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

In an integrated CPU-GPU system, the CPU executes code that is profoundly different than in past CPU-only environments. This new code's characteristics should drive future CPU design and architecture. Post-GPU code has lower instruction-level parallelism, more difficult branch prediction, and loads and stores that are significantly harder to predict. Post-GPU code exhibits much smaller gains from the availability of multiple cores, owing to reduced thread-level parallelism.

Manish Arora
Siddhartha Nath
Subhra Mazumdar
Scott B. Baden
Dean M. Tullsen
University of California, San Diego

We've seen the quick adoption of GPUs as general-purpose computing engines in recent years, fueled by high computational throughput and energy efficiency. There is heavier integration of the CPU and GPU, including the GPU appearing on the same die, further decreasing barriers to the 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 and more general processing units with support for control flow. However, researchers have placed little attention on the CPU and how it must adapt to this change.

This article demonstrates that the coming era of CPU and GPU integration requires us to rethink the CPU's design and architecture. 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).

Background

Modern GPUs contain hundreds of arithmetic logic units (ALUs), hardware thread management, and access to fast on-chip and high-bandwidth external memories. This translates to peak performance of teraflops per device.[1] We've also seen an emergence of new application domains that can utilize this performance.[2] 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× to 100× are common, although another study shows speedups over an optimized multicore CPU of 2.5×.[3]

These speedups do not imply that CPU performance is no longer critical. Many applications don't 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 Standard Performance Evaluation Corporation integer (SPECint) applications.

Published by the IEEE Computer Society

Even for applications with data-level parallelism, there are often serial portions that are still more effectively executed by the CPU. Furthermore, 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 common to only map to the GPU those portions of the code that 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 K-Means. We studied an optimized GPU implementation from the Rodinia benchmark suite.[4] The GPU implementation achieves a speedup of 5× on kernel code. Initially, about 50 percent of execution time is nonkernel code, yet because of the GPU acceleration, more than four-fifths of the execution time is spent in the CPU and less than one-fifth is spent on the GPU.

Kumar et al. argue that the most efficient heterogeneous designs for general-purpose computation contain no general-purpose cores (that is, cores that run everything well), but rather cores that each run a subset of codes well.[5] The GPU already exemplifies that, running some code lightning fast and other code poorly. Because 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 can't run on the GPU. Our research seeks to understand the nature of that code, and begins to identify the direction in which to 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 instruction-level parallelism (ILP), especially for large window sizes (a 10.9 percent drop). We see significant increases in the percentage of hard loads (17.2 percent) and hard stores (12.7 percent). We see a dramatic overall increase in the percentage of hard branches, which translates into a large increase in the misprediction rate of a reasonable branch predictor (55.6 percent). Average thread-level parallelism (TLP; defined by 32-core speedup) drops from 5.5 to 2.2.

Initial attempts at using GPUs for general-purpose computations used corner cases of the graphics APIs.[6] 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 multithreaded hardware via a programming interface. With GPU hardware becoming flexible, new programming paradigms such as OpenCL emerged. Typically, the programmer receives an abstraction of a separate GPU memory address space similar to CPU memory where data can be allocated and threads launched. Although this computing model is closer to traditional computing models, it has several limitations. Programming GPUs still require architecture-specific optimizations, which impacts performance portability. There is also performance overhead resulting from separate discrete memory used by GPUs.

Recently, AMD (Fusion accelerated processing units), 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 can 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.[7] Furthermore, programming becomes easier because explicit GPU memory management isn't required.

Benchmarks

Over the past 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 examined a spectrum of applications with varying levels of GPU offloading.

We relied as much as possible on published implementations so that the mapping

between GPU code and CPU code wouldn't be driven by our biases or abilities, but rather by the collective wisdom of the community. We made three exceptions for particularly important applications (SPEC) where the mapping was clear and straightforward. We performed our own CUDA implementations and used those results for these benchmarks.

We used three 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 studied it to identify CPU-mapped portions. If the code wasn't available, we obtained the partitioning information from publications. Finally, we ported the three mentioned benchmarks to the GPU ourselves. Table 1 summarizes our benchmarks' characteristics. 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. Because different publications tend to use different processor baselines and GPUs, we normalized the numbers to a single-core AMD Shanghai processor running at 2.5 GHz and Nvidia GTX 280 GPU with 1.3-GHz shader frequency. We used published SPECrate numbers and linear scaling of GPU performance with a number of Streaming Multiprocessors (SMs) and frequency to perform the normalization.

We also measured and collected statistics for pure CPU benchmarks—benchmarks with no publicly known GPU implementation. Combined with the previously mentioned benchmarks, these 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 don't show the CPU-only benchmarks in Table 1, because no CPU-GPU mapping was done.

Experimental methodology

Here, we describe our infrastructure and simulation parameters. We aimed to identify fundamental code characteristics, rather than the particular architectures' effects. This means, when possible, measuring inherent ILP and characterizing loads, stores, and branches into types, rather than always measuring particular hit rates. We didn't 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 simulated complete programs whenever possible.

