The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing

Manish Arora

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92092-0404
marora@cs.ucsd.edu

***Abstract*— GPU computing has emerged in recent years as a viable execution platform for throughput oriented applications or regions of code. GPUs started out as independent units for program execution but there are clear trends towards tight-knit CPU-GPU integration. In this work, we will examine existing research directions and future opportunities for chip integrated CPU-GPU systems.**

We first seek to understand state of the art GPU architectures and examine GPU design proposals to reduce performance loss caused by SIMT thread divergence. Next, we motivate the need of new CPU design directions for CPU-GPU systems by discussing our work in the area. We examine proposals as to how shared components such as last-level caches and memory controllers could be evolved to improve the performance of CPU-GPU systems. We then look at collaborative CPU-GPU execution schemes. Lastly, we discuss future work directions and research opportunities for CPU-GPU systems.

***Index Terms*—GPU Computing, CPU-GPU Design, Heterogeneous Architectures.**

I. Introduction

We are currently witnessing an explosion in the amount of digital data being generated and stored. This data is cataloged and processed to distill and deliver information to users across different domains such as finance, social media, gaming etc. This class of workloads is referred to as throughput computing applications[1]. CPUs with multiple cores to process data have been considered suitable for such workloads. However, fueled by high computational throughput and energy efficiency, there has been a quick adoption of Graphics Processing Units (GPUs) as computing engines in recent years.

The first attempts at using GPUs for non-graphics computations used corner cases of the graphics APIs. To use graphics APIs for general purpose computation, programmers mapped program data carefully to the available shader buffer memory and operated the data via the graphics pipeline. There was limited hardware support for general purpose programming; however for the correct workload, large speedups were possible [35]. This initial success for a few non-graphics workloads on GPUs prompted vendors to add explicit hardware and software support. This enabled a somewhat wider class of general purpose problems to execute on GPUs.

NVIDIA's CUDA [34] and AMD's CTM [4] solutions added hardware to support general purpose computations and exposed the massively multi-threaded hardware via a high level programming interface. The programmer is given an abstraction of a separate GPU memory address space similar to CPU memory where data can be allocated and threads launched to operate on the data. The programming model is an extension of C providing a familiar interface to non-expert programmers. Such general purpose programming environments for GPU programming have bridged the gap between GPU and CPU computing and led to wider adoption of GPUs for computing applications.

[1]GPU architects commonly refer to these as general purpose workloads as they are not pertaining to graphics. However, these are a portion of the CPU architects definition of general purpose, which consists of all important computing workloads.

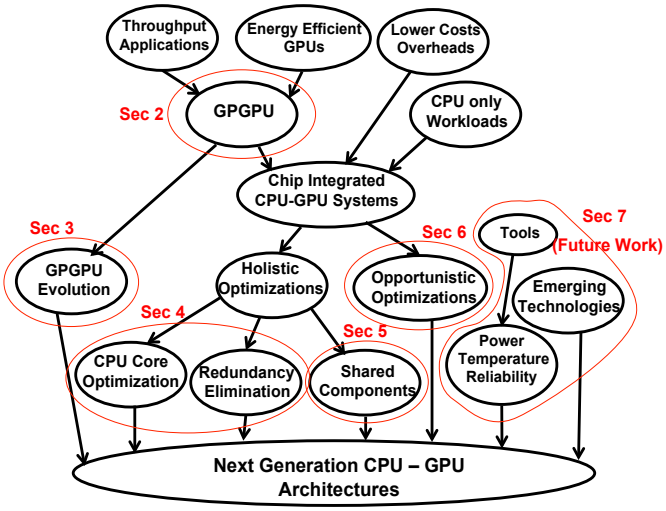

Fig. 1. **Evolution of CPU-GPU architectures.**

Recently AMD (Fusion APUs) [43], Intel (Sandy Bridge) [21] and ARM (MALI) [6] have released solutions that integrate general purpose programmable GPUs together with CPUs on the same die. In this computing model, the CPU and GPU share memory and a common address space. These solutions are programmable using OpenCL [25] or solutions such as DirectCompute [31]. Integrating a CPU and GPU on the same chip has several advantages. First is cost savings because of system integration and the use of shared structures. Second, this promises to improve performance because no explicit data transfers are required between the CPU and GPU [5]. Third, programming becomes simpler because explicit GPU memory management is not required.

Not only does CPU-GPU chip integration offer performance benefits but it also enables new directions in system development. Reduced communication costs and increased bandwidth have the potential to enable new optimizations that were previously not possible. At the same time, there are new problems to consider. Based on a literature survey, we have distilled the major research and development directions for CPU-GPU systems in figure 1.

The top portion of the figure shows the factors that have led to the development of current GPGPU systems. In [30], [46], [2] NVIDIA discusses unified graphics-computing architectures and makes a case for GPU computing. We discuss these papers and examine the architecture of GPGPU systems in section II. Continuous improvement of GPU performance on non-graphics workloads is currently a hot research topic. Current GPUs suffer from two key shortcomings – loss of performance under control flow divergence and poor scheduling

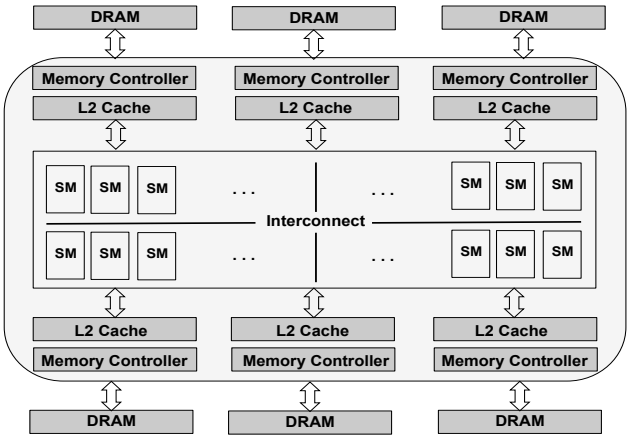

Fig. 2. **Contemporary GPU architecture.**

policies. In [14] and [32], authors explore mechanism for efficient control flow execution on GPUs via dynamic warp formation and a large warp microarchitecture. In [32], the authors propose a better scheduling policy. We discuss these techniques in section III.

One of the key steps in the development of next generation systems might be a range of optimizations. As shown in figure 1, we term the first of these as "holistic optimizations". Under these, the CPU+GPU system is examined as a whole to better optimize its components. Rather than being designed for all workloads, we expect CPU core design to be optimized for workloads that the GPGPU executes poorly. The current combination of CPUs and GPU contains redundant execution components that we expect to be optimized in future designs. In section IV, we discuss these aspects by explaining our work on CPU design directions for CPU-GPU systems [7]. There have been proposals to redesign shared components to account for the different demands of CPU-GPU architectures and workloads. In [28], the authors propose a thread level parallelism aware last-level cache management policy for CPU-GPU systems. In [20], the authors propose memory controller bandwidth allocation policies for CPU-GPU systems. We discuss these papers in section V.

Our research survey provides evidence of a second kind of system optimization that we term as "opportunistic optimizations". Chip integration reduces communication latency and but also opens up new communication paths. For example, previously the CPU and GPU could only communicate over a slow external interface but with chip-integration they share a common last level cache. This enables previously unexplored usage oppourtunities. The ideas discussed revolve around the use of idle CPU or GPU resources. COMPASS [47] proposes using idle GPU resources as programmable data prefetchers for CPU code execution. Correspondingly, in [48], the authors propose using a faster CPU to prefetch data for slower throughput oriented GPU cores. We discuss these collaborative CPU-GPU execution schemes in section VI. We discuss future work directions in section VII and conclude in section VIII.

II. General Purpose GPU Architectures

In this section, we will examine the design of GPU architectures for general purpose computations.

The modern GPU has evolved from a fixed function graphics pipeline which consisted of vertex processors running vertex shader programs and pixel fragment processors running pixel shader programs. Vertex processing consists of operations on point, line and triangle vertex primitives. Pixel fragment processors operate on rasterizer output to fill up the interiors of triangle primitives with interpolated values. Traditionally, workloads consisted of more pixels

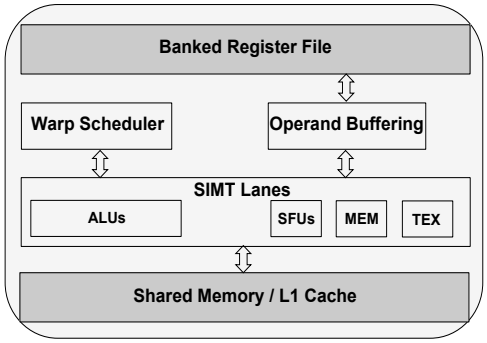

Fig. 3. **Streaming Multiprocessor (SM) architecture.**

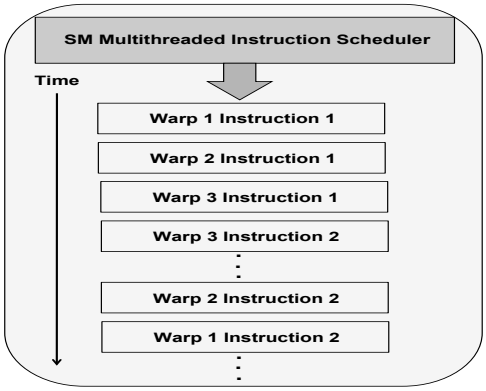

Fig. 4. **Example of warp scheduling.**

than vertices and hence there were greater number of pixel processors. However, unbalance in modern workloads influenced a unified vertex and pixel processor design. Unified processing, first introduced with the NVIDIA Tesla [30], enabled higher resource utilization and allowed the development of a single generalized design.

