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 is usually preferable.

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

There is a large, emerging and commercially-relevant class of applications enabled by the significant increase in computing density provided by accelerator architectures such as graphics processing units, physics accelerators, and attached co-processors such as the IBM Cell processor. These applications span many domains, from graphics and physics for gaming and interactive simulation, to data analysis for oil and gas exploration, scientific computing, 3D modeling for CAD, signal processing, digital content creation, and financial analytics. Applications in these domains benefit from architectural approaches that provide higher performance through parallelism.

In this paper we examine architecture-level tradeoffs for applications broadly associated with visual computing. Visual computing is a significant extension beyond standard raster-based graphics processing. It includes myriad graphics algorithms beyond classic raster-based graphics. It further includes application areas such as video processing, computer vision, imaging, tracking, and physics simulation.

To support our exploration, we develop an experimental benchmarking suite called VISBench (Visual, Interactive, Simulation Benchmarks) to serve as an experimental proxy for this visual computing domain. While the VISBench Suite * is a diverse workload, one major aspect is uniformly true: each application's execution time is dominated by a parallel section of code. Visual computing and VISBench are further described in Section 2.

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 offload and accelerate compute intensive work. Our meta-architecture goes beyond existing multicore designs in relying on parallelism for performance, and in relying on software to manage the parallelism. Our meta-architecture is built around a large array of simple compute cores, connected by a minimal on-chip network to a shared cache and a very high bandwidth memory system. The compute array is managed by a high-performance controller to handle scheduling and communication. In addition to proposing a basic architecture, 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 implementation of such synchronization can be made much simpler and slower than in a multicore CPU. We determine that for some applications, SIMD execution provides a major performance benefit, but other applications'

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

control flow divergence results in an even greater performance loss. The meta-architecture is further described in Section 3.

Our meta-architecture affords a wide degree of freedom in determining the detailed architecture. To analyze the design space, we've developed an area model from base technology parameters that we use for assessing the costs for various architectural choices. Using a performance model, we estimate the throughput attained by the overall chip architecture on VISBench applications. We explore the performance and area tradeoffs for a number of these 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. The design space exploration is presented in Section 4.

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 and rendering of visual information. Examples of applications in this domain include rendering of 3D scene information, photo-realistic scene lighting models, physical simulation, medical and scientific visualization, and image and video processing. Applications of this domain represent a significant, emergent, and commercially-relevant class of performance drivers for consumer, professional, and high-end computing. Higher baseline performance on such applications 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 given that they deal with large volumes of data that are visualized in a largely independent manner, or have interactions that are mostly uncommon or localized.

From another perspective, 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 [24]).

2.1 VISbench

To address the question of xPU architecture, we have created 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 (such as video games).

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.

We devote the following subsections to briefly describing the benchmarks and their chosen input sets, and outlining the intrinsic parallelism within the application.

2.1.1 Scanline rendering: Blender renderer

Blender [12] is a free-software animation and 3D modeling program. It can be used for a variety of tasks, including modeling, texturing, skinning, animating, rendering, and creating interactive 3D applications. We include Blender's internal 3D renderer as a VISBench application.

Blender supports multithreaded rendering. Each render can be broken up into tiles, with each tile rendered in a separate thread. Each tile can be as small as a pixel quad (4 pixels). For this study we replace the multithreading with annotations, as described in Section 4. 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. This image can be rendered by up to 76,800 threads, ranging in length from 500K to 2.4M instructions. In our study we simulate roughly one thirtieth of the render over 2B instructions.

2.1.2 Ray tracing: POVRay

An alternative to scanline rendering is ray tracing. Ray tracing is one of a number of highly compute-intensive global illumination methods; that is, methods that render not only direct lighting but also reflection, refraction, and diffusion. The ray tracing algorithm mimics optics by tracing the path

[†] It is our intention to publicly release VISBench in the coming months

taken by rays of light. The ray tracing algorithm is embarrassingly parallel. Each pixel calculation is independent of every other pixel. Although different pixels may intersect with the same object, requiring shared data structures, they need not write to any shared data.

As with Blender, we use a serial version of this code with annotations around the parallel loops. 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 384×384.

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. In our study we simulate roughly one tenth of the render in 2B instructions.

2.1.3 Video encoding: H.264 motion estimation kernel

Modern video encoders take a sequence of raw images and compress them, typically using a motion compensation-based encoding technique. There are a variety of video codecs currently in use, including H.262, H.263, and MPEG-2. An emerging codec for high definition video content is H.264, also known as MPEG-4 AVC.

The compute-intensive portion of the encoding pipeline is motion prediction, accounting for over 90% of execution time even for standard definition content. Motion compensation is a technique for describing a macroblock of pixels within an video frame relative to a block of pixels in a reference frame.

The most suitable motion vector is one that minimizes residual encoding information, and this is often done by computing the sum of the absolute differences (SAD) between each pixel in the translated macroblock in the reference frame versus the corresponding block in the current frame. The SAD computation is fast and simple, and easily parallelized.

The SAD kernel is embarrassingly parallel and can be parallelized in a number of ways. It can be vectorized to use SIMD instructions such as SSE. It can be parallelized to run on a GPU using an appropriate programming model (e.g., CUDA [8]). We use a standard CPU version of the kernel, written in C, with annotations around the parallel loops. 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 22K instructions long. In our study we simulate roughly one fifth of the computation in 2B instructions.

