Redefining the Role of the CPU in the Era of CPU-GPU Integration

Manish Arora, Siddhartha Nath, Subhra Mazumdar,
Scott Baden, and Dean Tullsen

Department of Computer Science & Engineering
University of California, San Diego

Abstract

GPU computing has emerged as a viable alternative to CPUs for throughput oriented applications or regions of code. Speedups of 10× to 100× over CPU implementations have been reported. This trend is expected to continue in the future with GPU architectural advances, improved programming support, scaling, and tighter CPU-GPU chip integration.

However, not all code will get mapped to the GPUs, even for many of those applications which map well to the GPU – the CPU still runs code that is not targeted to the GPU, and often that code is still very much performance-critical. This paper demonstrates that the code that the CPU will be expected to execute in an integrated CPU-GPU environment is profoundly different than the code it has been optimized for over the past many generations. The characteristics of this new code should drive future CPU design and architecture. Specifically, this work shows that post-GPU code tends to have lower ILP, significantly more difficult to predict loads, harder to predict stores, and more difficult branch prediction. Post-GPU code exhibits smaller gains from the availability of multiple cores because of reduced thread level parallelism.

1. Introduction

Fueled by high computational throughput and energy efficiency, we have seen the quick adoption of GPUs as general purpose computing engines in recent years. We are seeing heavier integration of the CPU and the GPU, including the GPU appearing on the same die, further decreasing barriers to use of the GPU to offload the CPU. Much effort has been made to adapt GPU designs to anticipate this new partitioning of the computation space, including better programming models, more general processing units with support for control flow, etc. However, little attention has been placed on the CPU and how it needs to adapt to this change.

This paper demonstrates that the coming era of CPU and GPU integration requires us to rethink the design and architecture of the CPU. We show that the code the CPU will run, once appropriate computations are mapped to the GPU, has significantly different characteristics than the original code (which previously would have been mapped entirely to the CPU).

Modern GPUs contain hundreds of ALUs, hardware thread management, and access to fast on-chip and high-bandwidth external memories. This translates to peak performance of teraFlops per device [16]. There has also been an emergence of new application domains [2] capable of utilizing this performance. These new applications often distill large amounts of data. GPUs have been architected to exploit application parallelism even in the face of high memory latencies. Reported speedups of 10 - 100× are common, although another study shows speedups over an optimized multicore CPU of 2.5× [15].

These speedups by no means imply that CPU performance is no longer critical. Many applications do not map at all to GPUs; others map only a portion of their code to the GPU. Examples of the former include applications with irregular control flow and without high data-level parallelism, as exemplified by many of the SPECint applications. Even for applications with data-level parallelism, there are often serial portions that are still more effectively executed by the CPU. Further, GPU programming currently requires considerable programmer effort, and that effort grows rapidly as the code maps less cleanly to the GPU. As a result, it is most common to only map to the GPU those portions of the code which map easily and cleanly.

Even when a significant portion of the code is mapped to the GPU, the CPU portion will in many cases be performance critical. Consider the case of Kmeans.We study an optimized GPU implementation from the Rodinia [3] benchmark suite. The GPU implementation achieves a speedup of 5× on kernel code. Initially, about 50% of execution time is non-kernel code, yet because of the GPU acceleration, over 4/5 of execution time is spent in the CPU and less than 1/5 is spent on the GPU.

Kumar, et al. [14] argue that the most efficient heterogeneous designs for general-purpose computation contain no general-purpose cores (i.e., cores that run everything well), but rather cores that each run a subset of codes well. The GPU already exemplifies that, running some code lightning fast, other code very poorly. As one of the first steps toward core heterogeneity will likely be CPU-GPU integration, the general-purpose CPU need no longer be fully general-purpose. It will be more effective if it becomes specialized to the code that cannot run on the GPU. This research seeks to understand the nature of that code, and begin to identify the direction in which that should push future CPU designs.

When we compare the code running on the CPU before and after CPU integration, we find several profound changes. We see significant decreases in ILP, especially for large window sizes (10.9% drop). We see significant increases in the percentage of "hard" loads (17.2%) and "hard" stores (12.7%). We see a dramatic overall increase in the percentage of "hard" branches, which translates into a large increase in the mispredict rate of a reasonable branch predictor (55.6%). Average thread level parallelism (defined by 32-core speedup), drops from 5.5 to 2.2.

2. Background

Initial attempts at using GPUs for general purpose computations used corner cases of the graphics APIs [17]. Programmers mapped data to the available shader buffer memory and used the graphics-specific pipeline to process data. NVIDIA's CUDA and AMD's Brook+ platform added hardware to support general computations and exposed the multi-threaded hardware via a programming interface. With GPU hardware becoming flexible, new programming paradigms like OpenCL emerged. Typically, 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. While this computing model is closer to traditional computing models, it has several limitations. Programming GPUs still requires architecture-specific optimizations, which impacts perfor-