Figure 2 shows a block diagram of contemporary NVIDIA GPGPUs [30], [46], [2]. The GPU consists of streaming multiprocessors (SMs), 6 high-bandwidth DRAM channels and on-chip L2 cache. The number of SMs and cores per SM varies as per the price and target market of the GPU. Figure 3 shows the structure of an SM. An SM consists of 32 single instruction multiple thread (SIMT) lanes that can collectively issue 1 instruction per cycle per thread for a total of 32 instructions per cycle per SM. Threads are organized into groups of 32 threads called "Warps". Scheduling happens at the granularity of warps and all the threads in a warp execute together using a common program counter. As shown in figure 3, SIMT lanes have access to a fast register file and on-chip low latency scratchpad shared memory / L1 caches. Banking of the register file enables sufficient on-chip bandwidth to supply each thread with two input and 1 output operand each cycle. The operand buffering unit acts as a staging area for computations.

GPUs rely on massive hardware multithreading to keep arithmetic units occupied. They maintain a large pool of active threads organized as warps. For example, NVIDIA Fermi supports 48 active warps for a total of 1536 threads per SM. To accommodate the large set of threads, GPUs provide large on-chip register files. Fermi has a per SM register file size of 128KB or 21 32-bit registers per thread at full occupancy. Each thread uses dedicated registers to enable fast switching. Thread scheduling happens at the granularity of warps. Figure 4 shows an example of warp scheduling. Each cycle, the scheduler selects a warp that is ready to execute and issues the next instruction to that warp's active threads. Warp selection considers

Date	Product	Family	Transistors	Tech Node	GFlops (SP MAD)	GFlops (DP FMA)	Processing Elements	Register File (per SM)	Shared Memory / L1 (per SM)	L2 Size	Memory Bandwidth (GB/s)	Total Threads
2006	GeForce 8800	Tesla	681 million	90nm	518	–	128	8KB	16KB	–	86.4	12,288
2008	GTX 280	Tesla	1.4 billion	65nm	933	90	240	16KB	16KB	–	142	30,720
2009	GF 100	Fermi	3.1 billion	40nm	1028	768	512	32KB	48KB	768KB	142	24,576
2012	GK 110	Kepler	7.1 billion	28nm	2880	960	2880	64KB	64KB	1536KB	192	30,720
2009	Core i7-960	Bloomfield	700 million	45nm	102	51	8 x 4 wide SIMD	–	32KB	8MB L3	32	8
2012	Core i7 Extreme	Sandy Bridge	2.3 billion (wGPU)	28nm	204[2]	102[2]	16 x 4 wide SIMD	–	32KB	20MB L3	37	16

TABLE I
GPU PERFORMANCE SCALING DATA FROM PUBLICATIONS [33], [24], [2], [29] AND OPEN SOURCES [1] HAVE BEEN USED TO GENERATE THIS TABLE.

factors such as instruction type and fairness while making a pick. Instruction processing happens in-order within a warp but warps can be selected out-of-order. This is shown in the bottom part of figure 4.

A SIMT processor is fully efficient when all the lanes are occupied. This happens when all 32 threads of a warp take the same execution path. If threads of a warp diverge due to control flow, the different paths of execution are serially executed. Threads not on the executing path are disabled and on completion all threads re-converge to the original execution path. SMs use a branch synchronization stack to manage thread divergence and convergence.

GPUs are designed to reduce the cost of instruction and data supply. For example, SIMT processing allows GPUs to amortize cost of instruction fetch since a single instruction needs to be fetched for a warp. Similarly, large on-chip register files reduce spills to main memory. Programmers have the additional option of manually improving data locality by using scratchpad style shared memory. There is explicit programmer support to enable this.

GPUs have been designed to scale. This has been achieved with the lack of global structures. For example, unlike CPUs, the SMs have simple in-order pipelines, albeit at a much lower single thread performance. Instead of seeking performance via caches and out-of-order processing over large instruction windows, GPUs incorporate zero overhead warp scheduling and hide large latencies via multithreading. There is a lack of global thread synchronization i.e. only threads within an SM can synchronize together and not across the whole machine. Lastly, there is a lack of global wires to feed data. Instead, a rich on-chip hierarchy of large registers files, shared memory and caches is used to manage locality. Such features reduce power consumption and allow GPUs to scale with lower technology nodes [33], [24].

Table I shows GPU scaling since 2006. We observe that floating point capabilities are scaling at or beyond Moore's law pace. Single precision multiple-add performance (MAD) has increased about 6×. Double precision fused multiply add (FMA), introduced first in 2008, has grown to about a teraflop of performance in the latest architectures. The total size of storage structures is increasing somewhat slowly. Shared memory and register files have increased in size 4× and 8× respectively, as compared to about 22.5× growth in the number of ALUs. Memory bandwidth is increasing at an even slower rate, seeing only about a 2.2× increase.

Memory bandwidth clearly represents a potential scaling bottleneck for GPUs. This is partially compensated by the nature of workloads. Typical GPU workloads tend to have high arithmetic intensity and hence can benefit from scaling in FLOP performance. However, bandwidth limited workloads are not expected to scale as well. As varied general purpose workloads start to get mapped to GPUs, there have been proposals for spatially multitasking [3] bandwidth intensive workloads together with arithmetically intensive workloads on the same GPU.

The last two rows of table I show CPU scaling for throughput oriented applications. Lee et al. [29] compared GTX 280 (row 2) vs a Core i7-960 (row 5) and found the performance gap to be about 2.5×. However, raw numbers comparing the state of the art GPUs (row 4) and CPUs (row 6) point to a different picture today. While CPU raw GFlop performance has doubled, GPU double precision raw performance has gone up almost 10×. This points to an increasing performance gap between GPUs and CPUs for throughput oriented applications.

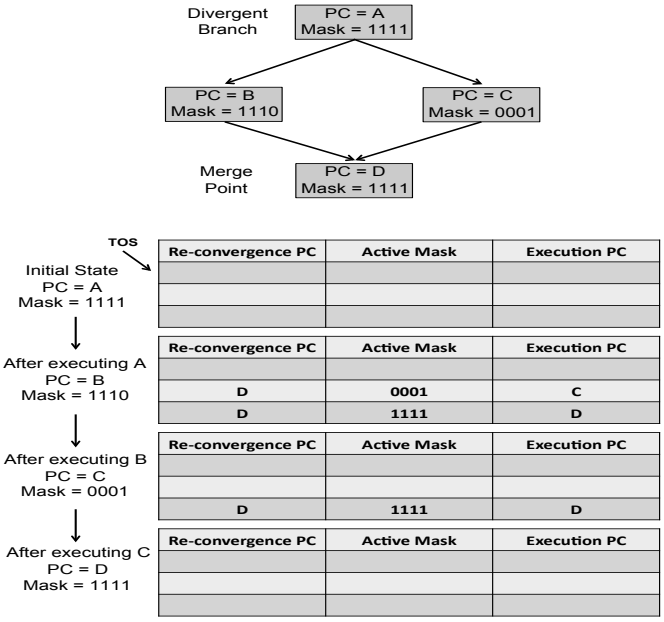

Fig. 5. **Example of stack based re-convergence.**

III. TOWARDS BETTER GPGPU ARCHITECTURES

We anticipate the integration of better general purpose GPGPU designs as one of the next steps in the evolution of CPU-GPU systems. One of the challenges in GPU architectures is efficient handling of control-flow. In this section, we will examine proposals to reduce the performance loss caused by SIMT thread divergence. We also discuss an improved warp scheduling scheme.

SIMT processing works best when all threads executing in the warp have identical control-flow behavior. Pure graphics code tends to not have control flow divergence. But as diverse code gets mapped to the GPU, there is a need to effectively manage performance loss because of divergence. GPUs typically employ the stack based reconveregence stream to split and join divergent thread streams. We will first describe this baseline scheme and then discuss enhancements.

A. Stack based Re-Convergence

Figure 5 illustrates the stack based divergence handling procedure employed by current GPUs. In this example we consider a single warp consisting of 4 threads. The threads execute the code with the control flow shown in the top portion of the figure. At address A,

[2]Estimated from Core i7-960 numbers assuming same frequency of operation.

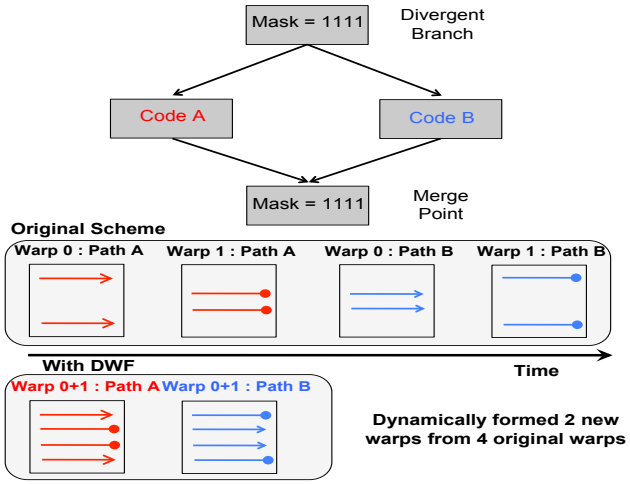

Fig. 6. **Dynamic warp formation example.**

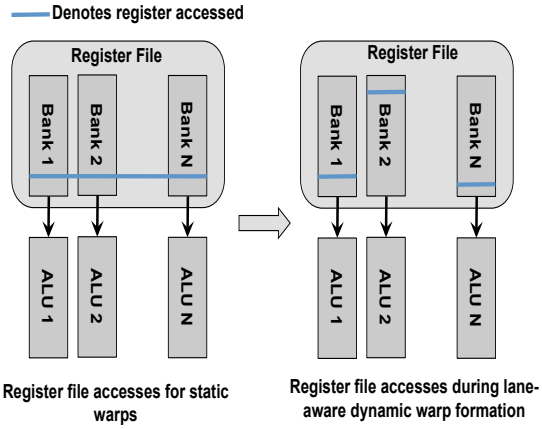

Fig. 7. **Register file accesses during dynamic warp formation.**

there is a conditional branch and 3 threads follow path with address B and the one remaining thread follows path given by address C. The control flow merges at address D. Since a warp can have only a single active PC, on control flow divergence, one of the paths is chosen and the other pushed on to the re-convergence stack and executed later. The re-convergence stack is also used to merge threads in the warp when the threads reach the control flow merge point. Each stack entry consists of three fields: a re-convergence PC, an active mask and an execution PC. Diverging paths execute serially but the re-convergence stack mechanism is used to merge back threads by operating in the following manner:

1) When a warp encounters a divergent branch, an entry with both the re-convergence and execute PCs set to the control flow merge point is pushed on to the stack. The control flow merge points are identified by the compiler. The active mask of the entry is set to the current active mask of the executing branch.