Although all original application source code was available, we were limited by the nonavailability of parallel GPU implementation source code for several important benchmarks. Thus, we used the published CPU-GPU partitioning information and kernel speedup information to drive our analysis.

We developed a PIN-based measurement infrastructure. Using each benchmark's CPU/GPU partitioning information, we modified the original benchmark code without any modifications for GPU implementation. We inserted 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. We simulated all benchmarks for the largest available input sizes. Programs were run to completion or for at least 1 trillion instructions.

We calculated the CPU time using the following steps. First, we calculated the proportion of application time that gets mapped to the GPU/CPU by inserting time measurement routines in marker functions and running the application on the CPU. Next, we used the normalized speedups to estimate the CPU time with the GPU. For example, consider an application with 80 percent of execution time mapped to the GPU and a normalized kernel speedup of 40×. Originally, just 20 percent of the execution time was spent on the CPU. However, post-GPU, $20/(20 + 80/40) \times 100$ percent, or about 91 percent of time, is spent executing on the CPU. We obtained time with conservative speedups by capping the maximum possible GPU speedup value to 10.0. We used a value of 10.0 as a conservative single-core speedup cap.[3] Hence, for the prior example, post-GPU with conservative speedups of $20/(20 + 80/10) \times 100$ percent, or about 71 percent of time, is spent executing on the CPU.

Table 1. CPU-GPU benchmarks used in our study.

Benchmark	Suite	Application domain	GPU kernels	Normalized kernel speedup (×)	GPU-mapped portions	Implementation source
K-Means	Rodinia	Data mining	2	5.0	Find and update cluster center	Che et al.[4]
H264	Spec2006	Multimedia	2	12.1	Motion estimation and intracoding	Hwu et al.[8]
SRAD	Rodinia	Image processing	2	15.0	Equation solver portions	Che et al.[4]
Sphinx3	Spec2006	Speech recognition	1	17.7	Gaussian mixture models	Harish et al.[9]
Particlefilter	Rodinia	Image processing	2	32.0	FindIndex computations	Goodrum et al.[10]
Blackscholes	Parsec	Financial modeling	1	13.7	BlkSchlsEqEuroNoDiv routine	Kolb et al.[11]
Swim	Spec2000	Water modeling	3	25.3	Calc1, calc2, and calc3 kernels	Wang et al.[12]
Milc	Spec2006	Physics	18	6.0	SU(3) computations across FORALLSITES	Shi et al.[13]
Hmmer	Spec2006	Biology	1	19.0	Viterbi decoding portions	Walters et al.[14]
LUD	Rodinia	Numerical analysis	1	13.5	LU decomposition matrix operations	Che et al.[4]
Streamcluster	Parsec	Physics	1	26.0	Membership calculation routines	Che et al.[4]
Bwaves	Spec2006	Fluid dynamics	3	18.0	Bi-CGstab algorithm	Ruetsch et al.[15]
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.[16]
Ammp	Spec2000	Molecular dynamics	1	6.8	Mm-fv-update-nonbon function	Own implementation
CFD	Rodinia	Fluid dynamics	5	5.5	Euler equation solver	Solano-Quinde et al.[17]
Mgrid	Spec2000	Grid solver	4	34.3	Resid, psinv, rprj3, and interp functions	Wang et al.[12]
LBM	Spec2006	Fluid dynamics	1	31.0	Stream collision functions	Stratton et al.[18]
Leukocyte	Rodinia	Medical imaging	3	70.0	Vector flow computations	Che et al.[4]
ART	Spec2000	Image processing	3	6.8	Compute_train_match and values_match functions	Own implementation
Heartwall	Rodinia	Medical imaging	6	7.9	Search and convolution in tracking algorithm	Szafaryn et al.[19]
Fluidanimate	Parsec	Fluid dynamics	6	3.9	Frame advancement portions	Sinclair et al.[20]

We categorized loads and stores into four categories on the basis of measurements on each of the address streams. Those categories are *static* (address is a constant), *strided* (predicted with 95-percent accuracy by a stride predictor that can track up to 16 strides per PC), *patterned* (predicted with 95-percent accuracy by a large Markov predictor with 8,192 entries, 256 previous addresses, and 8 next addresses), and *hard* (all other loads or stores).

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

We used the Microarchitecture-Independent Characterization of Applications (MICA) to obtain ILP information.[21] MICA calculates the perfect ILP by assuming perfect branch prediction and caches. Only true dependencies affect the ILP. We modified the MICA code to support instruction windows up to 512 entries.

We used a simple definition of TLP, based on real machine measurements, and exploited 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 defined TLP 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.

Results

Here, 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 what's 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.

CPU execution time

We start by measuring time spent on the CPU. To identify the CPU's utilization and performance criticality 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 runtime), 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 that the third bar assumes GPU speedup is capped at 10×. For the mixed set of applications in the middle, even though 80 percent 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 to 14 percent of the time. Overall, the CPU is still executing more often than the GPU and remains highly performance critical. We sort the benchmarks by CPU time, and we'll retain this ordering for subsequent graphs.