2.1.4 Dynamics simulation: ODE PhysicsBench

Simulation of rigid body physics is useful in a variety of contexts, including scientific applications, engineering mechanical systems, and for adding interactivity and realism to video games. Open Dynamics Engine (ODE) is a free library for simulating articulated rigid body dynamics [42]. PhysicsBench [47] is a suite of physical simulations built using the ODE library. A parallel implementation of PhysicsBench and ODE is described in [47]. The structure of computation in ODE is very different from graphics rendering and SAD computation, which are embarrassingly parallel if properly coded. Physics computation exhibits distinct phases with distinct levels of available parallelism. Broadly, one step of physics simulation consists of collision detection, followed by a solver which determines the forces and motion of interacting objects.

Our test simulation is the mixed simulation from PhysicsBench. The simulation contains 1608 objects, 933 of them as boxes in breakable walls. This is a fairly small scene, yet is still sufficient to allow the creation of enough threads to fill a hugely parallel xPU. Collision detection generates 61 threads ranging from 390K to 2.3M instructions, the constraint solver generates 192 threads ranging from 127K to 282K instructions, while cloth simulation generates 7000 threads ranging from 12K to 525K instructions. We simulate one timestep in roughly 900M instructions.

2.1.5 High quality MRI

In magnetic resonance imaging (MRI), scan data from a magnetic resonance scan is reconstructed into a 3D image of the object being scanned. Due to the physics involved, better images can be obtained by using a non-Cartesian scan trajectory; that is, sampled data points are not taken on a Cartesian grid.

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 $\left[F^H d\right]_n = \sum_{m=1}^{M} \phi^*(k_m) d(k_m) e^{(j2\pi k_m x_n)}$.

The computation of Q is performed offline, based only on the position of each sample. The computation of $\mathrm{F}^H\mathrm{d}$ depends on the actual data values obtained from the scanner. Both computations are nearly identical; we use only the $\mathrm{F}^H\mathrm{d}$ computation as a benchmark. We perform the computation of all $\mathrm{F}^H\mathrm{d}$ values in parallel, with each independent threads 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

Computer vision (CV) deals with systems for obtaining information from images. CV algorithms analyze images or video in order to recognize objects and reconstruct models of the scene. One use of CV is detection and recognition of human faces, important in such varied applications as autofocus in digital cameras, biometrics, and video surveillance.

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 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 classifer, 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 data set the application is operating on and the user experience. For instance, the quality of a rendered image can be increased by rendering more polygons, the fidelity of a physical simulation can be improved by modeling more objects, and appearance of a video image can be enhanced by increasing its resolution. 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.

It is uniformly the case that each of the VISBench applications contains short section(s) of serial code in addition to the time-intensive parallel portions. Running the VISBench applications with the inputs sets we studied, we find that over 95% of execution time on a serial machine is spent in portions deemed parallel by our parallelization model.

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 is an important thing to note as it implies that an accelerator architecture can be scaled through parallelism with Moore's Law and still provide value to the same application base.

[‡]O(n) with significant constants or worst case $O(n^2)$

3. Accelerator Meta-architecture

In this section, we examine the architecture for a visual computing accelerator at a high level. We introduce the basic architecture of the accelerator as a massively parallel co-processor that can speed up compute-intensive parallel portions of VISBench. Given this basic architecture, there are a number of important high-level issues to examine, including synchronization and data communication, and the execution model within the parallel compute fabric.

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

In addition, 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 be strongly preferable over the broadest set of applications.

3.1 Acceleration model

This study examines the accelerator co-processor execution model. That is, we are interested in mapping the long-running, compute-intensive portions of the VISBench benchmarks onto an accelerator architecture that is attached to a host CPU via a commodity system interconnect such as PCI Express or HyperTransport. In this paper, we specifically examine the chip-level tradeoffs associated with designing the accelerator architecture for maximizing throughput on these compute-intensive portions.

In this accelerator model, both the CPU host and the xPU have memories that are disjoint from the programmer's perspective and must be managed via explicit DMAs. A variety of strategies have been deployed over time on GPUs and on Cell for amortizing the latency, reducing the bandwidth, and reducing programmer burden [28] associated with the separated memories. In this paper, we focus in on the compute-intensive portions assuming that the transfer latencies between the host and xPU can be overlapped or are small.

3.2 Basic Architecture

Figure 1 shows the basic architecture of an xPU. The xPU consists of a compute array of a large number of minimal 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. The system can transfer data and commands to the xPU through DMAs. On the xPU side, communication with the rest of the system is handled by a single controller core, which is also responsible for initializing the program to be run 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. The global cache is a large, multi-banked cache accessible to all the cores in the array. The memory space, which is shared by all cores, is striped across the global cache banks at the granularity of a cache line, with one access allowed per bank per cycle. This arrangement allows us to have a very large global cache bandwidth, which in effect multiplies our memory bandwidth.

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

3.3 Synchronization and communication

One important element in the design of a parallel architecture is what support to provide for synchronization and data communication among parallel processing elements. 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.

As we extend the visual computing domain beyond classic raster graphics, we find that at minimum support for a bulk synchronous model is essential. For instance, in ray tracing we need to set up the data structures representing the geometry, then synchronize, and then begin the pixel-by-pixel ray tracing, then synchronize, and then output the result.

Only one of the VISBench benchmarks requires fine-grained synchronization support: ODE-based physics simulation. Physics simulation needs to deal with the occasional concurrent data access from different threads arising from some sort of interaction between the threads. For more information on the fine-grained data access, see [47].