2) One of the divergent paths is selected for execution and the PC and active mask for the warp are set to that of the selected path. Another entry for the yet to execute path is pushed on to the stack. The execute PC and active masks are set according to the yet to execute path. The re-convergence PC is set to the merge point PC. The second stack in figure 5 shows the status of the stack.

3) Each cycle, the warp's next PC is compared to the re-convergence PC at the top of the stack. If the two match, then the reconvergence point has been reached by the current execution path. Then the stack is popped and the current PC and active mask are set to the execution PC and active mask entries of the popped stack entry. This ensures that execution begins for the other divergent path. This is shown in the third stack in figure 5.

The stack re-convergence mechanism guarantees proper execution but not full machine utilization. As shown in figure 5, diverging paths execute serially and only the active threads of a path occupy the machine. The SIMD units corresponding to inactive threads remain un-utilized. In [14], Fung et al., propose "dynamic warp creation" to improve machine utilization during divergence. We will now discuss their scheme.

B. Dynamic Warp Formation

If there was only a single thread warp for execution, then the performance loss due to divergence is unavoidable. Typically GPUs support about 32 – 48 active warps and if there are multiple warps available at the same diverge point, then threads from the same execution path, but of different warps, can be combined to form new warps. Since these new warps follow the same execution path, there is no divergence and better machine utilization. The thread scheduler tries to form new warps from a pool of ready threads by combining threads whose PC values are the same. Figure 6 illustrates the idea. In this example, there are two warps named warp 0 and warp 1. Threads from these warps diverge over paths A and B. However, the scheduler dynamically combines threads from warp 0 and warp 1. Threads following the execution paths A and B are combined into new warps – warp 0+1 path A and warp 0+1 path B. The newly formed warps have no thread divergence. In this way, the pipeline can be better utilized under divergent control flow.

Dynamic warp formation mechanisms can reduce area overheads by accounting for the register-file configuration used in typical GPUs. This variant is called as "Lane-Aware" dynamic warp formation. The need for such a scheme arises because the SIMT lanes in which each thread executes is statically fixed in order to reduce the number of ports in the register file. The registers needed during the execution of a specific lane are allocated to its respective bank. For example, current GPU register files have 32 banks which are sufficient to simultaneously feed 32 SIMT lanes as shown in the left half of figure 7. When forming warps dynamically, the scheduler needs to ensure that all threads in the new warp map to different SIMT lanes and register file banks. Such a scheme removes the need for having a cross bar connection between different ALUs and register file banks. This simplifies design. If the warp formation scheduler can ensure this then the register file accesses would be as shown in the right half of figure 7. This particular scheme removes the need to add expensive ports to the register file. Another possible scheme is to stall the pipeline on a bank conflict and transfer data to the ALU via an interconnection network, but lane-aware dynamic warp formation removes the need for such modifications.

Dynamic warp formation has good potential to fully utilize the SIMT hardware but is dependent on the availability of many warps executing the same PC. If the warps progress at different rates, then there would be not enough warps available to dynamically regroup threads. To tackle this problem, the authors propose warp issue heuristics. A "majority heuristic", which issues warps with the most common PC amongst all ready to schedule warps was found to give good performance.

The authors analyzed overheads required to implement the lane-aware dynamic warp formation scheme with majority heuristics. They found an area overhead of about 4.7% needed for the scheme including that for extra register file multiplexing logic. The storage required to find the majority PC over 32 warps was the highest portion

Benchmark	Suite	Application Domain	GPU Kernels	CPU Time (%)	Kernel Speedup (×)	GPU Mapped Portions	Implementation Source
Kmeans	Rodinia	Data Mining	2	51.4	5.0	Find and update cluster center	Che et al. [9]
H264	Spec2006	Multimedia	2	42.3	12.1	Motion estimation and intra coding	Hwu et al. [19]
SRAD	Rodinia	Image Processing	2	31.2	15.0	Equation solver portions	Che et al. [9]
Sphinx3	Spec2006	Speech Recognition	1	25.6	17.7	Gaussian mixture models	Harish et al. [17]
Particlefilter	Rodinia	Image Processing	2	22.4	32.0	FindIndex computations	Goomrum et al. [15]
Blackscholes	Parsec	Financial Modeling	1	17.7	13.7	BlkSchlsEqEuroNoDiv routine	Kolb et al. [26]
Swim	Spec2000	Water Modeling	3	8.9	25.3	Calc1, calc2 and calc3 kernels	Wang et al. [45]
Milc	Spec2006	Physics	18	8.4	6.0	SU(3) computations across FORALLSITES	Shi et al. [39]
Hmmer	Spec2006	Biology	1	5.7	19.0	Viterbi decoding portions	Walters et al. [44]
LUD	Rodinia	Numerical Analysis	1	4.6	13.5	LU decomposition matrix operations	Che et al. [9]
Streamcluster	Parsec	Physics	1	3.3	26.0	Membership calculation routines	Che et al. [9]
Bwaves	Spec2006	Fluid Dynamics	3	2.9	18.0	Bi-CGstab algorithmn	Ruetsche et al. [37]
Equake	Spec2000	Wave Propagation	2	2.8	5.3	Sparse matrix vector multiplication (smvp)	Own implementation
Libquantum	Spec2006	Physics	4	1.3	28.1	Simulation of quantum gates	Gutierrez et al. [16]
Ammp	Spec2000	Molecular dynamics	1	1.2	6.8	Mm_fv_update_nonbon function	Own implementation
CFD	Rodinia	Fluid Dynamics	5	1.1	5.5	Euler equlation solver	Solano-Quinde et al. [41]
Mgrid	Spec2000	Grid Solver	4	0.6	34.3	Resid, psinv, rprj3 and interp functions	Wang et al. [45]
LBM	Spec2006	Fluid Dynamics	1	0.5	31.0	Stream collision functions	Stratton et al. [22]
Leukocyte	Rodinia	Medical Imaging	3	0.5	70.0	Vector flow computations	Che et al. [9]
ART	Spec2000	Image Processing	3	0.4	6.8	Compute_train_match and values_match functions	Own implementation
Heartwall	Rodinia	Medical Imaging	6	0.4	7.9	Search, convolution etc. in tracking algorithm	Szafaryn et al. [42]
Fluidanimate	Parsec	Fluid Dynamics	6	0.1	3.9	Frame advancement portions	Sinclair et al. [40]

TABLE II

CPU-GPU BENCHMARK DESCRIPTION CPU TIME IS THE PORTION OF APPLICATION TIME ON THE CPU WITH 1× GPU SPEEDUP. KERNEL SPEEDUP IS THE SPEEDUP OF GPU MAPPED KERNELS OVER SINGLE CORE CPU IMPLEMENTATION. ALL NUMBERS ARE NORMALIZED TO THE SAME CPU AND GPU.

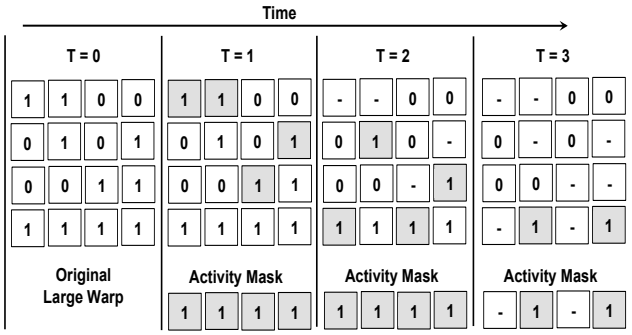

Fig. 8. **Dynamic sub-warp creation.**

of this overhead. The authors demonstrate an average performance benefit of 20.7% for the scheme.

C. Large Warp Microarchitecture and Two-Level Scheduling

Large warp microarchitecture proposed by Narasiman et al. [32] is a similar technique to create warps at runtime. However, it differs in method used to create the warps. The scheme starts out with a warp that is significantly larger in size than the SIMT width. It then dynamically creates SIMT width sized smaller-warps out of the large warp at run-time. While creating the new warps, it groups threads following the same divergence paths. This is illustrated in figure 8. The figure shows a large warp consisting of 16 threads arranged in a two-dimensional structure of 4 smaller warps of 4 threads each. In this example we assume that our SIMT width is 4 threads. As shown in the figure, each cycle, the scheduler creates threads from the original large warp that map to different lanes. Their scheme assumes a similar register file organization and access scheme as used in Fung et al's. [14] dynamic warp formation method.