Benchmark	Suite	Application Domain	GPU Kernels	Normalized Kernel Speedup (×)	GPU Mapped Portions	Implementation Source
Kmeans	Rodinia	Data Mining	2	5.0	Find and update cluster center	Che et al. [3]
H264	Spec2006	Multimedia	2	12.1	Motion estimation and intra coding	Hwu et al. [11]
SRAD	Rodinia	Image Processing	2	15.0	Equation solver portions	Che et al. [3]
Sphinx3	Spec2006	Speech Recognition	1	17.7	Gaussian mixture models	Harish et al. [9]
Particlefilter	Rodinia	Image Processing	2	32.0	FindIndex computations	Goomrum et al. [7]
Blackscholes	Parsec	Financial Modeling	1	13.7	BlkSchlsEqEuroNoDiv routine	Kolb et al. [13]
Swim	Spec2000	Water Modeling	3	25.3	Calc1, calc2 and calc3 kernels	Wang et al. [26]
Milc	Spec2006	Physics	18	6.0	SU(3) computations across FORALLSITES	Shi et al. [20]
Hmmer	Spec2006	Biology	1	19.0	Viterbi decoding portions	Walters et al. [25]
LUD	Rodinia	Numerical Analysis	1	13.5	LU decomposition matrix operations	Che et al. [3]
Streamcluster	Parsec	Physics	1	26.0	Membership calculation routines	Che et al. [3]
Bwaves	Spec2006	Fluid Dynamics	3	18.0	Bi-CGstab algorithmn	Ruetsche et al. [18]
Equake	Spec2000	Wave Propagation	2	5.3	Sparse matrix vector multiplication (smvp)	Own implementation
Libquantum	Spec2006	Physics	4	28.1	Simulation of quantum gates	Gutierrez et al. [8]
Ammp	Spec2000	Molecular dynamics	1	6.8	Mm_fv_update_nonbon function	Own implementation
CFD	Rodinia	Fluid Dynamics	5	5.5	Euler equlation solver	Solano-Quinde et al. [22]
Mgrid	Spec2000	Grid Solver	4	34.3	Resid, psinv, rprj3 and interp functions	Wang et al. [26]
LBM	Spec2006	Fluid Dynamics	1	31.0	Stream collision functions	Stratton et al. [23]
Leukocyte	Rodinia	Medical Imaging	3	70.0	Vector flow computations	Che et al. [3]
ART	Spec2000	Image Processing	3	6.8	Compute_train_match and values_match functions	Own implementation
Heartwall	Rodinia	Medical Imaging	6	7.9	Search, convolution etc. in tracking algorithm	Szafaryn et al. [24]
Fluidanimate	Parsec	Fluid Dynamics	6	3.9	Frame advancement portions	Sinclair et al. [21]

Table 1. CPU-GPU Benchmark Description.

mance portability. There is also performance overhead resulting from separate discrete memory used by GPUs.

Recently AMD (Fusion APUs), Intel (Sandy Bridge), and ARM (MALI) have released solutions that integrate general purpose programmable GPUs together with CPUs on the same chip. In this computing model, the CPU and GPU may share memory and a common address space. Such sharing is enabled by the use of an integrated memory controller and coherence network for both the CPU and GPU. This promises to improve performance because no explicit data transfers are required between the CPU and GPU, a feature sometimes known as zero-copy [1]. Further, programming becomes easier because explicit GPU memory management is not required.

3. Benchmarks

Over the last few years a large number of CPU applications have been ported to GPUs. Some implementations almost completely map to the GPU while other applications only map certain kernel codes to the GPU. For this study, we seek to examine a spectrum of applications with varying levels of GPU offloading.

We rely as much as possible on published implementations. This ensures that the mapping between GPU code and CPU code would not be driven by our biases or abilities, but rather by the collective wisdom of the community. We make three exceptions, for particularly important applications (SPEC) where the mapping was clear and straightforward. We perform our own CUDA implementations and used those results for these benchmarks.

We use 3 mechanisms to identify the partitioning of the application between the CPU and GPU. First, if the GPU implementation code was available in the public domain, we study it to identify CPU mapped portions. If the code was not available, we obtain the partitioning information from publications. Lastly we ported the three mentioned benchmarks to the GPU ourselves. Table 1 summarizes the characteristics of our benchmarks. The table lists the GPU mapped portions, and provides statistics such as GPU kernel speedup. The kernel speedups reported in the table are from various public domain sources, or our own GPU implementations. Since different publications tend to use different processor baselines and/or different GPUs, we normalized numbers to a single core AMD Shanghai processor running at 2.5GHz and NVIDIA GTX 280 GPU with 1.3GHz shader frequency. We used published SPECrate numbers and linear scaling of GPU performance with number of SMs/frequency to perform the normalization.

We also measure and collect statistics for pure CPU benchmarks, benchmarks with no publicly known GPU implementation. These, combined with the previously mentioned benchmarks, give us a total of 11 CPU-Only benchmarks, 11 GPU-Heavy benchmarks, and 11 Mixed applications where some, but not all, of the application is mapped to the GPU. We do not show the CPU-Only benchmarks in Table 1, because no CPU-GPU mapping was done.

4. Experimental Methodology

This section describes our infrastructure and simulation parameters. Our goal is to identify fundamental characteristics of the code, rather than the effects of particular architectures.This means, when possible, measuring inherent ILP and characterizing loads, stores, and branches into types, rather than always measuring particular hit rates, etc. We do not account for code that might run on the CPU to manage data movement, for example – this code is highly architecture specific, and more importantly, expected to go away in coming designs. We simulate complete programs whenever possible.

While all original application source code was available, we were limited by the non-availability of parallel GPU implementation source code for several important benchmarks. Hence, we use the published CPU-GPU partitioning information and kernel speedup information to drive our analysis.