However, while fine-grain synchronization support is required in general for Visual Computing, it is often not performance-critical (because the locking occurs with low frequency or the conflicts are rare), and can be isolated, as in the case of the physics application, by pre-mapping the interacting objects to co-located or non-concurrent threads [14]. Slow fine-grained synchronization can be implemented on a non-coherent machine by performing primitive operations at the shared level and keeping synchronization variables out of private caches, and for VISBench applications this is sufficient.

The data communication patterns in VISBench 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. Figures 3 and 4 show the distribution of loads and stores among data that is shared, input, and private. Most of the loads on most benchmarks are private. However, a large fraction of the loads (20%-83%) are input reads to data with multiple readers. The frequency of accesses to such read-shared data suggests that a shared memory model would be preferable to message passing.

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. In VISBench applications, stores to non-private data are rare, and while loads from non-private data 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.

Hence, we envision the xPU having a non-coherent shared memory. In such a model, stores to shared data must be communicated explicitly. Specifically, all stores to such data must propagate at least to the global cache, where it becomes visible to all readers. Furthermore, all stores to output data must be flushed out of private caches at synchronization barriers. Likewise, loads that may access write-shared data must also propagate at least to the global cache. Essentially, we require cache coherence to be maintained in software.

3.4 MIMD execution

Traditional supercomputer architectures were often vector machines that operated on large vectors. This sort of global SIMD, different from the 128-bit SIMD found in

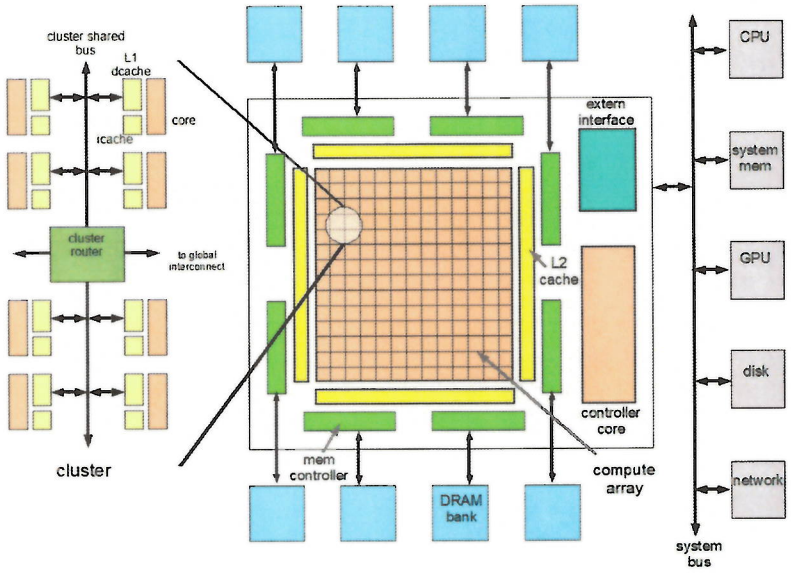

Figure 1. Block diagram of accelerator

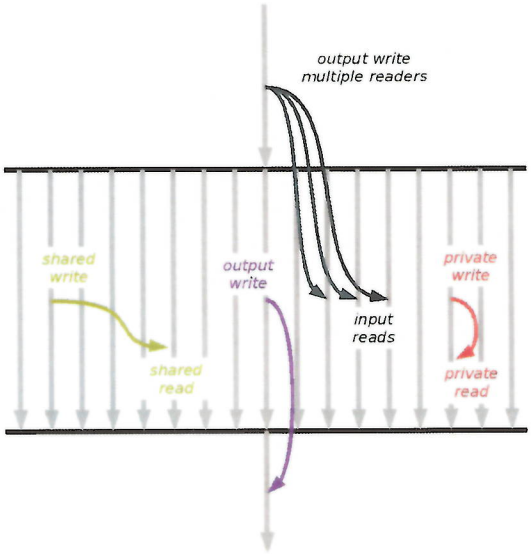

Figure 2. Bulk synchronous parallelism

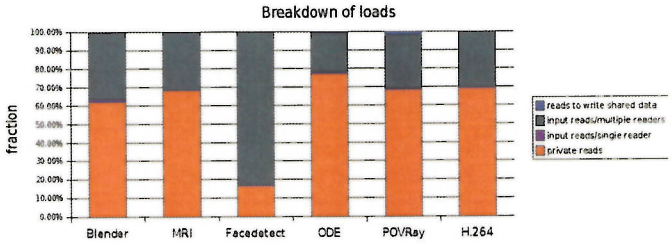

Figure 3. Loads to shared versus private data

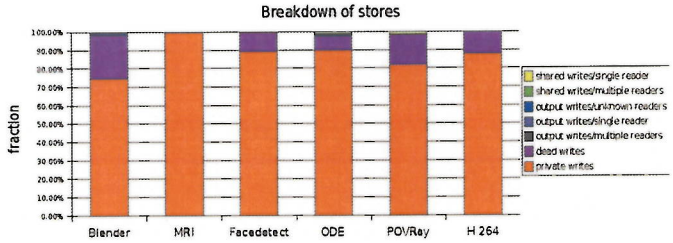

Figure 4. Writes to shared versus private data

most modern CPU architectures, was very effective at obtaining high performance on numerical applications. Modern graphics chips rely on SIMD execution for executing programmable kernels.

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 all of the parallel threads would be forced to take the longest control flow path.

