm^2) [3]. These commercial cores contain additional logic for features that our cores lack. However, they show that the core areas shown in Table 1 are achievable.

4.2 Performance Modeling Methodology

We use hand parallelized versions of the VISBench applications, as described in Section 2. We replace Pthread and OpenMP calls surrounding parallel loops with annotations for marking the beginning and end of parallel sections as well as the boundaries of individual threads.

We run our annotated binaries sequentially through a functional simulation frontend that simulates x86 code. This frontend fast-forwards the sequential code to reach the parallel portions that would run on our accelerator. The frontend detects the dummy function calls that serve as thread boundaries and generates an instruction trace for each thread. A cycle-based timing model simulates the performance of all cores, running them in parallel to capture the interleaving of memory accesses across threads.

We assume a hardware mechanism to distribute threads across cores. We assume a single task queue for all the cores, and do not optimize the thread distribution to take advantage

Table 1. Area breakdown by pipeline component in mm^2

	Frontend	Execution	Caches (4KI/8KD)	Other	Core(total)	+SIMDx4	+MTx2	+MTx4
1W in	0.016	0.068	0.140	0.009	0.330	0.200	0.032	0.096
2W in	0.066	0.092	0.150	0.026	0.420	0.250	0.074	0.220
2W out	0.200	0.092	0.150	0.057	0.590	0.250	0.120	0.360

Table 2. Baseline XPU Architecture

	Size	Latency	Organization
Int ALU	32-bit	1 cycle	1 or 2 ALUs
FPU	single-precision	5 cycle	1 FPU
L1 ICache	4KB	1 cycle	2-way
L1 DCache	8KB	1-2 cycles	4-way
GCache	8MB	20+ cycles	32 banks 8-way
DRAM	128GB/s	50ns	8x64bits

of any locality.

We've compiled our benchmarks using gcc 4.1.2, with the -O3, -ftree-vectorize, -ffast-math, -mfpmath=sse, and -march=pentium4 optimization flags. POVRay uses the additional flags -malign-double and -minline-all-stringops. We run each benchmark either until completion or until 2 billion instructions. We fast-forward through initial sequential code ranging from 20M instructions for the SAD kernel to 123M instructions for POVRay. As discussed in Section 3, we are assuming an accelerator model where the host CPU is responsible for executing startup code.

4.3 Experimental Results

In this subsection we experimentally approach the question of how to design an xPU accelerator architecture that maximizes throughput. We evaluate the different base pipeline organizations described above. We also evaluate the performance versus area tradeoff of SIMD instructions and fine-graind multithreading. Finally, we evaluate the performance effects of varying aspects of the memory hierarchy.

The baseline parameters of our chip architecture are listed in Table 2. We divide our area budget into $100mm^2$ for chip overheads, interconnect, and the contoller; $200mm^2$ for the compute array, which includes all the cores with their respective L1 caches; and $100mm^2$ for the global cache.

We used CACTI to find the largest global cache size that would fit in $100mm^2$ with 32 banks such that each bank has an access delay under 1ns (8MB). With a 1GHz clock rate this configuration provides a global cache bandwidth of 1024GB/s.

4.4 Core Pipeline Architecture

Initially we ask the question of how one should design the core pipeline architecture of each processing unit within the xPU given the tradeoff in area/performance for each style of core.

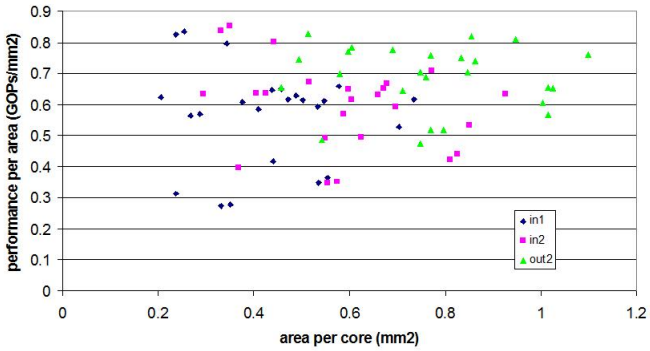

Figure 6. Performance for different pipeline configurations

Figure 6 plots the performance per area (total throughput divided by the $200mm^2$ of area used by the compute array) versus the area per core. Each data point represents the harmonic mean of the throughputs of the 6 benchmarks for a particular core architecture, cache size, and core count corresponding to that configuration. The smallest single-issue in-order configuration, with a core area of $0.205mm^2$ including L1 caches, allows us to fit 975 cores in the array. The largest 2-issue out-of-order configuration, with a core area of $1.10mm^2$, allows us to fit 182 cores in the array.