We develop a PIN based measurement infrastructure. Using the CPU/GPU partitioning information from each benchmark, we modify the original benchmark code without any modifications for GPU implementation. We insert markers indicating the start and end of GPU code, allowing our microarchitectural simulators built on top of PIN to selectively measure CPU and GPU code characteristics. 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 the following steps. First, the proportion of application time that gets mapped to the GPU/CPU is calculated. This is done by inserting time measurement routines in marker functions and running the application on the CPU. Next, we use the normalized speedups to estimate the CPU time with the GPU. For example, consider an application with 80% of execution time mapped to the GPU and a normalized kernel speedup of 40×. Originally, just 20% of the execution time is spent on the CPU. However, post-GPU, 20 / (20 + 80/40) × 100% or about 91% of time is spent executing on the CPU. Time with conservative speedups was obtained by capping the maximum possible GPU speedup value to 10.0. A value of 10.0 was used as a conservative

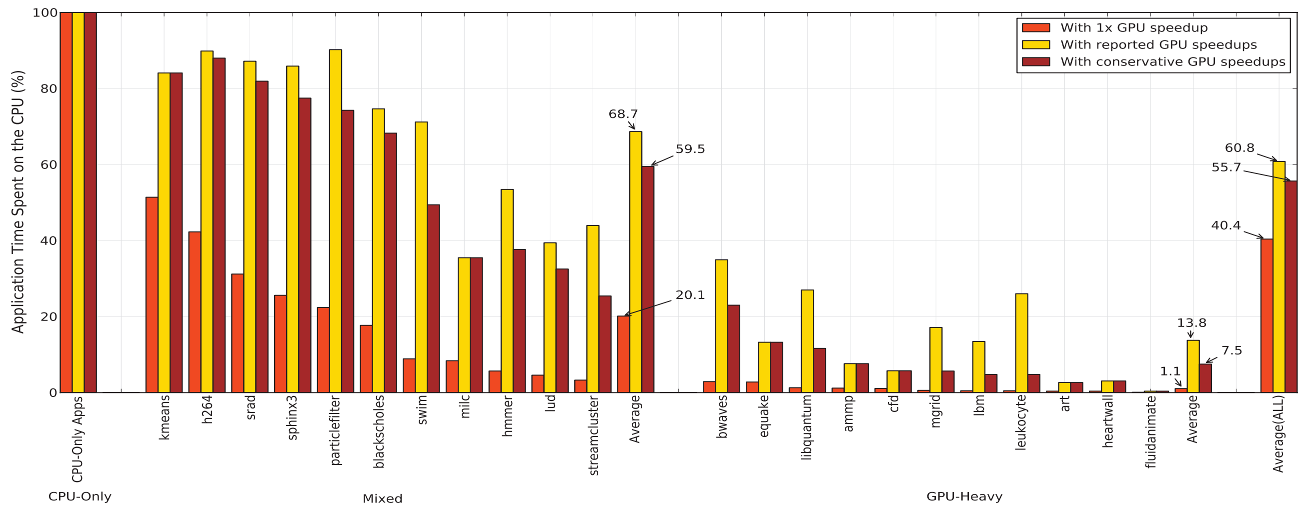


Figure 1. Time spent on the CPU. The 11 CPU-Only applications are summarized, since those results do not vary.

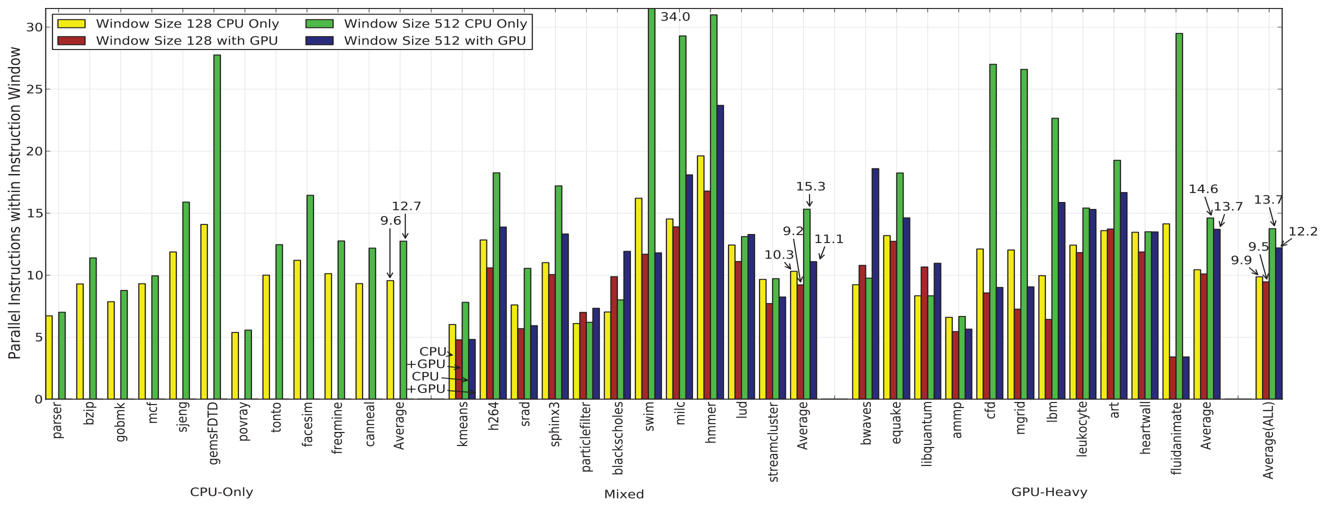


Figure 2. Instruction level parallelism with and without GPU. Not all bars appear in the CPU-Only applications, since they do not vary post-GPU. This is repeated in future plots.

single-core speedup cap [15]. Hence, for the prior example, post-GPU with conservative speedups 20 / (20 + 80/10) × 100% or about 71% of time is spent executing on the CPU.