The paper also proposes an improved scheduling algorithm known as "two-level scheduling". GPUs typically use a round-robin warp scheduling policy giving equal priority to all concurrently executing warps. This is beneficial since there is a lot of data locality across warps. The memory requests of one warp are quite likely to produce row buffer hits for memory requests of other warps. However as a consequence, all warps arrive at a single long latency memory operation at the same time. The key to fixing this problem is to have some warps progress together and arrive at the same long latency operation together, but to have other sets of warps that can be scheduled when all the warps of the first set are waiting. The authors achieve this by performing a two-level warp scheduling. The idea is to group the large set of warps into smaller sets. Individual warps of the sets are scheduled together but on long latency operations the scheduler switches to the different set of warps. The authors evaluated a combined large warp microarchitecture and two-level scheduling overhead scheme and found it to improve performance by 19.1%. Both the schemes combined have an area overhead of 224 bytes.

D. Dynamic Warp Formation vs Large Warp Microarchitecture

Dynamic warp formation gives better performance than the large warp architecture alone. This is because the combination of threads happens from only within the large warp but across all warps in dynamic warp formation. However large warp architecture when combined with two-level scheduling gives better overall performance. Since two-level scheduling is an independent scheme, it can be combined with dynamic warp formation to given even better performance than both the proposed schemes.

IV. HOLISTICALLY OPTIMIZED CPU DESIGNS

In this section, we discuss our work on CPU architecture design directions and optimization opportunities for CPU-GPU systems. The combination of multicore CPUs and GPUs in current systems offers significant optimization oppourtunities.

Although GPUs have emerged as general purpose execution engines, not all code maps well to GPUs. The CPU still runs performance critical code, either as complete applications or portions that cannot be mapped to the GPU. Our work [7] shows that the code running on the CPU in a CPU-GPU integrated system is significantly different than the original code. We think that the properties of this new code should form the basis of new CPU design.

Kumar et al. [27] argue that efficient heterogeneous designs are composed of cores that each run subsets of codes well. The GPGPU is

already a good example of that, it performs quite well on throughput applications but poorly on single threaded code. Similarly, the CPU need not be fully general-purpose. It would be sufficient to optimize it for non-GPGPU code. Our aim in this work is to first understand the nature of such code and then propose CPU architecture directions. We base our conclusions by partitioning important benchmarks on the CPU-GPU system. We begin by describing benchmarks used in the study.

A. Benchmarks

A large number of important CPU applications and kernels with varying levels of GPU offloading have been ported to GPUs. In this work, we relied as much as possible on published GPU implementations. We did this in order to perform code partitioning based on the decisions of the community and not by our abilities. We performed our own CUDA implementations for three important SPEC benchmarks. For all other applications, we use 2 mechanisms to identify the partitioning of the application between the CPU and GPU. First, we base it on the implementation code if available. If the code is not available, we use the partitioning information as stated in publications. Table II summarizes the characteristics of our benchmarks. The table lists out the GPU mapped portion and provides statistics such as time spent on the CPU and normalized reported speedups. We will also collect statistics for benchmarks with no publicly known GPU implementations. Together with the benchmarks listed in the table, we have a total of 11 CPU-heavy benchmarks, 11 mixed and 11 GPU-heavy benchmarks.

B. Methodology

Our goal is to identify fundamental characteristics of the code, rather than the effects of particular architectures. This means, when possible, we characterize as types, rather than measuring hit or miss rates. We do not account for code to manage data movement as this code is highly architecture specific and expected to be absent in chip integrated CPU-GPU systems [5]. We use a combination of real machine measurements and PIN [36] based measurements. Using the CPU/GPU partitioning information we modify the original benchmark code. We insert markers indicating the start and end of GPU code. This allows microarchitectural simulators built on top of PIN to selectively measure CPU and GPU code characteristics. We also insert measurement functions. This also allows us to perform timing measurements. All benchmarks are simulated for the largest available input sizes. Programs were run to completion or for at least 1 trillion instructions.

CPU Time is calculated by using measurement functions at the beginning and end of GPU portions and for the complete program. Post-GPU CPU time was calculated by dividing the GPU portion of the time with the reported speedup. Time with conservative speedups was obtained by capping the maximum possible GPU speedup value to 10.0 (single-core speedup cap from [29]).

Based on measurements of address streams, we categorize loads and stores into four categories – static, strided, patterned and hard. Static loads and stores have their addresses as constants. Loads and stores that can be predicted with 95% accuracy by a stride predictor with up to 16 strides per PC are categorized as strided. Patterned loads and stores are those that can be predicted with 95% accuracy by a large Markov predictor with 8192 entries, 256 previous addresses, and 8 next addresses. All remaining loads and stores are categorized as hard. We categorize branches similarly as – biased (95% taken or not taken), patterned (95% prediction accuracy using a large local predictor, using 14 bits of branch history), correlated (95% prediction accuracy by a large gshare predictor, using 17 bits of global history), and hard (all other branches).

We use the Microarchitecture Independent Workload Characterization (MICA) [18] to obtain instruction level parallelism information. MICA calculates perfect ILP by assuming perfect branch prediction and caches. We modified MICA to support instruction windows up to 512 entries. We define thread-level parallelism (TLP) as the speedup we get on an AMD Shanghai quad core $\times$ 8 socket machine. We used parallel implementations available for Rodinia, Parsec, and some SPEC2000 (those in SPEC OMP 2001) benchmarks for the TLP study. The TLP results cover a subset of all our applications. We could only perform measurements for applications where we have parallel source code available (24 out of 33 total benchmarks).

C. Results

In this section we examine the characteristics of code executed by the CPU, both without and with GPU integration. For all of our presented results, we group applications into three groups — CPU-heavy, mixed and GPU-heavy. We start by look at CPU time – the portion of the original execution time that gets mapped to the CPU.

CPU Execution Time To identify the criticality of the CPU after GPU offloading, we calculate the percentage of time in CPU execution after the GPU mapping. The first bar in Figure 9 is the percentage of the original code that gets mapped to the CPU. The other two bars represent the fraction of the total time spent on the CPU. The second and third bars account for GPU speedups with the third bar assuming that the GPU speedup is capped at 10$\times$. While the 11 CPU-heavy benchmarks completely execute on the CPU, for the mixed and GPU-heavy set of benchmarks about 80% and 7-14% of execution is mapped to the CPU respectively. On average, program execution spends more time on the CPU than the GPU. We see that the CPU remains performance critical. In the figure we have sorted the benchmarks by CPU time. We will use the same ordering for subsequent graphs. We weight post-GPU average numbers by the conservative cpu time (third bar) in all future graphs.

ILP is a measure of instruction stream parallelism. It is the number of average independent instructions within the window size. We measured ILP for two window sizes – 128 entries and 512 entries. As seen in figure 10, in 17 of the 22 applications, ILP drops noticeably, particularly for large window sizes. For benchmarks such as swim, milc and cfd, it drops by almost 50%. For the mixed set of benchmarks, the ILP drops by over 27% for large window sizes. In the common case, independent loops with high ILP get mapped to the GPU, leaving dependence-heavy code to run on the CPU. Occasionally dependent chains of instructions get mapped to the GPU. For example, the kernel loop in blackscholes consisted of long chains of dependent instructions. Overall, we see a 4% drop in ILP for current generation window sizes and a 11% drop for larger sizes. The gains from large windows sizes are degraded for the new CPU code.

Branches Figure 11 plots the distribution of branches based on our previously defined classification. We see a significant increase in hard branches. The frequency of hard branches increases by 65% (from 11.3% to 18.6%). Much of this is the reduction in patterned branches, as the biased branches are only reduced by a small amount. The overall increase in hard branches is because of the increase in hard branches for mixed benchmarks and a high number of hard branches in the CPU-heavy workloads. Hard branches increase primarily because loops with easily predictable backward looping branches get mapped to the GPU. This leaves irregular code to run on the CPU, increasing the percentage of hard branches. Occasionally, data-dependent branches are mapped to the GPU such as in equake and cfd benchmarks. Since data-dependent branches are more difficult to predict, the final CPU numbers appear as outliers. We simulated a

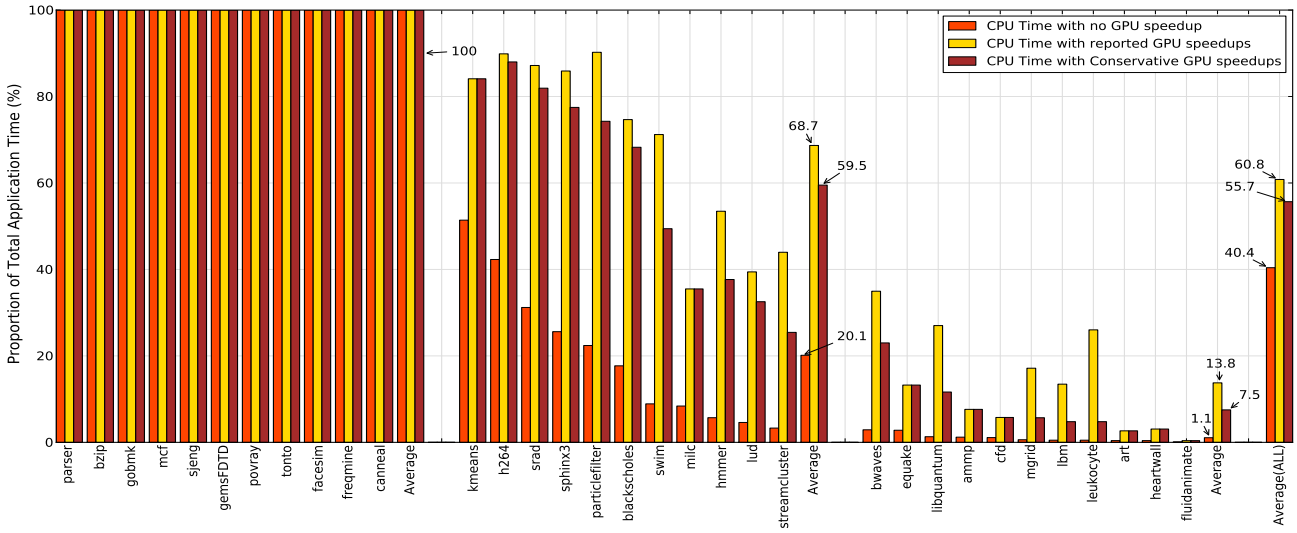

Fig. 9. **Time spent on the CPU**.