First, we note that the smallest configuration in1 is not the highest performing despite its high core count. With a small cache and a large number of cores, this configuration is global cache bandwidth bound on most of the benchmarks. On the one benchmark where it is not bandwidth bound (H.264 ME), it is the highest performer.

Second, we note that the highest performing configurations are in2. The in2 configurations provide the highest theoretical throughput, and even with code that is not optimally scheduled, we are able to take enough advantage of ILP in order to overcome its area penalty versus in1.

Third, we note that the best performing in1, in2, and out2 configurations are fairly close in performance, within the uncertainty margin of our area model. Moving from in1 to in2 reduces the pipeline utilization (achieved througput vs. theoretical throughput), but benefits from increased execution resources. Moving from in2 to out2 restores the utilization by scheduling around instructions such as FP operations with moderate latency, but the increased area overhead from the scheduling logic matches the performance gain.

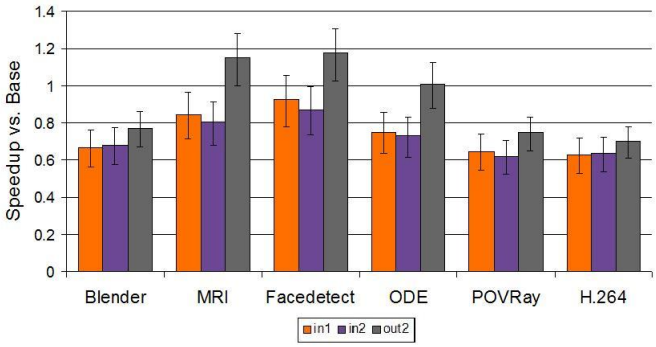

Figure 7. Relative performance of chip with SIMD instructions

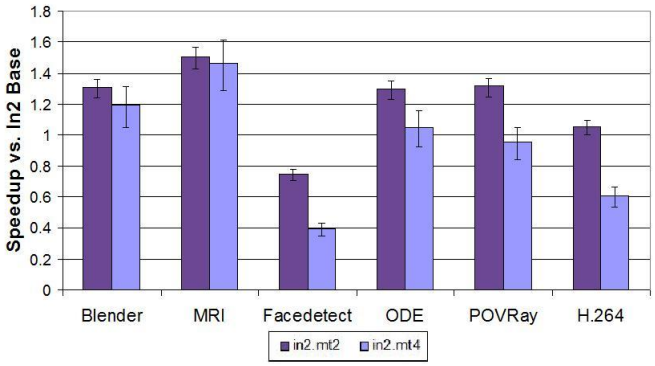

Figure 8. Performance of 2-wide in-order configuration with multithreading

4.5 Word-level SIMD

In Section 3, we examined the performance potential of scalar threads executing in SIMD lockstep. Here we examine word-level SIMD (SIMD operations on small vectors, e.g. SSE), featured in most high performance architectures.

Figure 7 shows the speedup from adding 128-bit SIMD instructions to our baseline architecture. For in1, the area overhead is 59% (39-89% with our area model confidence interval), for in2 it is 58% (39-87%), and for out2 it is 43% (28-64%). The minimum overhead is computed by comparing the upper bound of the baseline area with the lower bound of the extra area consumed by SIMD. Likewise the maximum overhead is computed by lower bound of the baseline with the upper bound of the SIMD area. The error bars on the figure account for the variable area estimates.

From the figure we see that SIMD generally results in a loss of performance on our versions of VISBench, especially with in-order pipelines. This is because, for small core sizes, the area of the FP unit is a large fraction of the total and the penalty for replicating it 4 times is very large relative to the utilization rate.

Furthermore, the benefit is limited to those applications for which the gcc compiler is able to generate substantial amounts of vector code. The MRI kernel is easily vectorized. The OpenCV library was heavily optimized for SSE instructions, with 14% of all operations being SIMD. These two applications were able to achieve modest speedups with the out-of-order configuration, but losses with in-order. Portions of ODE, POVRay, and Blender were also vectorized. However, in the other applications the amount of vectorized code is too small to overcome the area cost of the additional FP units.