In future graphs, we'll use the conservative CPU time (third bar) to weight our average (after GPU integration) results—for example, 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.

ILP

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

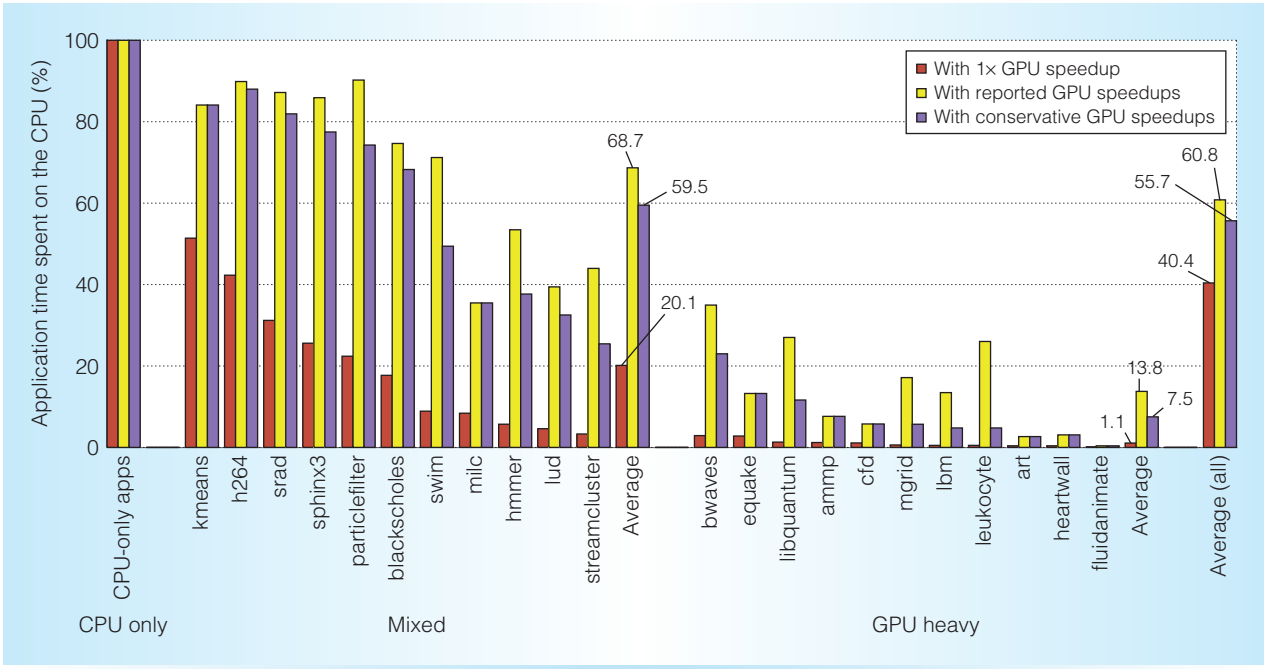

Figure 1. Time spent on the CPU. The 11 CPU-only applications are summarized, because those results do not vary.