We categorize loads and stores into four categories, based on measurements on each of the address streams. Those categories are static (address is a constant), strided (predicted with 95% accuracy by a stride predictor that is able to track up to 16 strides per PC), patterned (predicted with 95% accuracy by a large Markov predictor with 8192 entries, 256 previous addresses, and 8 next addresses), and hard (all other loads or stores).

Similarly, we categorize branches as biased (95% taken or not taken), patterned (95% predicted by a large local predictor, using 14 bits of branch history), correlated (95% predicted by a large gshare predictor, using 17 bits of global history), and hard (all other branches). To measure branch mispredict rates, we construct a tournament predictor out of the mentioned gshare and local predictors, combined through a large chooser.

We use Microarchitecture Independent Workload Characterization (MICA) [10] to obtain instruction level parallelism information. MICA calculates the perfect ILP by assuming perfect branch prediction and caches. Only true dependencies affect the ILP. We modify the MICA code to support instruction windows up to 512 entries.

We use a simple definition of thread-level parallelism, based on real machine measurements, and exploiting parallel implementations available for Rodinia, Parsec, and some Spec2000 (those in SpecOMP 2001) benchmarks. Again restricting our measurements to the CPU code marked out in our applications, we define thread level parallelism as the speedup we get on an AMD Shanghai quad core × 8 socket machine. The TLP results, then, cover a subset of our applications for which we have credible parallel CPU implementations.

5. 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 partition applications into three groups — those where no attempt has been made to map code to the GPU (CPU-Only), those where the partitioning is a bit more evenly divided (Mixed), and those where nearly all the code is mapped to the GPU (GPU-Heavy). We first look at CPU time – what portion of the original execution time still gets mapped to the CPU, and

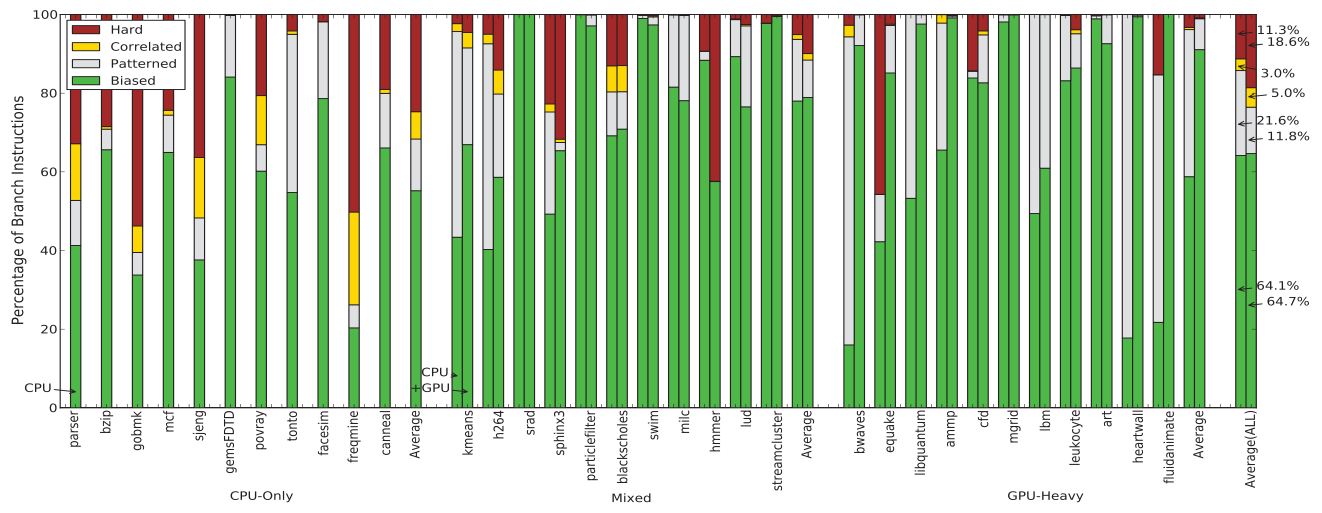


Figure 3. Distribution of branch types with and without GPU.

what is the expected CPU time spent running that code. We then go on to examine other dimensions of that code that still runs on the CPU.

5.1 CPU Execution Time

We start by measuring time spent on the CPU. To identify the utilization and performance-criticality of the CPU after GPU offloading, we calculate the percentage of time in CPU execution after the GPU mapping takes place. We use, initially, the reported speedups from the literature for each GPU mapping.

The first bar in Figure 1 is the percentage of the original code that gets mapped to the CPU. The other two bars represent time actually spent on the CPU (as a fraction of total run time), assuming that the CPU and GPU run separately (if they execute in parallel, CPU time increases further). Thus, the second and third bars account for the reported speedup expected on the GPU. The only difference is the third bar assumes GPU speedup is capped at 10×. For the mixed set of applications in the middle, even though 80% of the code on average is mapped to the GPU, the CPU is still the bottleneck. Even for the GPU-heavy set on the right, the CPU is executing 7-14% of the time. Overall, the CPU is still executing more often than the GPU and remains highly performance-critical. The benchmarks are sorted by CPU time – we'll retain this ordering for subsequent graphs.

In future graphs, we will use the conservative cpu time (third bar) to weight our average (after GPU integration) results – e.g., if you were to run sphinx3 and hmmer in equal measure, the CPU would be executing sphinx3 code about twice as often as hmmer code after CPU-GPU integration.