4.6 Multithreading

Multithreading is an important technique to get around stalls due to long latency memory operations, and for applications that are throughput-oriented, hardware multithreading seems like a natural fit.

In Figure 8, we show the relative performance of configurations with fine-grained multithreading added. The results show that, for most benchmarks, 2-way multithreading is able to provide a performance benefit. The exceptions are the H.264 motion estimation kernel, which achieves high utilization even without MT, and Facedetect, which suffers from load imbalance as the number of contexts exceeds the number of available threads.

Four-way multithreading, on the other hand, does not generally have a performance benefit. Our VISBench applications spend far less than half of their time stalled waiting for loads, and 2 threads is sufficient to cover most of the stall cycles.

4.7 Cache sizing

Cache sizing is an important parameter in architecture design. Previous CMP optimization studies have shown that optimal cache size is a function of the application being run. We examine performance per area for the core configurations with varying cache sizes. Figure 9 plots all the data points, this time highlighting the different cache sizes.

From the plot, we can see that most of the highest performing configurations have the 4K/8K L1 cache sizes, while many of the 8K/16K configurations also perform well. On the other hand, the configurations with the smallest and largest cache sizes perform worse.

4.8 Memory Bandwidth

Previous work on parallel applications has found memory bandwidth to be a first-order performance constraint, particularly on unoptimized code. In the preceeding studies, we simulated a chip with 128GB/s of bandwidth to the off-chip memory system, modeled as eight independent 16GB/s channels with blocks striped across the channels. This memory bandwidth is large, but is within the achievable limit for 65nm chips and is fairly close to the bandwidth available in the latest high-end GPUs. Figure 10 plots the performance of the 2-wide in-order configuration with varying memory bandwidths. From the graph, it is apparent that performance

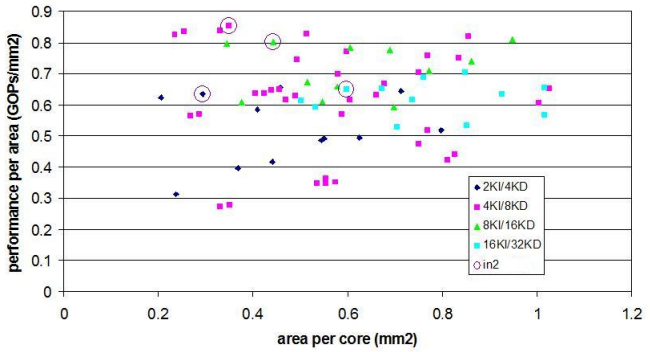

Figure 9. Scatter plot highlighting different cache sizes. Circled data points represent the same in2 configuration with varying cache sizes.

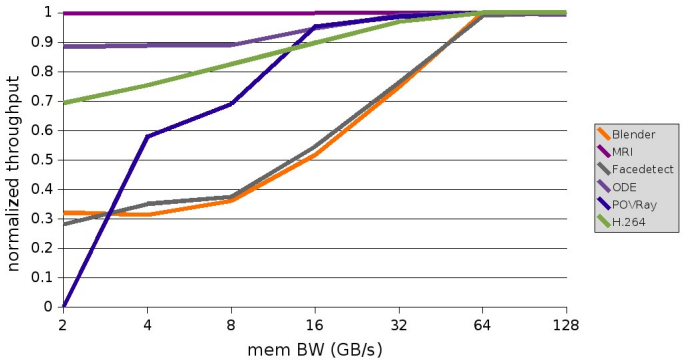

Figure 10. In2 performance versus memory bandwidth (normalized to maximum)

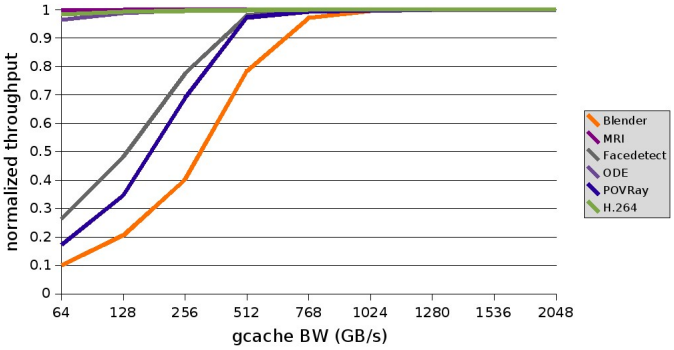

Figure 11. In2 performance versus gcache bandwidth (normalized to maximum)