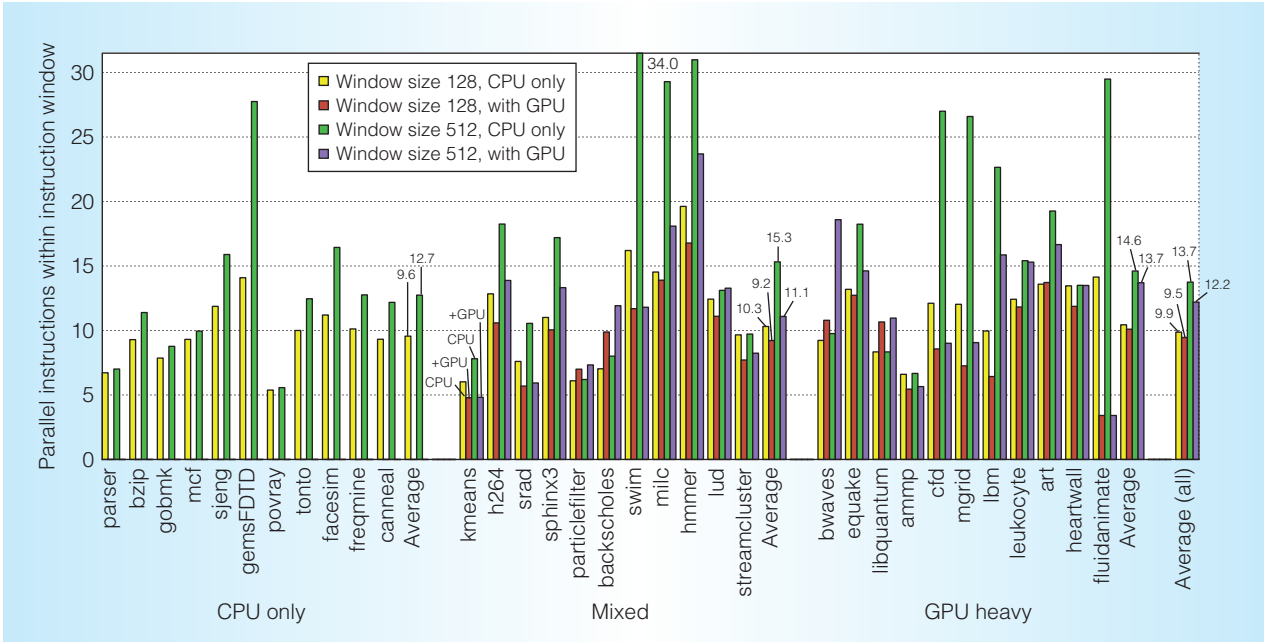

Figure 2. Instruction-level parallelism (ILP) with and without GPU. Not all bars appear in the CPU-only applications because they don't vary post-GPU. This is repeated in future plots.

As Figure 2 shows, 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 five cases), and the non-GPU applications' damping impact, the overall effect is a 10.9-percent drop in ILP for larger

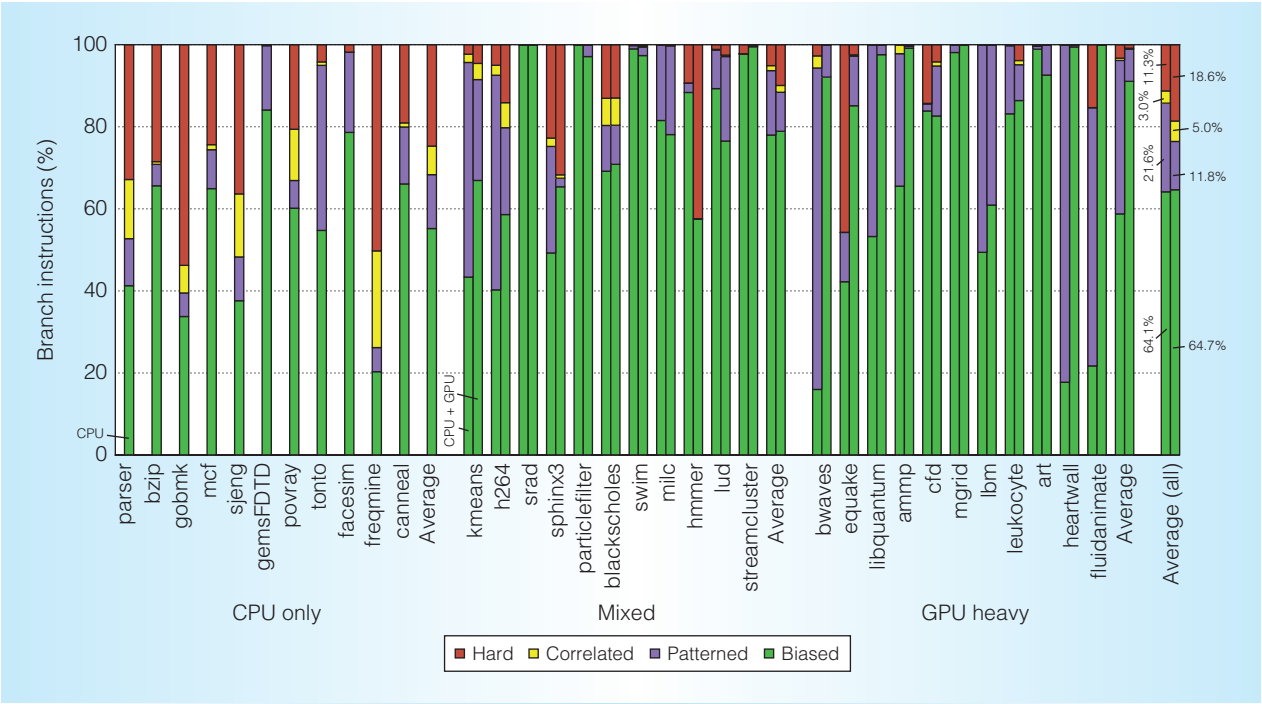

Figure 3. Distribution of branch types with and without GPU.

window sizes and a 4-percent drop for current-generation window sizes. For the mixed applications, the result is much more striking—a 27.5-percent drop in ILP for the remaining CPU code. In particular, we see that potential performance gains from large windows are 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.

Branch results

We classify static branches into four categories: *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 percent (from 11.3 percent to 18.6 percent). 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, because the easy (biased) branches are only reduced by a small amount.

Some of the same effects we discussed earlier apply here. Small loops with high iteration counts, dominated by looping branch behavior, are easily moved to the GPU (such as h264 and hmmer), leaving code with more irregular control-flow behavior.

The outliers (contrary results) in this case are instructive. Both 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 that prediction gets worse, because 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 that the frequency of mispredictions for the modeled predictor increases dramatically, by 56 percent (from 2.7 to 4.2 misses per thousand instructions). We don't show the graph here to conserve space. The increase in misses per instruction primarily reflects the increased overall misprediction rate, because the frequency of branches per instruction actually changes by less than 1 percent between the pre-GPU and post-GPU code.