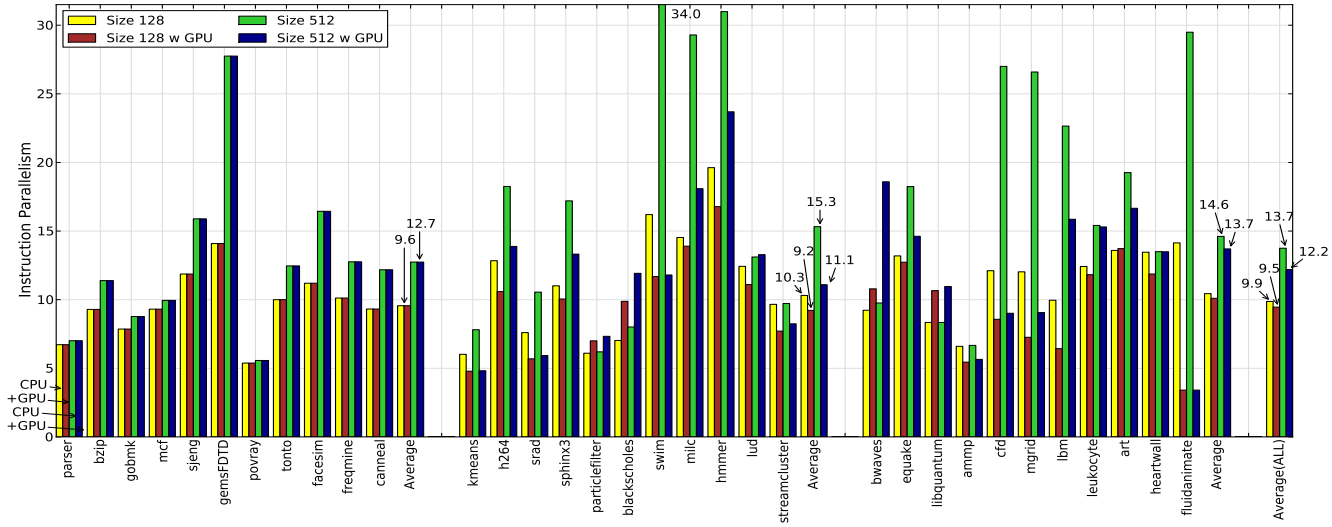

Fig. 10. **Instruction level parallelism with and without GPU**.

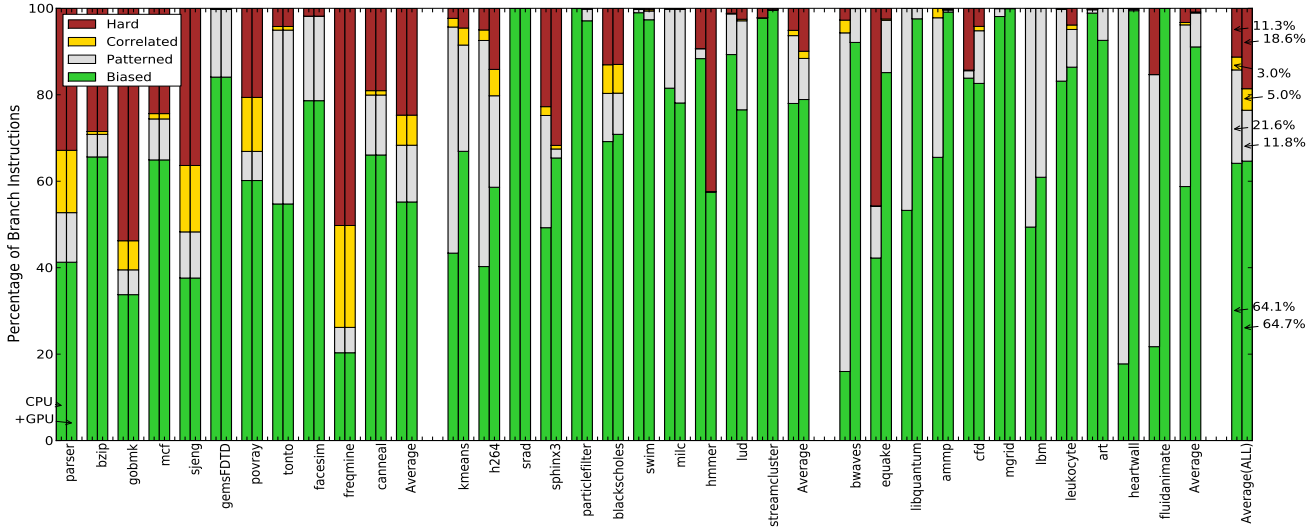

Fig. 11. **Distribution of branch types**.

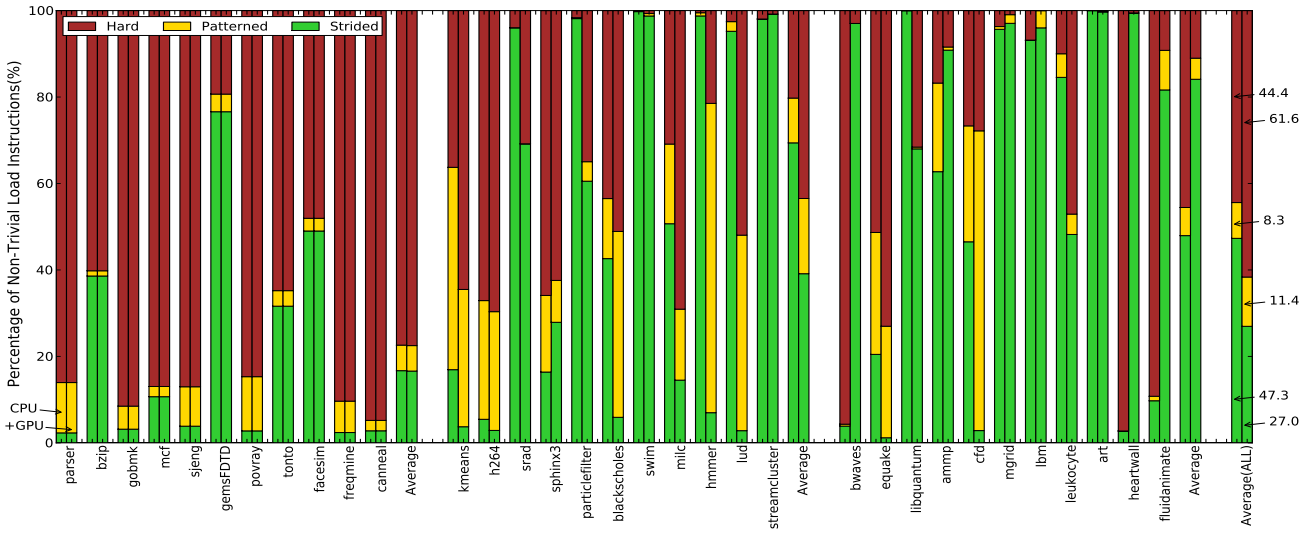

Fig. 12. **Distribution of loads types**.

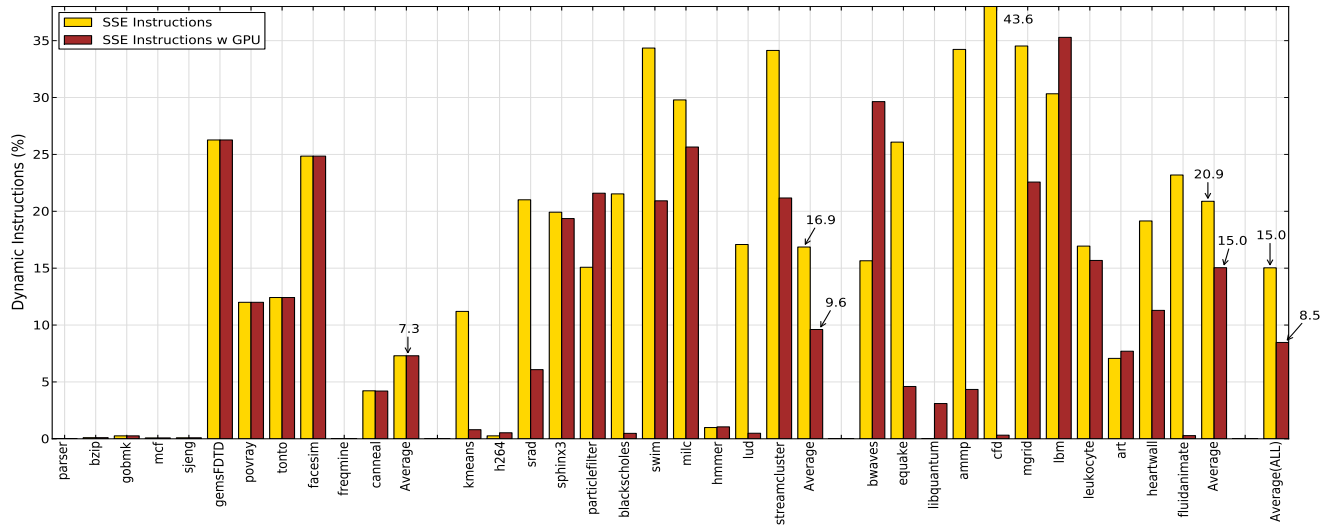

Fig. 13. **Frequency of vector instructions with and without GPU**.