To determine whether SIMD or MIMD is preferable, 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 into warps, with the threads in a warp operating on consecutive data elements to minimize the potential for divergence. As long as all threads in the 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, they once again execute in lockstep. We use the immediate postdominator of the divergence point as the reconvergence point, which has been found to be near optimal on real programs [21]. 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 5 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.

We want to compare the performance of MIMD and SIMD relative to 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-wide SIMD cluster has 1.6 times the area of a scalar pipeline, while a 4-wide has 2.8 times the area.

Figure 6 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 had 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 in practice this result is an upper bound (outside of a restructuring of the application). We must also point out that Blender and H.264 required a moderate (but reasonable) amount of

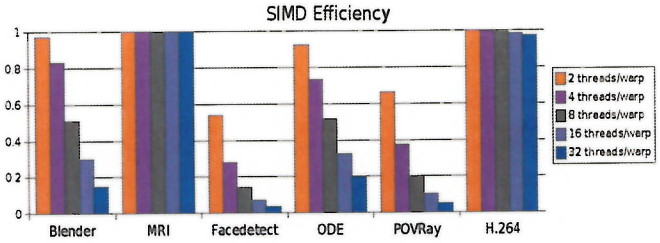

Figure 5. SIMD efficiency for varying warp sizes

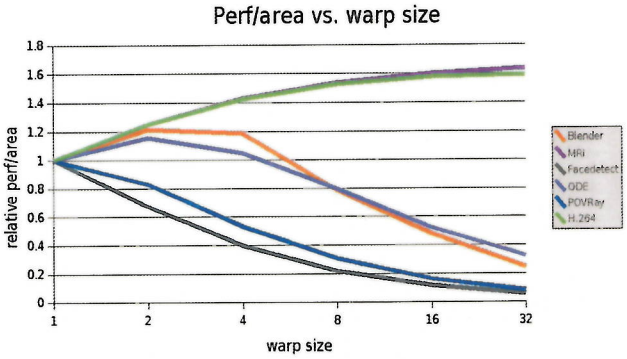

Figure 6. SIMD performance per area for varying warp sizes

hand tuning in order to achieve their level of SIMD performance. With substantial 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.

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. Furthermore, we examine these effects relative to

their cost in chip area.[§]

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 each compute core, at what level of sequential performance, and with what architectural and microarchitectural features. Furthermore, we examine the cache design for the chip, examining the tradeoff due to varying the fraction of the chip devoted to caching versus logic.

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. A variety of methods are available for estimating the area from a given core architecture.

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 [26]. [36] 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. Area estimates were obtained based on a detailed design of each of the pipeline configurations. A 65nm process is assumed for all estimates. Each configuration was decomposed into its major components, and area models were derived from base component counts. Cache and memory structures are modeled using CACTI-6 [37] when possible. Synthesis results for ALU and FPU components are used when possible ([44], [31]). When synthesis results are not available for components, we obtain area estimates by designing the structures down to the logic level and counting of the number of gates, flops, and SRAM bits required in the implementation. When necessary, data from earlier process generations is scaled to the 65nm node. Standard cell density is assumed to be 75% of the maximum logic density. Detailed information on VLSI circuit parameters such as average gate size may be found in [6].

We compose these components in order 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.

For each basic configuration, a 1GHz (low-speed) and 2GHz (high-speed) variants are considered. It is expected that the high-speed variant will have more intensive custom design effort applied to meet timing and area targets. These clock targets correspond roughly to 40FO4 and 20FO4 respectively [40].

The 1-wide configuration consists of a standard 5-stage fully-bypassed, in-order pipeline with a static BTFN predictor. The high-speed variant has a 9 stage pipeline and uses a simple bimodal 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 high-speed pipeline model has 10 stages and a bimodal predictor.

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. The high-speed design has 10 stages, uses a more sophisticated YAGS predictor [18], and doubles the size of the ROB and scheduler.

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. A hardware implementation of trigonometric and trascendental functions is described in [41]. We model this unit having a delay of 6 cycles in the low frequency case and 10 cycles in the high frequency case, and pipelined with 2 stages such that operations can begin once every 3 or 5 cycles, respectively.

Table 2 shows area by major hardware function for each of the basic configurations. Frontend includes fetch and decode as well as scheduling and renaming as appropriate. Other includes remaining logic and pipeline registers.

[§]Power, like area, is a dominant component of cost. For tractability and focus, we isolate the effects of area in the experimental section, and treat power in our discussion in Section 5.

Table 1. 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	8 channel, 64-bits/channel, 1GHz DDR

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.

Our area numbers are not meant to be "canonical" area numbers for the given configuration. 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 [32].

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 [3] and Tensilica 108Mini and 570T[4]. These commercial cores contain additional logic for features that our cores lack. However, they show that the core areas shown in Table 2 are achievable. Nvidia's G80 is essentially a chip multiprocessor with significant graphics-specific hardware. Intel's 80-core Teraflop Research Chip [43] and IBM's Cell [27] target high clock frequencies with a penalty in area. Table 3 shows area data for existing designs.

Table 3. Area Comparison of Commercial Cores (Normalized to 65nm)

	Area in mm^2
Tensilica 108Mini	.143
Tensilica 570T	.349
MIPS 74k	1.7 to 2.5
Nvidia G80	1.87
Nvidia G92	2.46
Intel 80-Tile	2.4
IBM Cell 4x Vector FPU	.65

4.2 Performance Modeling Methodology

Our performance modeling methodology is driven by sequentialized versions of each application, versions that contain the algorithm and data structures of the parallel version, but with the actual threading calls removed. In their place, we insert dummy function calls that serve as annotations to mark the boundaries of parallel execution. We place these annotations around the parallel loops and at the beginning and end of each iteration.