These results indicate that a branch predictor tuned for generic CPU code might in fact be insufficient for post-GPU execution.

Load and store results

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

Figure 4a shows the classification of CPU loads. The graph shows the breakdown of loads as a percentage of all nonstatic loads. That is, we have already taken out those loads that the cache will trivially handle. 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 nontrivial loads that remain, a much higher percentage of them aren't 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're dominant. Some applications (such as lud and hmmer) go from being almost completely strided, to the point where a strided prefetcher is useless.

The benchmarks 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. Both srad and milc are similar.

Although 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 4b shows that the store instructions also exhibit these same overall trends, 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 don't 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, many benchmarks have kernels with strided stores that 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.

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 Streaming SIMD Extensions (SSE) instructions drops significantly, as Figure 5 shows. We see an overall reduction of 44.3 percent in the usage of SSE instructions (from 15.0 to 8.5 percent). We expected this result because the SSE instruction set architecture (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` function heavily utilizes MMX instructions, but this function gets mapped to the GPU.

TLP

TLP captures parallelism that can be exploited by multiple cores or thread contexts, letting us 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×.

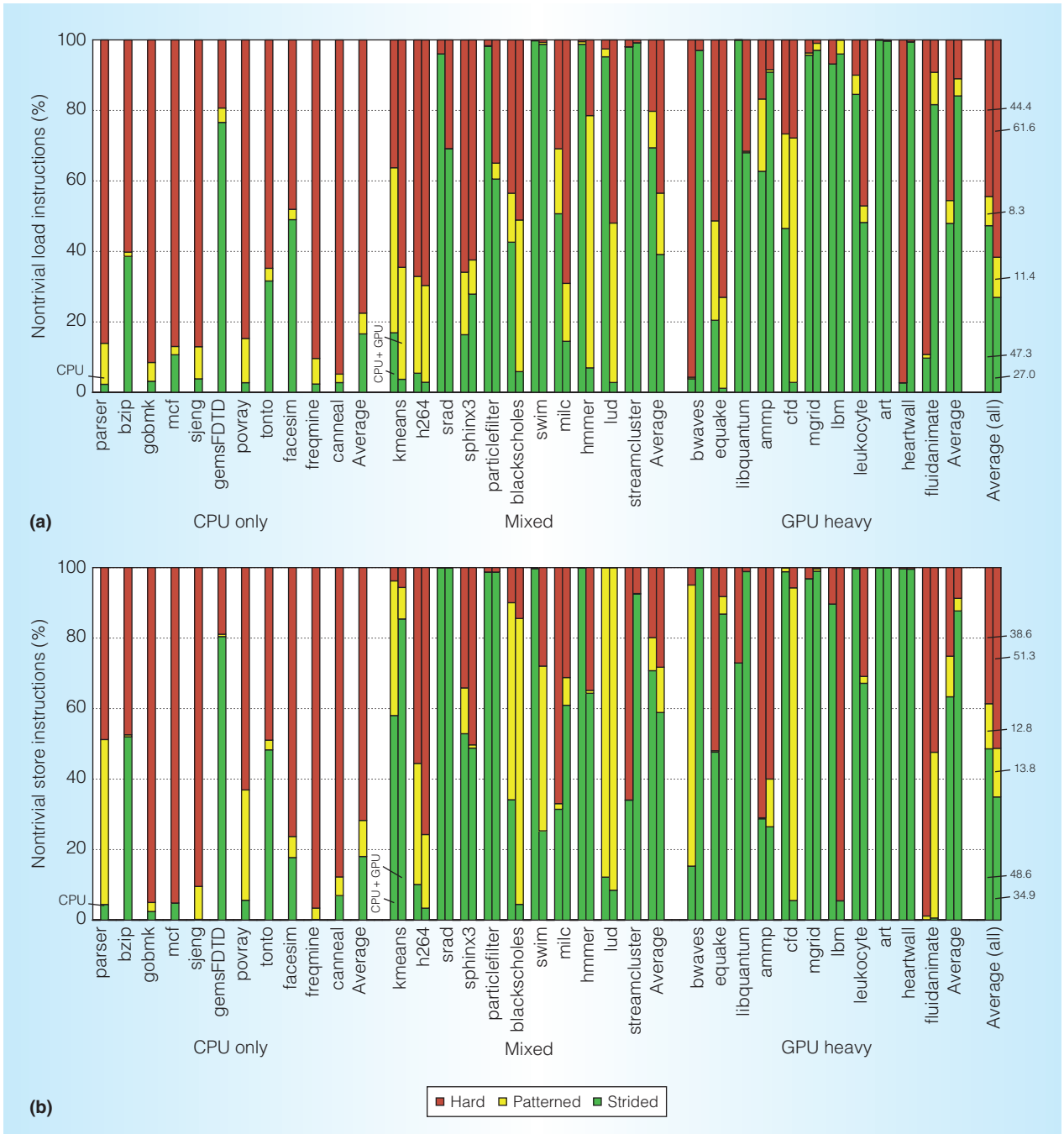

Figure 4. CPU load and store classification. Distribution of load types (a) and store types (b) with and without GPU.