5.2 ILP

ILP captures the inherent parallelism in the instruction stream – it can be thought of as measuring (the inverse of) the dependence critical path through the code. For out-of-order processors, ILP is heavily dependent on window size – the number of instructions the processor can examine at once looking for possible parallelism.

As seen in Figure 2, in 17 of the 22 applications, ILP drops noticeably, particularly for large window sizes. For *swim, milc, cfd, mgrid and fluidanimate*, it drops by almost half. Between the outliers (ILP actually increases in 5 cases), and the damping impact of the non-GPU applications, the overall effect is a 10.9% drop in ILP for larger window sizes and a 4% drop for current generation window sizes. For the mixed applications, the result is much more striking, a 27.5% drop in ILP for the remaining CPU code. In particular, we are seeing that potential performance gains from large windows is significantly degraded in the absence of the GPU code.

In the common case, independent loops are being mapped to the GPU. Less regular code, and loops with loop-carried dependencies restricting parallelism are left on the CPU. This is the case with h264 and milc, for example; key, tight loops with no critical loop-carried dependencies are mapped to the GPU, leaving less regular and more dependence-heavy code on the CPU.

5.3 Branch Results

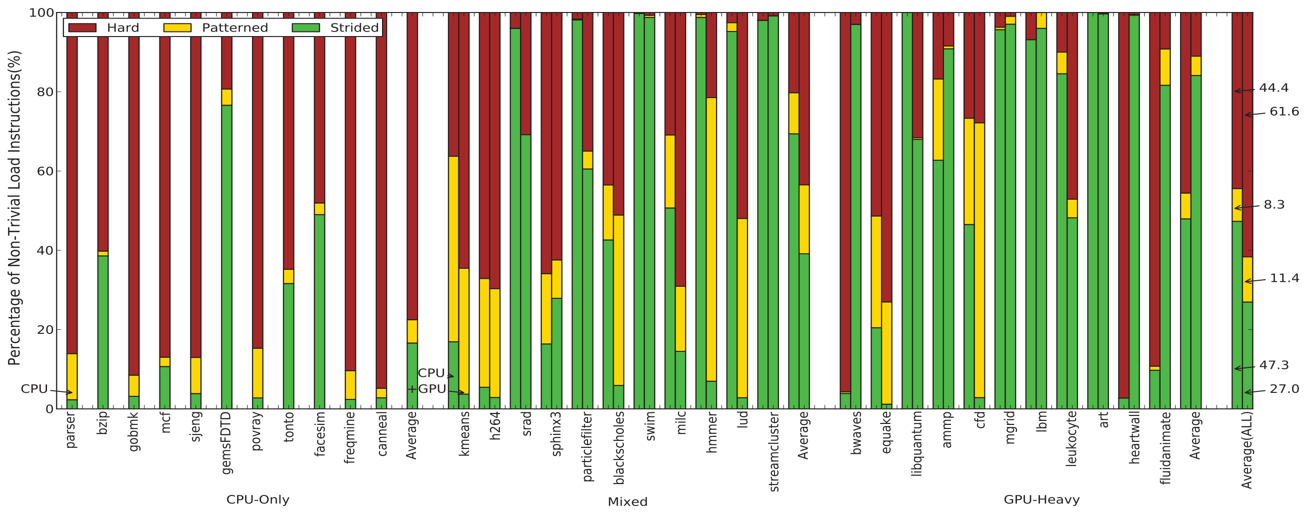
We classify static branches into four categories. The categories are biased (nearly always taken or not taken), patterned (easily captured by a local predictor), correlated (easily captured by a correlated predictor), or hard (none of the above). Figure 3 plots the distribution of branches found in our benchmarks.

Overall, we see a significant increase in hard branches. In fact, the frequency of hard branches increases by 65% (from 11.3% to 18.6%). The increase in hard branches in the overall average is the result of two factors – the high concentration of branches in the CPU-only workloads (which more heavily influence the average) and the marked increase in hard branches in the Mixed benchmarks. The hard branches are primarily replacing the reduced patterned branches, as the easy (biased) branches are only reduced by a small amount.