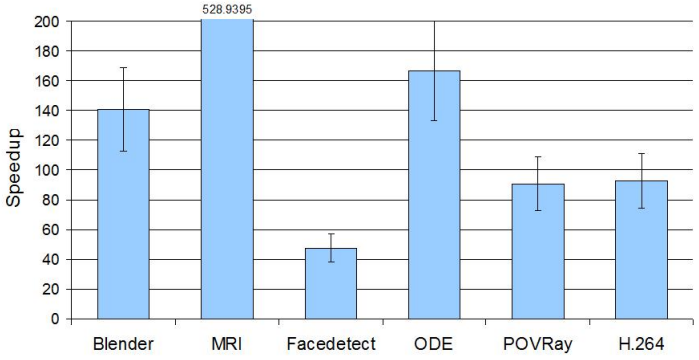

Figure 12. xPU performance versus 2.2GHz Opteron

on the more bandwidth-intensive apps saturates as the bandwidth reaches 64GB/s, indicating that 128GB/s is sufficient bandwidth.

Part of the reason this amount of memory bandwidth is sufficient is the global cache's ability to service a large fraction (80-90%) of the memory requests coming from the cores. In the preceeding studies we simulated a global cache capable of servicing 32 memory accesses per cycle, with each access being 32B, for a total bandwidth of 1024GB/s. Figure 11 plots the performance of the same 2-wide configuration with a varying global cache bandwidth. On the more bandwidth-intensive applications, the chart shows performace falling clearly into two regimes: one where performance varies linearly with bandwidth, and one where the performance is compute-bound. The performance saturates as bandwidth exceeds 768GB/s, indicating that 1024GB/s is a sufficient level.

Note that this result required a few simple optimizations to the application code. In particular, we were able to reduce the bandwidth requirement for MRI from nearly 2TB/s down to only 60GB/s by blocking the convolution. We were also able to reduce contention for gcache banks by offsetting each core's stack such that different cores' stacks began in different gcache banks.

4.9 Summary: an xPU prototype

Pulling the optimal result from Figure 6, we obtain a configuration with 2-wide in-order issue and a 4K icache/8K dcache, a configuration with 573 cores and an average throughput of 165GOPs. Figure 12 shows the speedup of this configuration over a single-core 2.2GHz Opteron. On the parallel sections of VISBench (86% to over 99% of the execution time), the xPU attains an average speedup of 103X. Even when sequential code sections are factored in (up to 14% of execution time in Facedetect and ODE), the total speedup obtained remains over 6X.

5 Limitations

Our evaluation of word-level SIMD is constrained by compiler technology and by the vectorizability of our code. We compile our benchmarks using a recent version of gcc. While gcc supports vectorization and in fact generates substantial vector code on our benchmarks, it is not as aggressive as the best available compiler. Hence, our results on subword SIMD, while based on standard compiler technology, are not an upper bound. From examining the generated code, we know that gcc vectorizes loops in 5 of the 6 bench-

marks (MRI, facedetect, povray, ode, and blender). The Intel compiler does a better job vectorizing povray, ode, and blender, but still vectorizes only a small portion of the code. We see 10-15% performance improvement on native runs using Intel binaries versus gcc. Even if this were entirely due to superior vectorization it would not alter the conclusions of the paper.

We do not model overheads for task creation and destruction. The only thread startup cost we model is flushing dirty data from caches. VISBench tasks are fairly course-grained (typically hundreds of thousands of instructions as described in Section 2). As a result, even at the chip level we are only initiating a new task every few to every few hundred cycles. Hence, we believe our assumption is valid; essentially we can build a task management system that alleviates task creation overheads as a major performance factor.

We do not fully model the on-chip interconnect. We assume that a network can be designed that allows essentially full utilization of the global cache, achievable by somewhat overprovisioning the network. Instead, we model the interconnect as having a fixed latency and model bank contention at the global cache.