Five benchmarks exhibit no post-GPU TLP; in contrast, five benchmarks originally had speedups greater than 15×. Perhaps the most striking result (also true for the mixed benchmarks) is that no benchmark's post-GPU code sees any significant gain going from eight cores to 32.

Overall for the mixed benchmarks, we again see a considerable reduction in post-GPU TLP; it drops by almost 50 percent for eight cores and about 65 percent 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

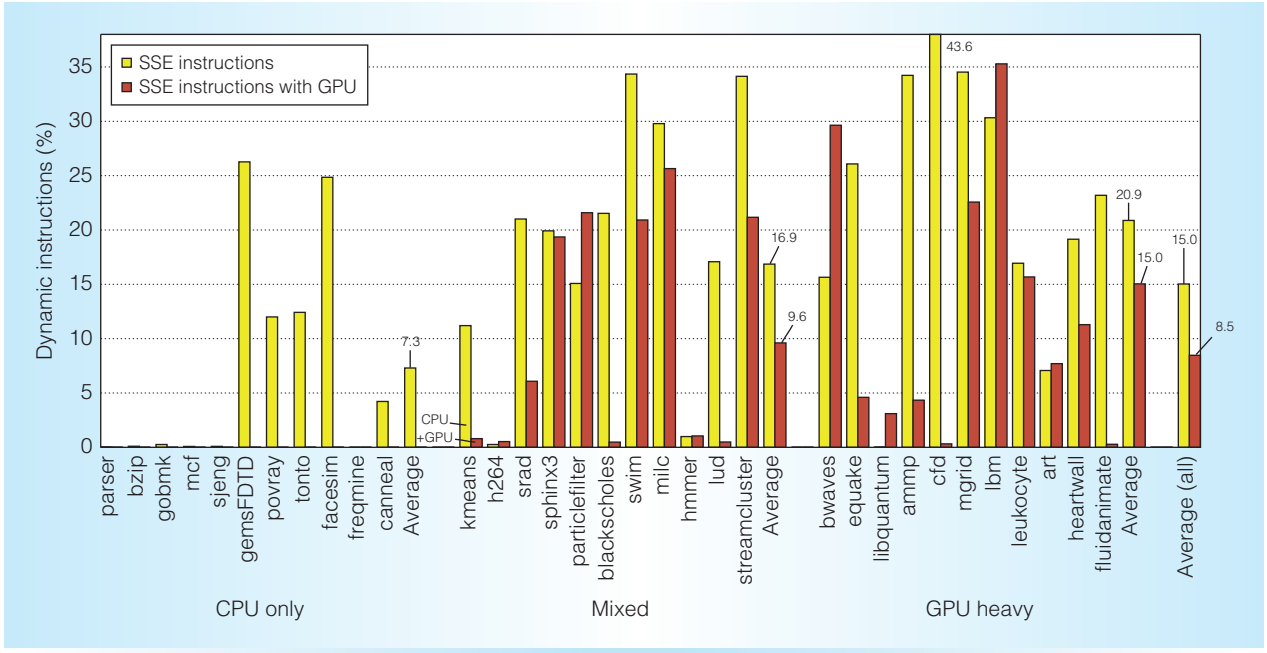

Figure 5. Frequency of vector instructions with and without GPU.