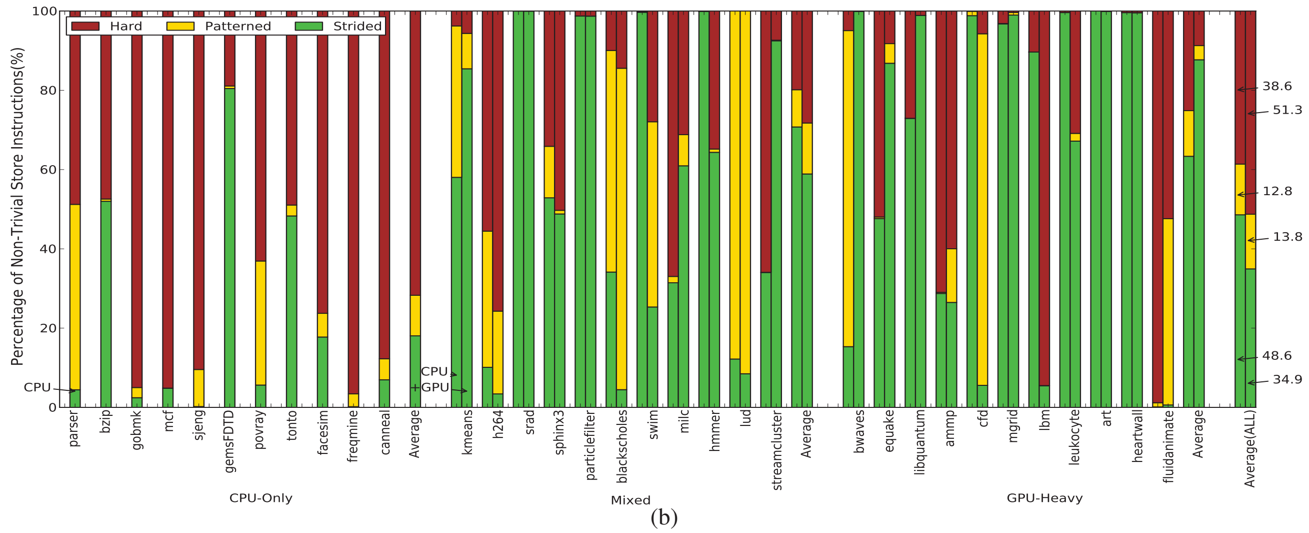
Some of the same effects discussed in the previous section apply here. Small loops with high iteration counts, dominated by looping branch behavior, are easily moved to the GPU (e.g., *h264* and *hmmer*) leaving code with more irregular control flow behavior.

The outliers (contrary results) in this case are instructive, however. *equake and cfd* map data-intensive loops to the GPU. That includes data-dependent branches, which in the worst case can be completely unpredictable.

Even with individual branches getting hard to predict, it is not clear prediction gets worse, as it is possible that with fewer static branches being predicted, aliasing would be decreased. However, experiments on a realistic branch predictor confirmed that the new CPU code indeed stresses the predictor heavily. We found the frequency of mispredicts, for the modeled predictor, to increases dramatically, by 56% (from 2.7 to 4.2 misses per kilo instructions). The graph is not shown here to conserve space. The increase in misses per instruction is primarily a reflection of the increased overall mispredict rate, as the frequency of branches per instruction



(a)



(b)

Figure 4. Distribution of (a) load types and (b) store types with and without GPU.

actually changes by less than 1% between the pre-GPU and post-GPU code.

These results indicate that a branch predictor tuned for generic CPU code may in fact not be sufficient for post-GPU execution.

5.4 Load and Store Results

Typically, code maps to the GPU most effectively when the memory access patterns are regular and ordered. This means we would expect to see a significant drop in ordered (easy) accesses for the CPU.

Figure 4(a) shows the classification of CPU loads. What is graphed in this figure is the breakdown of loads as a percentage of all non-static loads. That is, we have already taken out those loads that will be trivially handled by the cache. In this figure, we see a sharp increase in hard loads, which is perhaps more accurately characterized as a sharp decrease in strided loads.

Thus, of the non-trivial loads that remain, a much higher percentage of them 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, and hard loads increase very significantly, to the point where they are the dominant type. Some applications (e.g., *lud* and *hmmer*) go from being almost completely strided, to the point where a strided prefetcher is useless.

kmeans, *srad* and *milc* each show a sharp increase in the number of hard loads. We find that the key kernel of *kmeans* generates highly regular, strided loads. This kernel is offloaded to the GPU. *srad* and *milc* are similar.

Though the general trend shows an increase in hard loads, we see a notable exception in *bwaves*, in which an important kernel with highly irregular loads is successfully mapped to the GPU.

Figure 4(b) shows that these same trends are also exhibited by the store instructions, as again the strided stores are being reduced, and hard stores increase markedly. Interestingly, the source of the shift is different. In this case, we do not see a marked decrease in the amount of easy stores in our CPU-GPU workloads. However, the high occurrence of hard stores in our CPU-Only benchmarks results in a large increase in hard stores overall.

Similar to the loads, we see that many benchmarks have kernels with strided stores which go to the GPU. This is the case with *swim* and *hmmer*. On the other hand, in *bwaves* and *equake*, the code that gets mapped to the GPU does irregular writes to an unstructured grid.