We run our annotated binaries 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-accurate timing model simulates the performance of all cores. The submodel for each core executes the instruction traces of the threads it has been assigned.

In addition to cores, we model the cache hierarchy as proposed earlier. We model the global cache as multi-banked with 1 access allowed per bank per cycle. The main memory is modeled as having 8 channels, with memory addresses striped across the channels.

Note that we have access to parallel implementations of each of our benchmarks; however, the parallel versions of our benchmarks are unsuitable for this study. The parallel implementations suffer from performance overheads due to system calls for thread creation and scheduling. In our generalized accelerator model, we assume that such tasks will

Table 2. Area breakdown by pipeline component in mm^2 (Low-Speed variants)

	Frontend	Execution	Caches (4KI/8KD)	Other	Core(total)	+SIMDx4	+MTx2	+MTx4	+Transcendentals
1W in	0.016	0.068	0.140	0.009	0.330	0.200	0.032	0.096	0.019
2W in	0.066	0.092	0.150	0.026	0.420	0.250	0.074	0.220	0.019
2W out	0.200	0.092	0.150	0.057	0.590	0.250	0.120	0.360	0.019

be performed in hardware. We remove these overheads in order to focus on the requirements for the compute portion of the accelerator.

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 1. We assume that we have an area budget of $300mm^2$ for the compute array and the shared cache, the portions of the accelerator architecture we are studying. We divide this area into $200mm^2$ 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). This value is consistent with the cache sizes seen on current generation microprocessors. We also assume a fixed memory bandwidth of 128GB/s; memory bandwidth is a function of the number of pins available on a package, and tends to have an upper bound in any given process generation. The gcache bandwidth we need to fully utilize the memory bandwidth is the memory bandwidth divided by the expected miss rate. With a miss rate of 12%, 32 gcache accesses per cycle (1024GB/s) is a reasonable value.

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 7 plots the performance per area (total throughput divided by the $200mm^2$ of area used by the compute fabric) 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, in1 configuration 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 such as ours 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 due to the increased area overhead from the scheduling logic blah.

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), which most high performance architectures feature. Examples include AltiVec on PowerPC and SSE on x86.

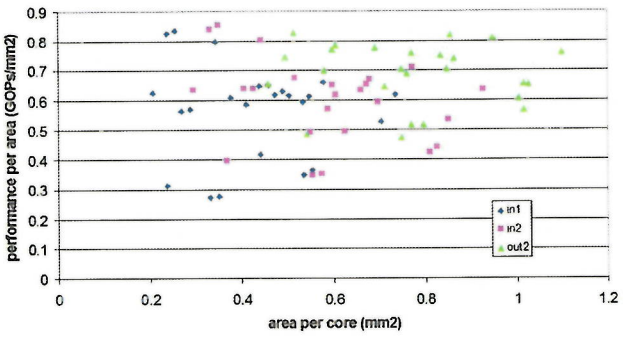

Figure 7. Performance for different pipeline configurations

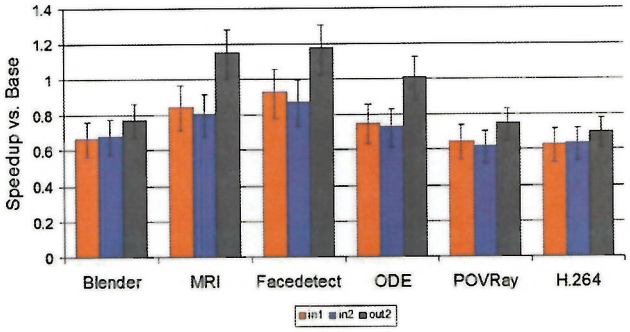

Figure 8. Relative performance of chip with SIMD instructions

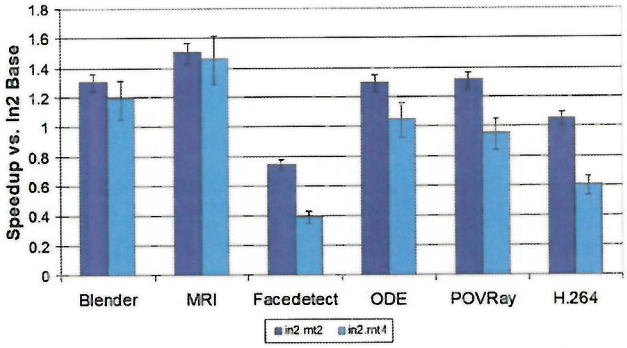

Figure 9. Performance of 2-wide in-order configuration with multithreading

Figure 8 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 is 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.

Traditional GPU architectures also used 4-wide vector SIMD, in addition to lockstep execution. However, the newest GPUs have moved to scalar shader pipelines, reinforcing our result. One should note that word-level SIMD makes more sense on a processor optimized for single-thread performance (such as a multi-core desktop CPU) where area per core is large and the cost of SIMD support is relatively minor.

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 Figures 9, 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 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 average stall cycles. Furthermore, the area cost of multithreading grows linearly as thread count, as you need to continue replicating structures such as the register file.