For tractability and focus, we isolate the effects of area and do not consider power in our analysis. Maximizing performance per watt requires minimizing energy per operation. In general, the more complex the pipeline, the greater the number of gates and latches a given operation will need to traverse, and hence the greater the energy consumed per operation. As a result, a power supply constraint generally favors a simpler and in particular a shorter pipeline.

6 Related Work

Related work falls into three categories: benchmarking of parallel applications, accelerator architectures, and design space exploration for parallel architectures.

A number of benchmark suites have been published in the area of parallel computing, including SPLASH [35] and SPEComp [9], which target HPC applications. The PARSEC benchmark suite [11] targets Recognition, Mining, and Synthesis application areas, a similar but much broader application area than what we examine. Other examples of benchmark suites include MediaBench[27], targeting multimedia applications, and EEMBC [16], targeting applications for embedded computing. Ad-hoc benchmarks have been published for specific application areas included in our study. Benchmarks for graphics rendering include 3DMark by Futuremark, used to measure real-time rendering performance. The graphics benchmarks we use in this paper differ in that they are aimed at high-fidelity, non-real-time rendering. PhysicsBench [38] is a benchmark suite for physics simulation which we use in this paper.

Accelerator architectures are becoming increasingly important as a number of vendors have proposed or are providing accelerator chips. These make up the general notion of an xPU [20], a co-processor accelerator more general than a traditional graphics chip. Examples include Intel's 80-core VLIW research chip[17], the Cell processor used in the PlayStation 3[19], and Tilera TILE64 [4]. GPUs are moving in the direction of general purpose accelerators. The use of GPUs as general purpose accelerators is the subject of GPGPU research. [29] provides a survey of this work. NVIDIA's CUDA [7] and AMD's CTM [6] provide programming interfaces for GPGPU programming. Application specific architectures have been proposed for some of the applications we examine. Ageia PhysX chip [1] and ParallAX [38] accelerate physics simulation. NETRA [14] was a parallel architecture for computer vision. The Ray Processing Unit [36] is an accelerator for ray tracing.

Our study extends a large body of work on area-efficient architecture. A number of CMP studies have focused on area-tradeoffs for maximizing throughput/area or throughput/watt for general purpose workloads. In [22], Huh et. al. compare fixed-area CMPs made up of either in-order or out-of-order cores. Kumar et. al. examine the microarchitectural optimization of cores [23] and on-chip interconnects [24]. [10] also examines on-chip communication networks. [30] studied the impact of core count, cache hierarchy, and interconnects on CMP power consumption. [28] perform a design space exploration with core count and core complexity under various power and area constraints. [21] studies the cache design space for many-core CMPs.

These previous studies have influenced our methodology for modeling processor area. However, they have generally examined a very different set of architectural parameters and using a very different set of applications. Whereas we examine architectural choices like multithreading and SIMD execution, the prior work tends to emphasize interconnection networks. Both our work and the prior work examine cache hierarchy, dynamic scheduling, and core count. The previous work also focuses on purely general purpose architecture, whereas we examine an accelerator architecture that lies between general purpose and application-specific.

7 Conclusions

In this paper, we examine workloads in the visual computing application class. We compile a benchmark suite, VISBench, to serve as a proxy for this application class. We use VISBench to examine some important high level decisions for an accelerator architecture. We examine the need for synchronization and data communication. We also examine GPU-style SIMD execution and find that while SIMD is preferrable for some applications, for most applications MIMD provides higher performance.

We use VISBench to perform a detailed area-performance tradeoff study. We find that optimal performance is attained by relatively simple cores. We find that

fine-grained multithreading improves performance, but only up to a point. We also find that word-level SIMD provides a poor performance to area ratio on our set of applications.

8 Acknowledgments

This research was supported by the Focus Center for Circuit & System Solutions (C2S2), one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program.

The MRI reconstruction kernel was provided by Sam Stone, while the motion estimation kernel was provided by Chris Rodrigues, both of the IMPACT group at UIUC. PhysicsBench code was provided by Tom Yeh and Glenn Reinmann of UCLA.

John Kelm, Steve Lumetta, Matt Frank, and Wen-mei Hwu all provided valuable assistance in developing this project.