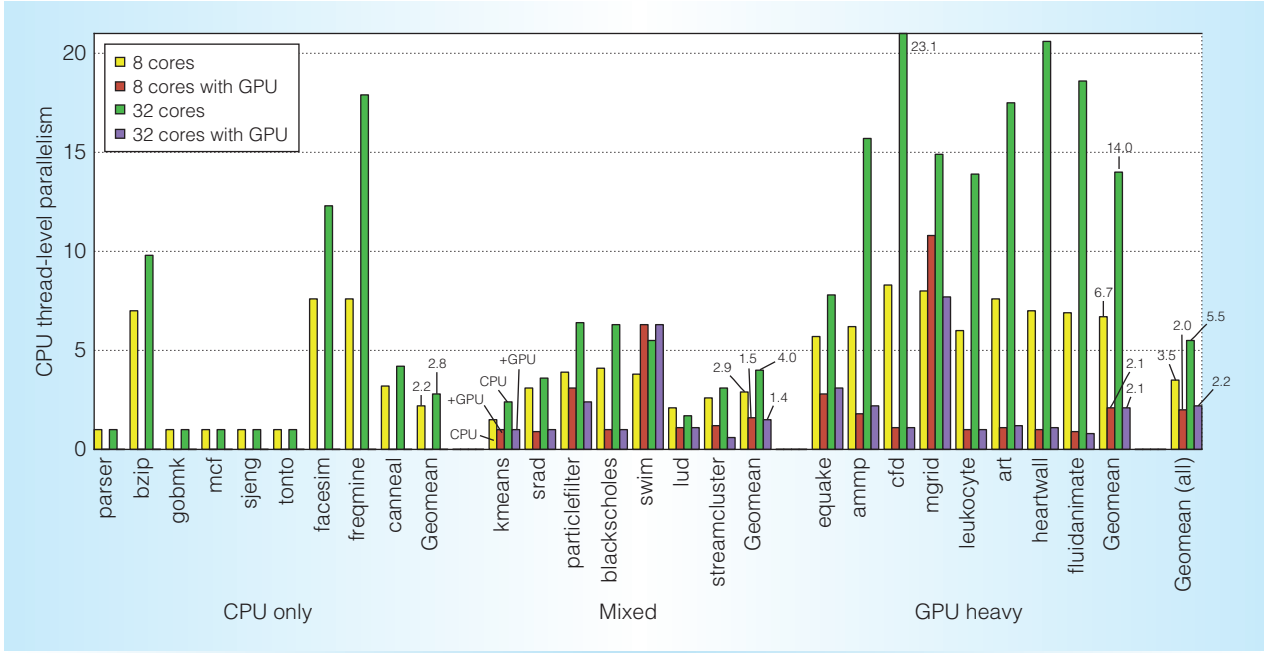

Figure 6. Thread-level parallelism (TLP) with and without GPU.

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; eight-core TLP dropped by 43 percent from 3.5 to 2.0, and 32-core TLP dropped by 60 percent from 5.5 to 2.2. While going from eight cores to 32 cores yields a nearly twofold increase in TLP in

the original code, post-GPU the TLP grows by just 10 percent over that region—extra cores are nearly useless.

Impact on CPU design

Good architectural design is tuned for the instruction execution stream that's expected to run on the processor. This work indicates that, for those general-purpose CPUs, the definition of ''typical'' code is changing. This work is the first attempt to isolate and characterize the code that the CPU will now be executing. Here, we identify 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.[22] Although large windows continue to appear useful, the incremental gains in performance of going to larger windows might be modest.

Branch predictors

We show that post-GPU code dramatically increases pressure on the branch predictor, 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 seem to be a promising direction because they better attack fewer, harder branches.[23]

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 heavily influence CPU and memory performance. Stride-based prefetchers are commonplace on modern architectures but are likely to become significantly less relevant on the CPU. What are 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 accurately predicting a few problematic loads and stores. Several past approaches had exactly this flavor, but haven't yet had a big impact on commercial designs. These include Markov-based predictors,[24] which target patterned accesses but can capture complex patterns, and predictors targeted at pointer-chain computation.[25,26] Researchers should pursue these types of solutions with new urgency. We've also seen significant research into helper-thread prefetchers that have impacted some compilers and hardware, but their adoption still isn't widespread.

Vector instructions

SSE instructions haven't been rendered unnecessary, but they're certainly less important. GPUs can execute typical SSE code faster and at lower power. Elimination of SSE support might 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.

TLP

Heterogeneous architectures are most effective when diversity is high.[5] 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.

As GPUs become heavily integrated into the processor, they inherit computations traditionally executing on the CPU. As a result, the nature of computations that remain on the CPU is changing. This requires us to rethink CPU design and architecture. Chip integrated CPU-GPU systems provide ample opportunities to rethink the design of shared components such as the last-level caches, memory controller, and new techniques to share a total chip power budget. We plan to look at these aspects of CPU-GPU systems in the future. MICRO