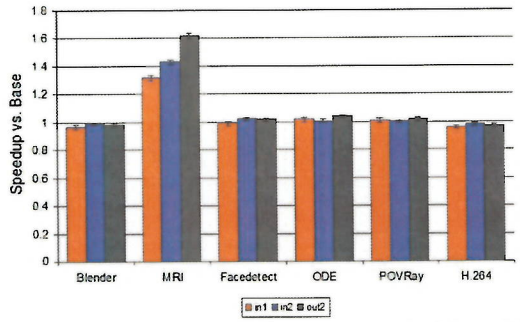

Figure 10. Relative performance of chip with hardware trig unit

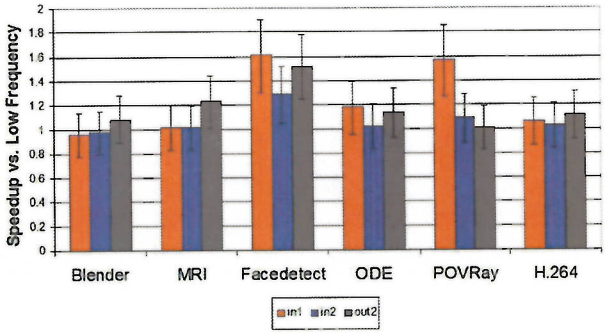

Figure 11. Performance for high frequency chip

4.7 Hardware Implementation of Trigonometric and Transcendental Functions

VIS workloads rely heavily on a small set of complex functions, such as sin, cos, atan, exp, sqrt, and rsqrt for calculating transforms, angles of incidence, and various other geometric and visual actions. Table 4 shows the frequency of such operations in VISBench.

Figure 10 shows the relative performance of the configurations with an added hardware support for trig and transcendentals. The figure shows improved in-order performance with the trig units. The performance benefit is particularly strong for MRI, which has the heaviest concentration of such operations, with one of every 32 operations being either sin or cos. SAD, on the other hand, shows very little performance effect from such units. Facedetect, ODE and POVRay, meanwhile show a small performance benefit, roughly within the margin of error. The actual trig calculation is performed with a series of table lookups, FP adds, and FP multiplies, each of which is sequentially dependent. Hence, a slow trig calculation occupies the FP unit and blocks progress in the in-order machine but not the out-of-order machine.

4.8 High Frequency Pipelines

ASIC design can be carried out using one of several different design flows, each of which is targeted to a different level of performance. A low-frequency design is carried out using a different design flow than high-frequency design, with high-frequency design requiring some amount of custom design and placement.

A core designed to achieve high frequency will utilize both deeper pipelining and circuit level techniques such as the use of fast logic styles and resizing transistors for minimum delay. Both factors are likely to increase area and design time. Given this, as well as power considerations, we would need to see a clear benefit to a higher clock rate in order to justify high frequency.

We assume we can design a high frequency running at 2GHz core. Such a core would need to have both a deeper pipeline and faster logic for individual components. The 5 stage single-issue pipeline grows to 9 stages, while the 6 stage dual-issue pipelines grow to 10 stages. Furthermore, the area cost of integer and FP ALUs grows considerably in order to maintain the same latencies in terms of the number of cycles.

Figure 11 shows the speedup from our high frequency design. The figure shows, for most benchmarks and pipeline configurations, a small performance benefit, within the error margin of our area model.

4.9 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 12 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. One the other hand, the configurations with the smallest and largest cache sizes perform worse.

4.10 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 enjoyed

Table 4. Frequency (per 1000 instructions) of trig and transcendental operations

benchmark	sin	cos	atan	sqrt	exp	total
blender	.013	.006	.006	.06	0	.089
mri.fh	15.0	15.0	0	0	0	30.0
facedetect	0	0	0	.59	0	.59
povray	.001	.08	.03	.03	.52	1.7
ode	.02	0	.002	0	.01	.03

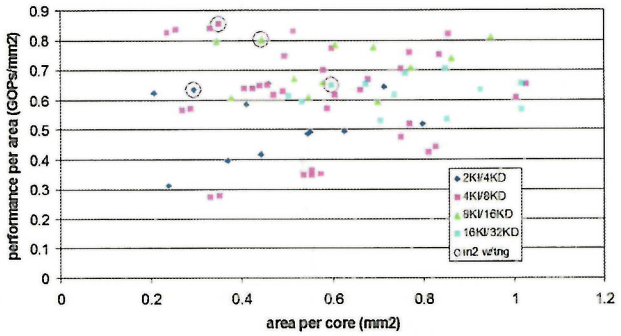

Figure 12. Scatter plot highlighting different cache sizes. Circled data points are examined in detail

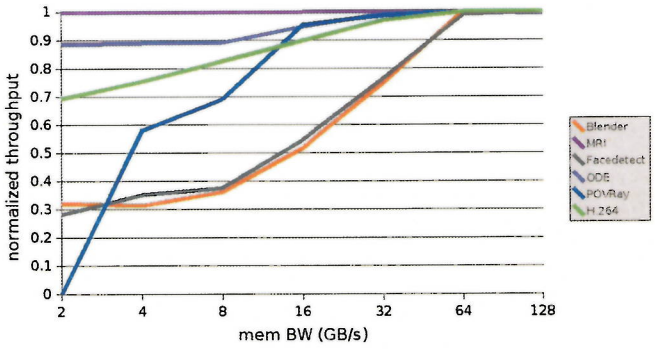

Figure 13. Performance versus memory bandwidth (normalized to maximum)

by the latest high-end GPUs. Figure 13 plots the performance of the low frequency 2-wide in-order configuration with varying memory bandwidths. From the graph, it is apparent that performance on the more bandwidth-intensive apps saturates as the bandwidth reaches 64GB/s, indicating that 128GB/s is an appropriately sufficient bandwidth for the xPU configurations we evaluate.