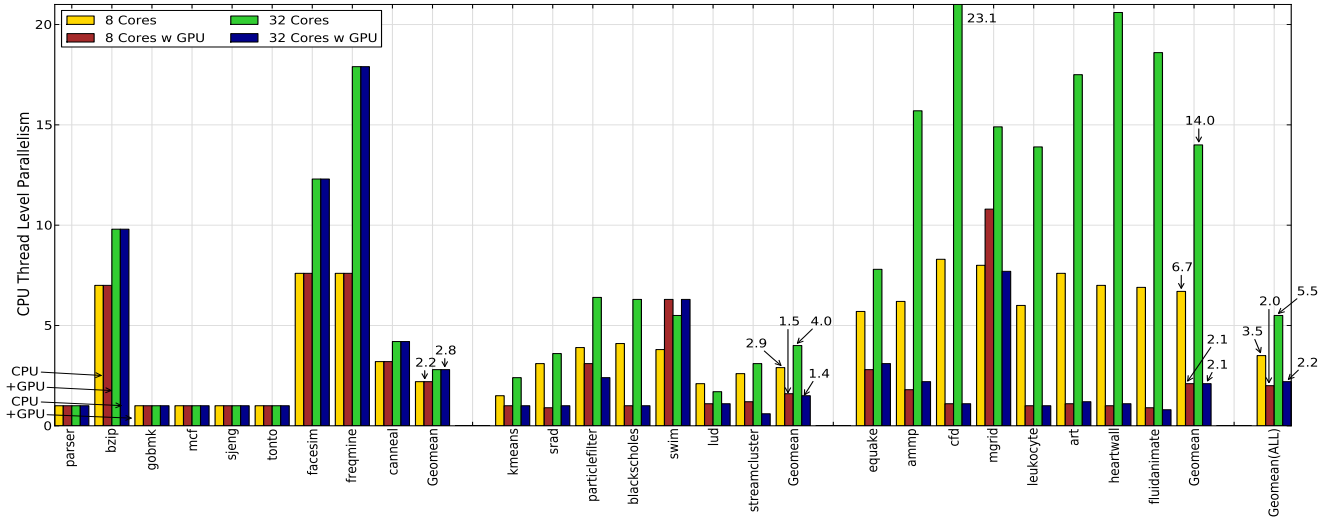

Fig. 14. **Thread level parallelism with and without GPU**.

realistic branch predictor, in order to evaluate the performance impact of hard branches on real branch prediction rates. We found our miss prediction rate to increase by 56%. We have omitted the graph in order to conserve space.

Loads Figure 12 shows the classifications of CPU loads. We show the breakdown of loads as a percentage of non-static loads i.e. loads that are not trivially cached. We see that there is a sharp decrease in strided loads and a corresponding increase in hard loads. In the common case, regularly ordered code maps well to the GPU. We observe that in our results. The hard loads remaining on the GPU are not easily handled by existing hardware prefetchers or inline software prefetching. The percentage of strided loads is almost halved, both overall and for the mixed workloads. Patterned loads are largely unaffected, but hard loads increase and become the most common type. Applications such as lud and hmmer see an almost complete change in behavior from strided to hard. We see an exception in bwaves which goes from being almost completely hard to strided. This is because the kernel with highly irregular loads is successfully mapped to the GPU. To conserve space, we do not show results for stores in this paper. We found stores to exhibit similar results as loads.

Vector Instructions We find the usage of SSE instructions to drop significantly as shown in figure 13. We saw an overall reduction of 44.3% in the usage of SSE instructions (from 15.0% of all dynamic instructions to 8.5%). This shows that SSE ISA enhancements target the same code regions as the GPGPUs. For example, in kmeans we found the find_nearest_point functions to heavily utilize SSE instructions. This function was part of the GPU region.

Thread Level Parallelism TLP captures parallelism that can be exploited by multiple cores or thread contexts. This allows us to measure the application level utility of having an increasing number of CPU cores. Figure 14 shows TLP results. Let us first consider the GPU-heavy benchmarks. CPU-only implementations of the benchmarks show abundant TLP. We see an average speedup of 14.0× for 32 cores. However, post-GPU the TLP drops considerably, yielding only a speedup of 2.1×. Five of the benchmarks exhibit no TLP post-GPU, in contrast, five benchmarks originally had speedups greater than 15×. Perhaps the most striking result is that no benchmark's post-GPU code sees any significant gain from going from 8 cores to 32. Overall for the mixed benchmarks, we again see a considerable reduction in post-GPU TLP; it drops by almost 50% for 8 cores and about 65% for 32 cores. We see that applications with abundant TLP are good GPU targets. In essence, both multicore CPUs and GPUs are targeting the same parallelism. However, as we have seen, post-GPU parallelism drops significantly. On average, we see a striking reduction in exploitable TLP. 8 core TLP dropped by 43% from 3.5 to 2.0 and 32 core TLP dropped by 60% from 5.5 to 2.2. While going from 8 cores to 32 cores yielded a nearly two fold increases in TLP, post-GPU the TLP grows by just 10% over that region. Post-GPU, extra cores provide almost no benefit.

D. Impact on CPU Design

We group the architectural implications of the changing CPU code base into two sets – CPU core optimizations and redundancy eliminations.

CPU Core Optimizations Since out-of-order execution benefits from large instruction windows, we have seen a steady increase in processor window sizes for commercial designs and research that increases window sizes or creates such an illusion [13]. We do not see evidence that large windows are not useful. However the gains from increasing window sizes might be muted. We see that post-GPU, pressure increases on the branch predictor. Recently proposed techniques [38] that use complex hardware with very long histories might be more applicable because they better attack harder branches. Memory accesses will continue to be a performance bottleneck for future processors. The commonly used stride-based or next-line prefetchers are likely to become significantly less relevant. We recommend using significant resources towards accurate prediction of loads and stores. Several past approaches that can capture complex patterns including Markov-based predictors [23], predictors targeted at pointer-chain computation [12], [10] and helper-thread prefetching [8], [49], [11] should be pursued with new urgency.

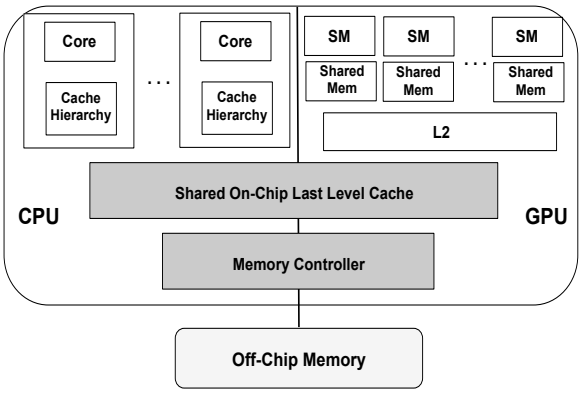

Fig. 15. **Chip integrated CPU-GPU architecture.**

Redundancy Eliminations With the addition of a GPU, SSE instructions have been rendered less important. Much of code that gets mapped to CPU vector units can be executed faster and with lower energy on the GPU. While there is no empirical evidence to completely eliminate SSE instructions, some cores might choose to not support SSE instructions or share SSE hardware with other cores. Recent trends show that both CPU and GPU designs are headed in the same direction with ever increasing core and thread counts. Our data suggests that the CPU should refocus on addressing highly irregular code with low degrees of parallelism.

V. Holistic Optimization of Shared Structures

In this section, we will discuss the design of two important shared components. Figure 15 shows a block diagram of an integrated CPU-GPU system. As we can see from the figure, last level caches and the memory controller are shared amongst the CPU and GPU. The integrated system brings new challenges for these components because of CPU and GPU architectural differences.

While CPUs depend on caches to hide long memory latencies, GPUs employ multithreading together with caching to hide latencies. TAP [28] utilizes this architectural difference to allocate cache capacity for GPU workloads in CPU-GPU systems. GPUs trade-off memory system latency to bandwidth by having a lot of outstanding requests to the memory system. Memory intensive CPU workloads can potentially cause GPU delays and lead to missed real-time deadlines on graphics workloads. In [20], Jeong et al. discuss a CPU-GPU memory bandwidth partitioning scheme to overcome this problem. We start by describing the TAP [28] system.

A. Shared Last-level Cache Design

In TAP [28], the authors utilize two key insights to guide decisions on GPU workload cache capacity allocation. First, since GPUs are massively multi-threaded, caching is effective only when the benefits of multi-threading are limited. While for CPU workloads cache hit rates can directly translate to performance, this is not always the case for GPUs because of multi-threading based latency hiding. They

find that traditional cache management policies are tuned to allocate capacity on higher hit-rates. This might not hold for GPUs where hit or miss rates do not always direct translate to performance increase or loss. To solve this problem they propose a "core sampling controller" to measure actual performance differences of cache policies.

Second, GPUs and CPUs have different access rates. GPUs have orders of magnitude more threads and generate more caches accesses than CPU workloads. The authors propose a "cache block lifetime normalization" method to enforce similar cache lifetimes for both CPU and GPGPU applications, even under the GPGPU workload producing excessive accesses. Using the core sampling controller and cache block lifetime normalization blocks, the authors propose TLP aware cache partitioning schemes. We will now describe the design on core sample controller and cache block lifetime normalization blocks. Next we will explain how the authors modified the utility based cache partitioning (UCP) scheme to propose TAP-UCP.