Acknowledgments

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

References

1. ''NVIDIA's Next Generation CUDA Compute Architecture: Fermi,'' Nvidia, 2009.
2. K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley,* tech. report, EECS Dept., Univ. of California, Berkeley, 2006.
3. V.W. Lee et al., ''Debunking the 100x GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU,'' *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), ACM, 2010, pp. 451-460.
4. S. Che et al., ''Rodinia: A Benchmark Suite for Heterogeneous Computing,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 09), IEEE CS, 2009, pp. 44-54.
5. R. Kumar, D.M. Tullsen, and N.P. Jouppi, ''Core Architecture Optimization for Heterogeneous Chip Multiprocessors,'' *Proc. 15th Int'l Conf. Parallel Architecture and Compilation Techniques* (PACT 06), ACM, 2006, pp. 23-32.
6. J.D. Owens et al., ''A Survey of General-Purpose Computation on Graphics Hardware,'' *Computer Graphics Forum,* 2007, vol. 26, no. 1, pp. 80-113.
7. A. Munshi et al., *OpenCL Programming Guide,* Addison-Wesley, 2011.
8. W.M. Hwu et al., ''Performance Insights on Executing Nongraphics Applications on CUDA on the NVIDIA GeForce 8800 GTX,'' *Hot Chips 19,* 2007, http://www.hotchips.org/archives/hc19.
9. S.C. Harish et al., ''Scope for Performance Enhancement of CMU Sphinx by Parallelizing with OpenCL,'' *J. Wisdom Based Computing,* Aug. 2011, pp. 43-46.
10. M.A. Goodrum et al., ''Parallelization of Particle Filter Algorithms,'' *Proc. Int'l Conf. Computer Architecture,* Springer-Verlag, 2010, pp. 139-149.
11. C. Kolb and M. Pharr, ''Options Pricing on the GPU,'' *GPU Gems 2,* M. Pharr and R. Fernando, eds., Addison-Wesley, 2005, chapter 45.
12. G. Wang et al., ''Program Optimization of Array-Intensive SPEC2K Benchmarks on Multithreaded GPU Using CUDA and Brook+,'' *Proc. 15th Int'l Conf. Parallel and Distributed Systems,* IEEE CS, 2009, pp. 292-299.
13. G. Shi, S. Gottlieb, and V. Kindratenko, *MILC on GPUs,* tech. report, NCSA, Univ. Illinois, Jan. 2010.
14. J. Walters et al., ''Evaluating the Use of GPUs in Liver Image Segmentation and HMMER Database Searches,'' *Proc. IEEE Int'l Symp. Parallel & Distributed Processing,* IEEE CS, 2009, doi:10.1109/IPDPS.2009.5161073.
15. G. Ruetsch and M. Fatica, ''A CUDA Fortran Implementation of BWAVES,'' http://www.pgroup.com/lit/articles/nvidia_paper_bwaves.pdf.
16. E. Gutierrez et al., ''Simulation of Quantum Gates on a Novel GPU Architecture,'' *Proc. 7th Int'l Conf. Systems Theory and Scientific Computation,* WSEAS, 2007, pp. 121-126.
17. L. Solano-Quinde et al., ''Unstructured Grid Applications on GPU: Performance Analysis and Improvement,'' *Proc. 4th Workshop General Purpose Processing on Graphics Processing Units,* ACM, 2011, doi:10.1145/1964179.1964197.
18. J. Stratton, ''LBM on GPU,'' http://impact.crhc.illinois.edu/parboil.aspx.
19. L.G. Szafaryn, K. Skadron, and J.J. Saucerman, ''Experiences Accelerating Matlab Systems Biology Applications,'' *Workshop Biomedicine in Computing: Systems, Architectures, and Circuits,* 2009.
20. M. Sinclair, H. Duwe, and K. Sankaralingam, *Porting CMP Benchmarks to GPUs,* tech. report 1693, Computer Sciences Dept., Univ. of Wisconsin, Madison, June 2011.
21. K. Hoste and L. Eeckhout, ''Microarchitecture-Independent Workload Characterization,'' *IEEE Micro,* May/June 2007, pp. 63-72.
22. A. Cristal et al., ''Toward Kilo-Instruction Processors,'' *ACM Trans. Architecture and Code Optimization,* Dec. 2004, pp. 389-417.
23. A. Seznec, ''The L-TAGE Branch Predictor,'' *J. Instruction-Level Parallelism,* May 2007; http://www.jilp.org/vol9/v9paper6.pdf.
24. D. Joseph and D. Grunwald, ''Prefetching Using Markov Predictors,'' *Proc. 24th Ann. Int'l Symp. Computer Architecture* (ISCA 97), ACM, 1997, pp. 252-263.
25. J. Collins et al., ''Pointer Cache Assisted Prefetching,'' *Proc. 35th Ann. ACM/IEEE*

Int'l Symp. Microarchitecture, IEEE CS, 2002, pp. 62-73.

26. R. Cooksey, S. Jourdan, and D. Grunwald, ''A Stateless, Content Directed Data Prefetching Mechanism,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2002, pp. 279-290.

Manish Arora is a PhD student in computer science and engineering at the University of California, San Diego. His research interests include heterogeneous GPU-CPU systems and computer architecture for irregular computations. Arora has an MS in computer engineering from the University of Texas at Austin.

Siddhartha Nath is a PhD student in the VLSI CAD lab at the University of California, San Diego. His research interests include network-on-chip modeling for area and power estimation and reliability mechanisms for deep-submicron technologies. Nath has a BE in electrical engineering from the Birla Institute of Technology & Science.

Subhra Mazumdar is a software engineer in the Solaris kernel group at Oracle. His research interests include memory architecture and power management. Mazumdar has an MS in computer engineering from the University of California, San Diego, where he performed the work for this article.

Scott B. Baden is a professor in the Computer Science and Engineering Department at the University of California, San Diego. His research interests include high-performance and parallel computation, focusing on programming abstractions, domain-specific translation, performance programming, adaptive and data-centric applications, and algorithm design. Baden has a PhD in computer science from the University of California, Berkeley. He is a senior member of IEEE and the Society for Industrial and Applied Mathematics (SIAM) and a senior fellow at the San Diego Supercomputer Center.

Dean M. Tullsen is a professor in the Computer Science and Engineering Department at the University of California, San Diego. His research interests include architecture and compilers for multicore and multithreaded architectures. Tullsen has a PhD in computer science from the University of Washington. He is a fellow of IEEE and the ACM.

Direct questions and comments about this article to Manish Arora at the University of California, San Diego, Computer Science and Engineering Department, 9500 Gilman Drive, #0404, La Jolla, CA 92093-0404; marora@eng.ucsd.edu.