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 14 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 an appropriately sufficient level.

Note that this result was achieved by performing a few simple optimizations on 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.

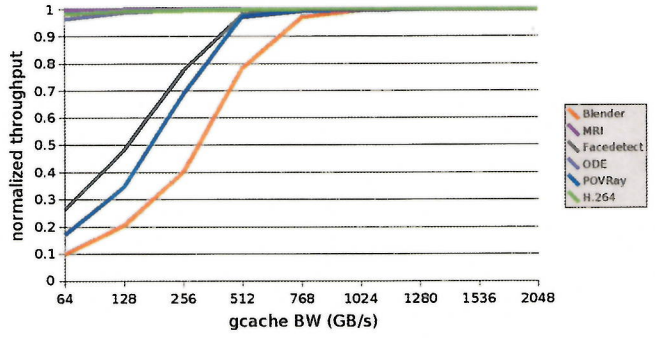

Figure 14. Performance versus gcache bandwidth (normalized to maximum)

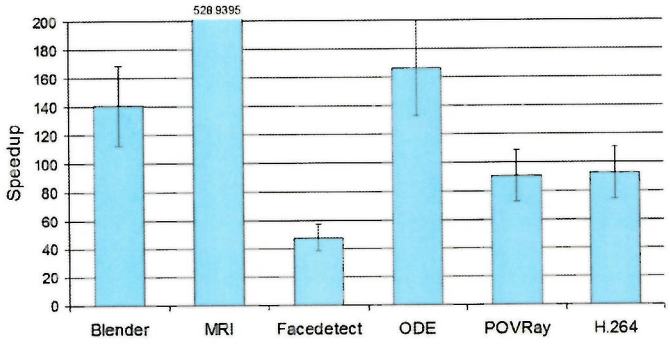

Figure 15. xPU performance versus 2.2GHz Opteron

4.11 Summary: an xPU prototype

Pulling the optimal result from Figure 7, 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 15 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 Discussion

In this section we discuss a few important topics that are beyond the scope of our experiments.

5.1 Power consumption

Power consumption is a first-order consideration consideration in processor design. Often, a chip's clock speed is constrained by a power supply or cooling limitation rather than cycle time. High performance architectures are developed to fit within a certain maximum power budget under all conditions. The design goal is to achieve maximum performance without exceeding the power envelope.

The power consumed by the chip is equal to the energy consumed per operation times the throughput in operations per second. If power is the limiting factor, then throughput is maximized by minimizing the energy consumed 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.

We leave it to future work to develop a power model and revisit our conclusions in the presence of a power budget. We do not expect power considerations to have a major impact regarding our conclusions about SIMD (at either level), hardware supported trancendental functions, or the effect of memory bandwidth. It may impact our conclusions regarding multithreading, dynamic scheduling, and superscalar execution, as well as cache sizing.

5.2 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 the industry standard 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 realistic compiler technology, are not an upper bound.

Another limitation is a result of our simulation methodology. Because of the slow speed of simulation, we must use reduced size inputs in order to capture overall program behavior within the limited run time. The reduced inputs understate the level of parallelism available. They may also stress the memory system less than large inputs, though presumably we would be able to block the computation to alleviate this stress.

We do not model overheads due to synchronization at barriers. We assume that we can implement a fast barrier mechanism, and with our shortest threads running around 20K instructions, the relative overhead of barriers should be small. The only thread startup cost we model is flushing dirty data from caches; again, we assume that task queues can be implemented efficiently. On the other hand, we do model the cost of the non-temporal loads and stores required for our software coherence model from Section 3.

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, which is achievable by somewhat overprovisioning the network. Instead, we model the interconnect as having a fixed latency and model bank contention at the global cache.

6 Related Work

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

6.1 Benchmarking of Visual Computing

A number of benchmark suites has been published in the area of parallel computing. An early effort to provide

a parallel benchmark suite was SPLASH [45], which consists of numerical kernels and some parallel applications ported to a variety of parallel architectures. Another parallel benchmark suite is SPEComp [9], which consists mainly of OpenMP versions of SPECfp. The 13 Dwarves [20] are a set of kernels important to parallel applications. Our benchmark suite differs from these general purpose parallel benchmarks in that it is targeted at a more specific application area. We also seek to study a new and emerging applications rather than traditional HPC workloads.

Other early examples of benchmark suites include MediaBench[33] and EEMBC [19]. MediaBench targets multimedia and network applications that were emerging applications at the time. EEMBC targets applications, including visual applications, that are important for embedded processors rather than high-performance processors.

More recently, Intel has identified Recognition, Mining, and Synthesis (RMS) [17] as important application areas for upcoming parallel architectures. RMS represents application areas meant to capture emerging uses of computing power, like VISBench, but consists of a broader set of areas. The PARSEC benchmark suite [11] was developed to target these application areas. PARSEC targets a somewhat different application area than VISBench.

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 [47] is a benchmark suite for physics simulation which we use in this paper.

6.2 Accelerator Architectures

Accelerator architectures are becoming more and more important as a number of vendors have proposed or are providing accelerator chips. These make up the general notion of an xPU [24], a co-processor accelerator than a traditional graphics chip. Intel has produced an 80-core VLIW research chip[23]. IBM produces the Cell processor, used in the PlayStation 3[22]. A number of smaller companies have also introduced parallel accelerators, including Tilera [5], ClearSpeed [2], and Ambric [15].