Core sampling controller measures the cache sensitivity of workloads. It does so by using two completely different cache policies on cores (e.g. first core LRU insertion and second core MRU insertion) and checking if core performance is different. A performance different indicates cache sensitivity for the particular GPU workload.

Cache block lifetime normalization first measures number of cache accesses for each workload. Next the ratio of caches access counts are calculated for all workloads. TAP uses these ratios to enforce similar cache residual times for CPU and GPU applications.

TAP-UCP algorithm proposed by the authors is a modification of the well known UCP scheme. UCP is a dynamic cache partitioning scheme that divides cache ways amongst applications at runtime. UCP uses a hardware mechanism to calculate the utility of allocating ways to particular applications. The goal is to maximize hit-rate and hence cache ways are periodically allocated to applications with higher marginal utility (utility per unit cache resources). In UCP, hit-rate is assumed to lead to better performance. The authors modify UCP to allocate just a single way for GPGPU applications with little benefit. They scheme allocates less number of ways for cache insensitive GPGPU applications. The performance sensitivity measurement is achieved using the core sampling controller. They also modify the UCP scheme such that GPGPU applications hit-rates and utilities are normalized by the ratios of workload access counts calculated by the cache block lifetime normalization block.

On similar lines, the authors propose modifications to the re-reference interval prediction algorithm and propose the TAP-RRIP algorithm. We omit the details of the scheme to conserve space. The authors evaluated the TAP-UCP scheme over 152 heterogeneous workloads and found it to improve performance by 5% over UCP and 11% over LRU.

B. Memory Controller Design

Jeong et al. [20] propose dynamic partitioning of off-chip memory bandwidth between the CPU and GPU to maintain a high quality of service for the overall system. Typical memory controllers prioritize CPU requests over GPU requests as the CPU is latency sensitive and the GPU is designed to tolerate long latencies. However, such a static memory controller policy can lead to an unacceptably low frame rate for the GPU. Correspondingly, prioritizing GPU requests can degrade the performance of the CPU. The authors scheme is targeted towards system-on-chip architectures with multicore CPUs and graphics only GPUs. Nevertheless the technique is quite relevant in the context of systems consisting of general purposes GPUs.

Figure 16 (from [20]) shows the impact of prioritizing CPU workloads over GPU. In the top part of the figure we see that with the dual core mcf-art workload the GPU is barely able to maintain

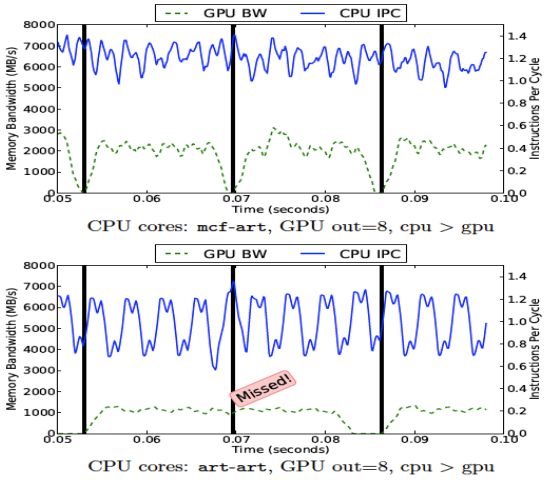

Fig. 16. **GPU bandwidth consumption and CPU performance. GPU has up-to 8 outstanding requests and CPU requests have higher priority. Vertical lines represent frame deadlines (from [20]).**

deadlines. However for the bandwidth-intensive CPU workload art-art, and with the same policy of prioritizing CPU workloads, GPU deadlines are missed.

Since static management policies cause problems for bandwidth intensive CPU workloads, the authors propose a dynamic quality of service maintenance scheme. In this scheme, the memory controller first evaluates the current rate of progress on the GPU frame. Since the frame is decomposed into smaller tiles, progress can be measured by counting the number tiles completed versus the total number of tiles. This current frame rate is then compared with the target frame rate. They use a default policy to prioritize the CPU requests over the GPU. However, if the current frame progress is slower than the target frame rate the CPU and GPU priorities are set to equal. This provides some opportunity for the GPU to catch up as its priority increases from lower than CPU to same as CPU. However, if the GPU is still lagging behind, when close to the frame deadline, the GPU priority is boosted over the CPU.

The authors evaluated the proposed scheme over a variety of CPU and GPU workloads. They found the proposed mechanism to significantly improved GPU frame rates with minimal impact on CPU performance.

VI. Opportunistic Optimizations via Collaborative Execution

In this section, we will discuss opportunistic optimization schemes for CPU-GPU systems. The CPU-GPU combination is shaping towards a system where the GPU is expected to run throughput oriented portions of code and CPU runs the non-parallel regions of code. However, the GPU, while occupying significant area budgets does not contribute towards the performance of serial applications. Similarly, the CPU is idle while running parallel GPU applications. Woo et al.'s COMPASS [47] proposes the use of GPU resources to boost CPU performance. We discuss their scheme first. Next we will discuss Yang et al. [48] scheme to use CPU resources to boost GPGPU performance.

A. Idle GPU Shader based Prefetching

COMPASS [47] proposes the use of idle gpu resources to act as data prefetchers for CPU execution. The authors suggest using GPU resources in two specific ways. First, they propose the use of large GPU register files as prefetcher storage structures. The 32KB

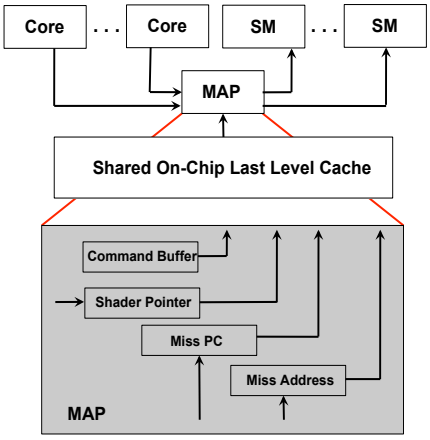

Fig. 17. **Miss Address Provider.**

```
__global__ void VecAdd (float *A, *B, *C, int N) {
   int I = blockDim.x * blockIdx.x + threadIdx.x;
   C[i] = A[i] + B[i] }

float mem_fetch (float *A, *B, *C, int N) {
   return A[N] + B[N] + C[N] }

void cpu_prefetching (...) {
    unroll_factor = 8
   //traverse through all thread blocks (TB)
   for (j = 0;  j < N_TB; j += Concurrent_TB)
   //loop to traverse concurrent threads TB_Size
      for (i = 0; i < Concurrent_TB*TB_Size;
      i += skip_factor*batch_size*unroll_factor) {
        for (k=0; j<batch_size; k++) {
           id = i + skip_factor*k*unroll_factor
              + j*TB_Size
           //unrolled loop
           float a0 = mem_fetch (id + skip_factor*0)
           float a1 = mem_fetch (id + skip_factor*1)
           . . .
           sum += a0 + a1 + . . . }
           update skip_factor
}}}
```

Fig. 18. **GPU kernel and the generated pre-execution program.**

– 64KB of register file space per SM provides sufficient storage for the implementation of state of the art prefetching algorithms. These schemes have prohibitive costs which makes their inclusion into commercial designs difficult. Using GPU resources drastically reduces overhead. Second, the authors propose the use of programmable GPU execution threads as logic structures to flexibly implement prefetching algorithms.

Instead of a completely hardware based scheme, the authors propose the use of an OS based interface to control the GPU based prefetcher operation. The authors describe a Miss Address Provider (MAP) hardware block to provide an interface between the GPU, shared last-level cache (LLC) and the OS. Figure 17 illustrates MAP. Once the OS has no pending GPU job, it assigns a prefetching shader via the shader pointer. Upon an LLC miss or prefetched line hit, the PC and miss address are forwarded to MAP, which first sends a GPU command to assign a GPU shader and or thread to generate prefetch requests for the particular program address. If a GPU shader has already been allocated, the shader stores the miss information in the GPU register files and executes prefetching algorithms to bring future data into the LLC. The OS disables COMPASS shaders before context switching and then re-enables after the context switch. Since COMPASS is programmable, the OS can select prefetching algorithms from a collection of different such implementations.

In the paper, the authors demonstrate different COMPASS based prefetching algorithms such as strided prefetching, markov prefetching and application custom predictors. One of the problems of the GPU is poor single thread performance. This could increase the latency of processing the miss information to generate timely prefetch requests. The authors address this by demonstrating multithreaded GPU prefetchers that reduce prefetch calculation latency. Overall the authors report low area overheads since most of the GPU hardware is used as it is. They demonstrate a average performance benefit of 68% with their scheme.

B. CPU Assisted GPGPU Processing

Yang et al. [48] propose the use of CPU based execution to prefetch requests for GPGPU programs. First, they develop a compiler based infrastructure to extract memory address generation and accesses from GPU kernels to create a CPU pre-execution program. Once the GPU kernel is launched, the CPU runs the pre-execution program. To make the pre-execution effective, the CPU needs to run sufficiently ahead so as to bring relevant data into the shared LLC. However, the execution should not run too far ahead that the prefetched data are replaced before being utilized. The authors propose schemes to manage prefetch effectiveness. The authors argue that while CPUs have considerably less throughput than GPUs, very few CPU instructions are required to perform address generation and prefetching. This is primarily because each prefetching request can bring in a single LLC block, which is considerably large in size and serves multiple GPU threads together.