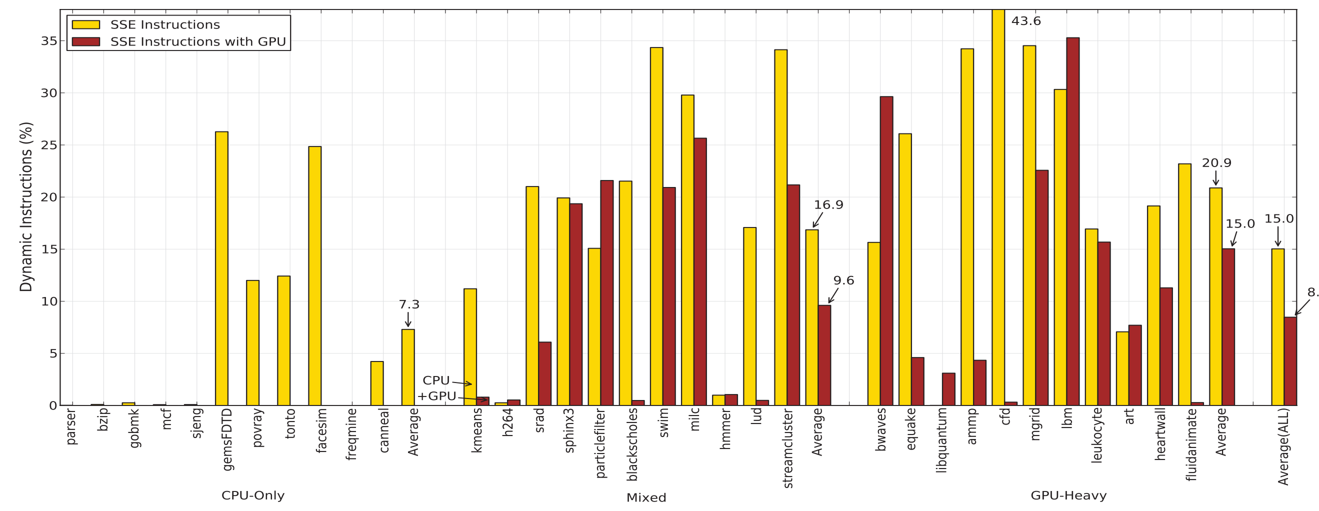


Figure 5. Frequency of vector instructions with and without GPU.

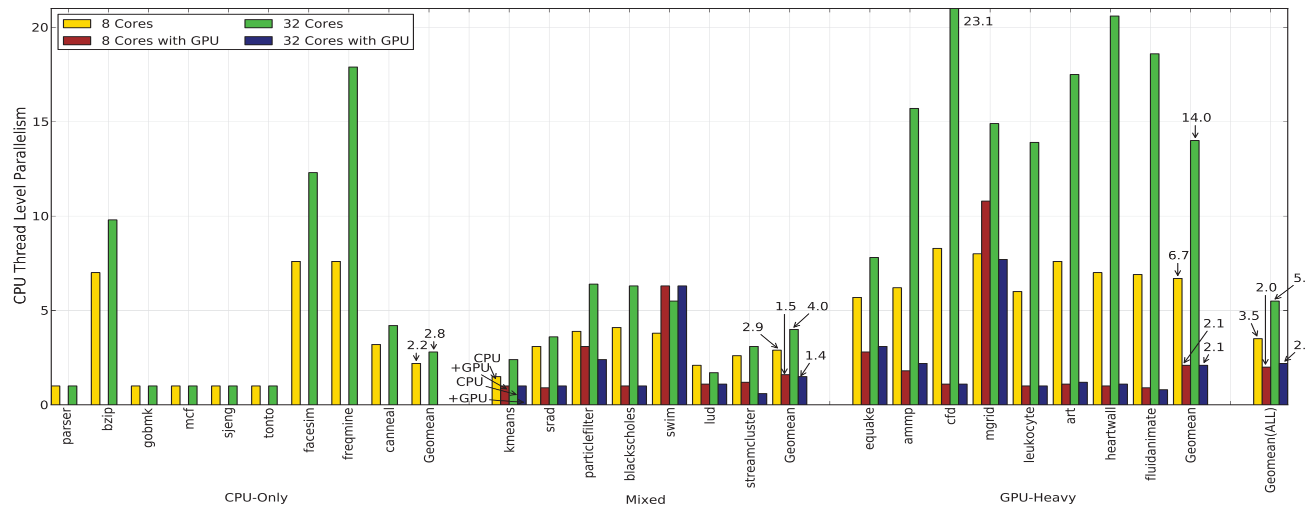


Figure 6. Thread level parallelism with and without GPU.

5.5 Vector Instructions

We were also interested in the distribution of instructions, and how it changes post-GPU. Somewhat surprisingly, we find little change in the mix of integer and floating point operations. However, we find that the usage of SSE instructions drops significantly, as shown in figure 5. We see an overall reduction of 44.3% in the usage of SSE instructions (from 15.0% to 8.5%). This is an expected result, as the SSE ISA enhancements target in many cases the exact same code regions as the general-purpose GPU enhancements. For example, in *kmeans* the find_nearest_point functions heavily utilizes MMX instructions, but this function gets mapped to the GPU.

5.6 Thread Level Parallelism (TLP)

TLP captures parallelism that can be exploited by multiple cores or thread contexts, enabling us to measure the utility of having an increasing number of CPU cores. Figure 6 shows measured TLP results for our benchmarks.

Let us first consider the GPU-heavy benchmarks. CPU 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 (also true for the mixed benchmarks) – no benchmark's post-GPU code sees any significant gain 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. CPU-Only benchmarks exhibit lower TLP than both the Mixed and GPU-Heavy sets, but do not lose any of that TLP because no code runs on the GPU. Overall, 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 yields a nearly two fold increases in TLP in the original code, post-GPU the TLP grows by just 10% over that region – extra cores are nearly useless.

6. Impact on CPU Design

Good architectural design is tuned for the instruction execution stream that is expected to run on the processor. This work indicates that, for those general purpose CPUs, the definition of "typical" code is in the process of changing. This work is the first attempt to isolate and characterize the code the CPU will now be executing. This section identifies some architectural implications of the changing code base.

Sensitivity to Window Size. It has long been understood that out-of-order processors benefit from large instruction windows. As a result, much research has sought to increase window size, or create the illusion of large windows [6]. While we do not see evidence that large windows are not useful, the incremental gains may be more modest.

Branch Predictors. We show that post-GPU code dramatically increases pressure on the branch predictor. This is despite the fact that the predictor is servicing significantly fewer static branches. Recent trends targeting very difficult branches using complex hardware and extremely long histories [19] seem to be a promising direction because they better attack fewer, harder branches.