GPUs are moving in the direction of general purpose accelerators. The use of GPUs as general purpose accelerators is the subject of GPGPU research. [35] provides a survey of this work. NVIDIA's CUDA [8] and AMD's CTM [7] provide programming interfaces for GPGPU programming.

Application specific architectures have been proposed for some of the applications we examine. Ageia produces the PhysX chip [1], intended for physics simulation, which consists of an array of vector cores. Another proposed physics architectures is ParallAX [47], a heterogeneous architecture containing both coarse and fine grained cores. NETRA [16] was a parallel architecture for computer vision, consisting of a programmable crossbar and clusters of processors that could operate in SIMD, MIMD, or systolic modes. The Ray Processing Unit [46] is an accelerator for ray tracing.

6.3 Area-efficient architecture

As such, our study extends a large body of work on area-efficient architecture. An early example of area-efficient architecture was the RISC project. The motivation for RISC was to put a whole processor onto a single CMOS chip. [39]. Likewise, the motivation for early CMP work was to put a multiprocessor onto a single chip, thus achieving high-throughput on multiple applications [38].

A number of CMP studies have focused on area-tradeoffs for maximizing throughput/area or throughput/watt. In one early study, [26], Huh et. al. compare fixed-area CMPs made up of either in-order or out-of-order cores, and find that on SPEC workloads the out-of-order configurations with fewer cores still provide higher throughput. Kumar et. al. examine the microarchitectural optimization of cores [29] and on-chip interconnects [30], again using SPEC workloads. [10] also examines on-chip communication networks. [36] studied the impact of core count, cache hierarchy, and interconnects on CMP power consumption. [34] perform a design space exploration with core count and core complexity under various power and area constraints. [25] 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, and consequently uses general purpose workloads, whereas we examine an accelerator architecture that lies between general purpose and application-specific.

7 Conclusions and Future Work

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. VISBench includes the classical visual applications of scanline graphics rendering, ray tracing, and video encoding. It also includes a few emerging visual computing applications, such as physics simulation, high quality MRI reconstruction, and real-time face detection.

We use VISBench to examine some important high level decisions for an accelerator architecture. We define the accelerator model, and propose a highly parallel basic architecture. We examine the need for synchronization and data communication among parallel elements of the benchmark applications. We also examine GPU-style SIMD execution and find that while SIMD is preferrable for some applications, for most applications MIMD provides higher performance.

Given these choices, we use VISBench to explore the design space for the compute portion of the accelerator. We analyze area versus performance tradeoffs in the design of the individual compute cores and in the memory hierarchy. We find that such 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 also provides a poor performance to area ratio, particularly when the VIS applications are compiled using a standard compiler with aggressive optimizations.

8 Acknowledgements

The authors acknowledge the support of 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, and in particular the custom version used in this study, was generously provided by Tom Yeh and Glenn Reinmann of UCLA.

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] ClearSpeed Technology Primer. http://www.clearspeed.com/products/overview.

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

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

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

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

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

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

[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 *Proceedings of the 20th International Conference on Supercomputing*, 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. IEEE 10th International Symposium on*, pages 114–119, 27-29 Sept. 2007.

[14] A. Bond. Havok FX: GPU Accelerated Physics For PC Games. In *Game Developers Conference*, 2006.

[15] M. Butts, A. M. Jones, and P. Wasson. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 55–64, Washington, DC, USA, 2007. IEEE Computer Society.

[16] 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.

[17] P. Dubey. Recognition, Mining, and Synthesis Moves Computers to the Era of Tera. *Intel Technology Journal*, 9(2), 2005.

[18] A. N. Eden and T. Mudge. The yags branch prediction scheme. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 69–77, 1998.

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

[20] K. A. et. al. The Landscape of Parallel Computing Research: A View from Berkeley, 2006.

[21] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*, December 2007.

[22] 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.

[23] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview.

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

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

[26] J. Huh, D. Burger, and S. Keckler. Exploring the design space of future CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.

[27] Hwa-Joon Oh et al. A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor. *IEEE Journal of Solid-State Circuits*, 41:759–771, April 2006.

[28] J. H. Kelm, I. Gelado, M. J. Murphy, N. Navarro, S. Lumetta, and W. mei Hwu. CIGAR: Application Partitioning for a CPU/Coprocessor Architecture. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 317–326, Washington, DC, USA, 2007.

[29] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM.

[30] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads, and Scaling. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, 2005.

[31] 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.

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

[33] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[34] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP Design Space Exploration Subject to Physical Constraints. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[35] 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.

[36] M. Monchiero, R. Canal, and A. Gonzlez. Design space exploration for multicore architectures: a power/performance/thermal view. In *Proceedings of the 20th International Conference on Supercomputing*, pages 178–186, 2006.

[37] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches With CACTI 6.0. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*, December 2007.

[38] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[39] D. A. Patterson and C. H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[40] M. A. H. Ron Ho, Kenneth W. Mai. The Future of Wires. In *Proceedings of the IEEE*, volume 89, April 2001.

[41] M. Y. Siu. A high-performance area-efficient multifunction interpolator. In *ARITH '05: Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 272–279, Washington, DC, USA, 2005. IEEE Computer Society.

[42] R. Smith. Open Dynamics Engine. http://www.ode.org/.

[43] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *IEEE International Solid-State Circuits Conference, 2007. Digest of Technical Papers.*, February 2007.

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

[45] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–6, 1995.

[46] 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.

[47] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinmann. ParallAX: An Architecture for Real-Time Physics. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.