References

[1] AGEIA PhysX. http://www.ageia.com.

[2] MIPS32 74K. http://www.mips.com/products/cores/32-bit-cores/mips32-74k/index.cfm.

[3] Tensilica Diamond 570T. http://www.tensilica.com/diamond/di_570t.htm.

[4] Tilera TILE64 Processor Overview. http://www.tilera.com/pdf/Pro-Brief_Tile64_Web.pdf.

[5] The International Technology Roadmap for Semiconductors 2005 Edition, System Drivers, 2005.

[6] ATI CTM Guide, 2007. http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[7] CUDA Programming Guide 1.0, 2007. http://developer.nvidia.com/ob-ject/cuda.html.

[8] Aqeel Mahesri et al. Tradeoffs in designing accelerator architectures for visual computing. Technical Report UILU-ENG-08-2208, University of Illinois, May 2008.

[9] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. *Lecture Notes in Computer Science*, 2104, 2001.

[10] J. Balfour and W. J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *ICS-20*, pages 187–198, 2006.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Princeton University, January 2008.

[12] Blender.org. Blender. http://www.blender.org.

[13] D. Bolme, M. Strout, and J. Beveridge. Faceperf: Benchmarks for face recognition algorithms. *Workload Characterization, 2007. IISWC 2007*, pages 114–119, 27-29 Sept. 2007.

[14] A. N. Choudhary, J. H. Patel, and N. Ahuja. NETRA: A Hierarchical and Partitionable Architecture for Computer Vision Systems. *IEEE Trans. Parallel Distrib. Syst.*, 4(10):1092–1104, 1993.

[15] R. L. Cook, L. Carpenter, and E. Catmull. The REYES image rendering architecture. In *ACM SIGGRAPH*, July 1987.

[16] EEMBC. Embedded Microprocessor Benchmark Consortium. http://www.eembc.org.

[17] S. V. et. al. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *ISSCC Digest of Technical Papers.*, February 2007.

[18] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Micro-40*, December 2007.

[19] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[20] P. Hester. *Multi-Core and Beyond: Evolving the x86 Architecture*. AMD, Aug 2007. HotChips presentation.

[21] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale CMPs. *ACM SIGARCH Computer Architecture News*, 33(4):24–33, 2005.

[22] J. Huh, D. Burger, and S. Keckler. Exploring the design space of future CMPs. In *PACT2001*, pages 199–210, 2001.

[23] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06*, pages 23–32, New York, NY, USA, 2006. ACM.

[24] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads, and Scaling. In *ISCA-32*, 2005.

[25] T.-J. Kwon, J. Sondeen, and J. Draper. Design trade-offs in floating-point unit implementation for embedded and processing-in-memory systems. In *IEEE International Symposium on Circuits and Systems*, volume 4, May 2005.

[26] H. A. Landman. Visualizing the Behavior of Logic Synthesis Algorithms. In *SNUG 98: Proceedings of the Synopsys User Group Conference*, 1998.

[27] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Micro-30*, 1997.

[28] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP Design Space Exploration Subject to Physical Constraints. In *HPCA-12*, 2006.

[29] D. Luebke, M. Harris, J. Krger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. GPGPU: general purpose computation on graphics hardware. In *ACM SIGGRAPH*, August 2004.

[30] M. Monchiero, R. Canal, and A. Gonzlez. Design space exploration for multicore architectures: a power/performance/thermal view. In *ICS-20*, pages 178–186, 2006.

[31] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches With CACTI 6.0. In *Micro-40*, December 2007.

[32] S. S. Stone, H. Yi, W. mei W. Hwu, J. P. Haldar, B. P. Sutton, and Z.-P. Liang. How GPUs Can Improve the Quality of Magnetic Resonance Imaging . *The 1st Workshop on GPGPU*, 2007.

[33] I. Wald, C. P. Gribble, S. Boulos, and A. Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, 2007.

[34] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2005.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA-22*, pages 24–6, 1995.

[36] S. Woop, J. Schmittler, and P. Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

[37] L. Yang, K. Yu, J. Li, , and S. Li. Prediction-based Directional Fractional Pixel Motion Estimation for H.264 Video Coding. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2005.

[38] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinmann. ParallAX: An Architecture for Real-Time Physics. In *ISCA-34*, 2007.