Load and Store Prefetching. Memory access will continue to be perhaps the biggest performance challenge of future processors. Our results touch particularly on the design of future prefetchers, which currently have a heavy influence on CPU and memory performance. Stride-based prefetchers are commonplace on modern architectures, but are likely to become significantly less relevant on the CPU. What is left for the CPU are very hard memory accesses. Thus, we expect the existing hardware prefetchers to struggle.

We actually have fewer static loads and stores that the CPU must deal with, but those addresses are now hard to predict. This motivates an approach that devotes significant resources toward accurate prediction of a few problematic loads/stores. Several past approaches had exactly this flavor, but have not yet had a big impact on commercial designs. These include Markov-based predictors [12] which target patterned accesses but can capture very complex patterns, and predictors targeted at pointer-chain computation [4, 5]. These types of solutions should be pursued with new urgency. We have also seen significant research into helper-thread prefetchers which have impacted some compilers and some hardware, but their adoption is still not widespread.

Vector Instructions. SSE instructions have not been rendered unnecessary, but certainly less important. Typical SSE code can be executed faster and at lower power on GPUs. Elimination of SSE support may be unjustified, but every core need not support it. In a heterogeneous design, some cores could drop support for SSE, or even in a homogeneous design, multiple cores could share hardware.

Thread Level Parallelism. Heterogeneous architectures are most effective when diversity is high [14]. Thus, recent trends in which CPU and GPU designs are converging more than diverging are suboptimal. One example is that both are headed to higher and higher core and thread counts. Our results indicate that the CPU will do better by addressing codes that have low parallelism and irregular code, and seeking to maximize single-thread, or few-thread, throughput.

7. Conclusion

As GPUs become more heavily integrated into the processor, they will inherit computations that have been traditionally executed on the CPU. As a result, the nature of the computation that remains on the CPU will change. This research looks at the changing workload of the CPU as we progress towards higher CPU-GPU integration. This changing workload impacts the way that future CPUs should be designed and architected.

This research shows that even when significant portions of the original code are offloaded to the GPU, the CPU is still frequently performance critical. It further shows that the code the CPU is running is different than before GPU offloading along several dimensions. ILP becomes harder to find. Loads become significantly more difficult to prefetch. Store addresses become more difficult as well. Post-GPU code places significantly higher pressure on the branch predictor. We also see a decrease in the importance of vector instructions and the ability to exploit multiple cores. Hence the coming era of CPU-GPU integration requires us to rethink CPU design and architecture.

Acknowledgment

This work was funded in part by NSF grant CCF-1018356 and a grant from AMD.

References

[1] AMD OpenCL Programming Guide. http://developer.amd.com/zones/openclzone.
[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, UC Berkeley, 2006.
[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, L. Sangha, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.
[4] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Micro*, 2002.
[5] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, 2002.
[6] A. Cristal, O. Santana, J. Martinez, and M. Valero. Toward kilo-instruction processors. *ACM TACO*, 2004.
[7] M. A. Goodrum, M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron. Parallelization of particle filter algorithms. In *Workshop on Emerging Applications and Many-Core Architectures*, 2010.
[8] E. Gutierrez, S. Romero, M. Trenas, and E. Zapata. Simulation of quantum gates on a novel GPU architecture. In *International Conference on Systems Theory and Scientific Computation*, 2007.
[9] S. C. Harish, D. Balaji, M. Vignesh, D. Kumar, and V. Adinarayanan. Scope for performance enhancement of CMU Sphinx by parallelising with OpenCL. *Journal of Wisdom Based Computing*, 2011.
[10] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 2007.
[11] W. M. Hwu, D. Kirk, S. Ryoo, C. Rodrigues, J. Stratton, and K. Huang. Performance insights on executing non-graphics applications on CUDA on the NVIDIA GeForce 8800 GTX. *Hotchips*, 2007.
[12] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.
[13] C. Kolb and M. Pharr. Options pricing on the GPU. In *GPU Gems 2*.
[14] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT*, 2006.
[15] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.
[16] NVIDIA. NVIDIA's next generation cuda compute architecture: Fermi, 2009. http://nvidia.com/content/pdf/fermi_white_papers/.
[17] 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 *Computer Graphics Forum*, 2007.
[18] G. Ruetsch and M. Fatica. A CUDA fortran implementation of BWAVES. http://www.pgroup.com/lit/articles/.
[19] A. Seznec. The L-TAGE branch predictor. In *Journal of Instruction-Level Parallelism*, 2007.
[20] G. Shi, S. Gottlieb, and V. Kindratenko. Milc on GPUs. Technical report, NCSA, 2010.
[21] M. Sinclair, H. Duwe, and K. Sankaralingam. Porting CMP benchmarks to GPUs. Technical report, UW Madison, 2011.
[22] L. Solano-Quinde, Z. J. Wang, B. Bode, and A. K. Somani. Unstructured grid applications on GPU: performance analysis and improvement. GPGPU, 2011.
[23] J. Stratton. LBM on GPU. http://impact.crhc.illinois.edu/parboil.php.
[24] L. G. Szafaryn, K. Skadron, and J. J. Saucerman. Experiences accelerating matlab systems biology applications. In *Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits*, 2009.
[25] J. Walters, V. Balu, S. Kompalli, and V. Chaudhary. Evaluating the use of GPUs in liver image segmentation and HMMER database searches. In *International Symposium on Parallel and Distributed Processing*, 2009.
[26] G. Wang, T. Tang, X. Fang, and X. Ren. Program optimization of array-intensive SPEC2K benchmarks on multithreaded GPU using CUDA and Brook+. In *Parallel and Distributed Systems*, 2009.