Figure 18 shows an example of a vector add GPU kernel and the compiler generated pre-execution program. As shown, the pre-execution generation algorithm first extracts memory accesses with address generation. All stores are converted into loads. Next, loops are added to prefetch data for concurrent threads organized into separate thread blocks. The iterator update is set as a product of three factors. The first, skip_factor is used to adjust the timeliness of CPU prefetching by skipping threads. The authors propose an adaptive scheme to vary skip_factor by tracking the LLC hit rate. A too high hit rate value means that the data is already in the cache because of GPU execution. A too low hit rate might indicate that the CPU is running too far ahead. The batch_size parameter is used to control how often the skip_factor parameter is updated. The unroll_factor parameter is used to boost CPU requests under CPU-GPU memory contention.

The authors proposed scheme has two drawbacks. First they assume that blocks are scheduled linearly i.e. the block with id 0 is scheduled first, then with id 1 and so on. However, block scheduling policies could differ and in that case GPU would need to communicate the executing block id information to the CPU. This communication could impact the timeliness of the prefetcher. Secondly, since the CPU pre-execution program is stripped of actual computation, any data or computation dependent memory accesses cannot be handled by this approach. Most of the benchmarks used in the study did not have data dependent memory accesses. This is a drawback of the scheme under increasing code diversity. The authors demonstrate a 21% performance benefit of their proposal.

VII. Future Work Directions

In this section we will discuss oppourtunities for future work in the area of CPU-GPU systems. We characterize these oppourtunities into 4 categories – continued system optimization, research tool development, oppourtunities in power, temperature and reliability and lastly the use of emerging technologies in CPU-GPU systems.

Continued System Optimizations We see both holistic and opportunistic optimizations to continue on CPU-GPU systems. The shared LLC and memory controller works presented in this report are the first papers on the area, providing abundant scope to improve performance further. For example, the LLC paper suggests TLP aware

cache management based on effective utilization of cache capacity. It would be interesting to consider bandwidth effects for shared cache management policies. For the memory controller, it would be interesting to explore the effects of GPU bandwidth usage on CPU workloads. Previously, several techniques have been proposed that use idle CPU cores to boost performance of CPU execution threads. Perhaps techniques such as these could be applied to gpgpu systems, where we could use idle gpu resources to boost the performance of gpu execution.

Research Tools One of the major factors that is limiting research in the area is the lack of research tools. While GPGPU performance models are available, there are no GPGPU power models. There is some work in the area with the use of empirical measurement and modeling but the academic community desires flexible analytical GPU power models. Once developed, further work needs to be done to integrate such GPGPU models with CPU power models. Similarly, there are no tools available to model GPU temperature. The development of such tools represents short term research oppourtunities.

Power, Temperature and Reliability Although GPUs are severely power and energy constrained, there is almost no work in the area of effective power and temperature management for GPUs. Similarly, there is no work in the area of GPU reliability. Lack of work in these areas can perhaps be attributed to the lack of tools. We expect this to change as tools become available. As a first order work, it will be interesting to study the application of CPU power and temperature management techniques to GPU systems.

Emerging Technologies There has been almost no work in the application of emerging technologies such as non-volatile memory (NVM) technologies and 3D stacking to GPUs. Low leakage and low power NVMs offer performance benefits to power constrained GPUs. The key would be to find structures with low write activity to mitigate some of NVM disadvantages. Similarly, 3D stacking has the potential to provide much needed memory system bandwidth to GPU systems. Hence, it would be interesting to investigate stacked CPU-GPU-Main memory systems. They key would be the effective management of temperature effects.

VIII. Conclusions

In this work we investigate the architecture and evolution of general purpose CPU-GPU systems. We started by describing state of the art in GPGPU designs. We considered solutions to key GPGPU problems – performance loss due to control-flow divergence and poor scheduling. As a first step, chip integration offers better performance. However, reduced latencies and increased bandwidth are enabling optimizations previously not possible. We described holistic CPU-GPU system optimization techniques such as CPU core optimizations, redundancy elimination and the optimized design of shared components. We studied opportunistic optimizations of the CPU-GPU system via collaborative execution. Lastly, we suggested future work oppourtunities for CPU-GPU systems.

References

[1] NVIDIA GPU and Intel CPU family comparision articles. http://www.wikipedia.org.
[2] NVIDIA's next generation cuda compute architecture: Kepler GK110. Technical report, 2012.
[3] J. T. Adriaens et al. The case for gpgpu spatial multitasking. *High Performance Computer Architecture*, 2012.
[4] AMD Close to the Metal (CTM). http://www.amd.com/.
[5] AMD OpenCL Programming Guide. http://developer.amd.com.
[6] ARM Mali-400 MP. http://www.arm.com.
[7] M. Arora et al. Redefining the role of the cpu in the era of cpu-gpu integration. *In submission to IEEE Micro*, 2012.
[8] R. Chappel et al. Simultaneous subordinate microthreading (SSMT). In *International Symposium on Computer Architecture*, 1999.
[9] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization*, 2009.
[10] J. Collins et al. Pointer cache assisted prefetching. In *International Symposium on Microarchitecture*, 2002.
[11] J. D. Collins et al. Speculative precomputation: Long-range prefetching of delinquent loads. In *International Symposium on Computer Architecture*, 2001.
[12] R. Cooksey et al. A stateless, content-directed data prefetching mechanism. In *Architectural Support for Programming Languages and Operating Systems*, 2002.
[13] A. Cristal et al. Toward kilo-instruction processors. *ACM TACO*, 2004.
[14] W. Fung et al. Dynamic warp formation and scheduling for efficient gpu control flow. In *International Symposium on Microarchitecture*, 2007.
[15] M. A. Goodrum et al. Parallelization of particle filter algorithms. In *Emerging Applications and Many-Core Architectures*, 2010.
[16] E. Gutierrez et al. Simulation of quantum gates on a novel GPU architecture. In *International Conference on Systems Theory and Scientific Computation*, 2007.
[17] S. C. Harish et al. Scope for performance enhancement of CMU Sphinx by parallelising with OpenCL. *Wisdom Based Computing*, 2011.
[18] K. Hoste et al. Microarchitecture-independent workload characterization. *IEEE Micro*, 2007.
[19] W. M. Hwu et al. Performance insights on executing non-graphics applications on CUDA on the NVIDIA GeForce 8800 GTX. *Hotchips*, 2007.
[20] M. K. Jeong et al. A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *Design Automation Conference*, 2012.
[21] H. Jiang. Intel next generation microarchitecture code named Sandy-Bridge. Intel Developer Forum, 2010.
[22] John Stratton. https://csu-fall2008-multicore-gpu-class.googlegroups.com/web/LBM_CO-State_151008.pdf.
[23] D. Joseph et al. Prefetching using markov predictors. In *International Symposium on Computer Architecture*, 1997.
[24] S. W. Keckler et al. GPUs and the future of parallel computing. *IEEE Micro*, 2011.
[25] Khronos Group. OpenCL - the open standard for parallel programming on heterogenous systems . http://www.khronos.org/opencl/.
[26] C. Kolb et al. Options pricing on the GPU. In *GPU Gems 2*.
[27] R. Kumar et al. Core architecture optimization for heterogeneous chip multiprocessors. In *Parallel architectures and compilation techniques*, 2006.
[28] J. Lee et al. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architectures. In *High Performance Computer Architecture*, 2012.
[29] V. W. Lee et al. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *International Symposium on Computer Architecture*, 2010.
[30] E. Lindholm et al. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 2008.
[31] Microsoft Corporation DirectCompute architecture. http://en.wikipedia.org/wiki/DirectCompute.
[32] V. Narasiman et al. Improving GPU performance via large warps and two-level warp scheduling. In *International Symposium on Microarchitecture*, 2011.
[33] J. Nickolls et al. The GPU computing era. *IEEE Micro*, 2010.
[34] NVIDIA Corporation. CUDA Toolkit 4.0. http://developer.nvidia.com/category/zone/cuda-zone.
[35] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics*, 2005.
[36] V. J. Reddi et al. Pin: A Binary Instrumentation Tool for Computer Architecture Research and Education. *Workshop on Computer Architecture Education*, 2004.
[37] G. Ruetsch et al. A CUDA fortran implementation of bwaves. http://www.pgroup.com/lit/articles/.
[38] A. Seznec. The L-TAGE Branch Predictor. In *Journal of Instruction-Level Parallelism*, 2007.
[39] G. Shi et al. Milc on GPUs. NCSA technical report, 2010.
[40] M. Sinclair et al. Porting CMP benchmarks to GPUs. Technical report, CS Department UW Madison, 2011.
[41] L. Solano-Quinde et al. Unstructured grid applications on GPU: performance analysis and improvement. In *GPGPU*, 2011.
[42] L. G. Szafaryn et al. Experiences accelerating matlab systems biology applications.
[43] The AMD Fusion Family of APUs. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.
[44] J. P. Walters et al. Evaluating the use of GPUs in liver image segmentation and HMMER database searches. In *International Symposium on Parallel and Distributed Processing*, 2009.
[45] G. Wang et al. Program optimization of array-intensive SPEC2k benchmarks on multithreaded GPU using CUDA and Brook+. In *Parallel and Distributed Systems*, 2009.
[46] C. M. Wittenbrink et al. Fermi GF100 gpu architecture. *IEEE Micro*, 2011.
[47] D. H. Woo et al. COMPASS: a programmable data prefetcher using idle GPU shaders. In *Architectural Support for Programming Languages and Operating Systems*, 2010.
[48] Y. Yang et al. CPU-assisted GPGPU on fused CPU-GPU architectures. In *High Performance Computer Architecture*, 2012.
[49] C. Zilles et al. Execution-based prediction using speculative slices. In *International Symposium on Computer Architecture*, 2001.