PEPSC: A Power-Efficient Processor for Scientific Computing

Ganesh Dasika[1] Ankit Sethia[2] Trevor Mudge[2] Scott Mahlke[2]

[1]ARM R&D
Austin, TX
Email: ganesh.dasika@arm.com

[2]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
Email: {asethia, tnm, mahlke}@umich.edu

***Abstract*—The rapid advancements in the computational capabilities of the graphics processing unit (GPU) as well as the deployment of general programming models for these devices have made the vision of a desktop supercomputer a reality. It is now possible to assemble a system that provides several TFLOPs of performance on scientific applications for the cost of a high-end laptop computer. While these devices have clearly changed the landscape of computing, there are two central problems that arise. First, GPUs are designed and optimized for graphics applications resulting in delivered performance that is far below peak for more general scientific and mathematical applications. Second, GPUs are power hungry devices that often consume 100-300 watts, which restricts the scalability of the solution and requires expensive cooling. To combat these challenges, this paper presents the *PEPSC architecture* – an architecture customized for the domain of data parallel scientific applications where power-efficiency is the central focus. PEPSC utilizes a combination of a two-dimensional single-instruction multiple-data (SIMD) datapath, an intelligent dynamic prefetching mechanism, and a configurable SIMD control approach to increase execution efficiency over conventional GPUs. A single PEPSC core has a peak performance of 120 GFLOPs while consuming 2W of power when executing modern scientific applications, which represents an increase in computation efficiency of more than 10X over existing GPUs.**

***Keywords*-Low power, SIMD, GPGPU, Throughput computing, Scientific computing**

I. Introduction

Scientists have traditionally relied on large-scale supercomputers to deliver the computational horsepower to solve their problems. This landscape is rapidly changing as relatively cheap computer systems that deliver supercomputer-level performance can be assembled from commodity multicore chips available from Intel, AMD, and Nvidia. For example, the Intel Xeon X7560, which uses the Nehalem microarchitecture, has a peak performance of 144 GFLOPs (8 cores, each with a 4-wide SSE unit, running at 2.266 GHz) with a total power dissipation of 130 Watts. The AMD Radeon 6870 graphics processing unit (GPU) can deliver a peak performance of nearly 2 TFLOPs (960 stream processor cores running at 850 MHz) with a total power dissipation of 256 Watts. For some applications, including medical imaging, electronic design automation, physics simulations, and stock pricing models, GPUs present a more attractive option in terms of performance, with speedups of up to 300X over conventional x86 processors (CPUs) [12], [13], [20], [17], [15]. However, these speedups are not universal as they depend heavily on both the nature of the application as well as the performance optimizations applied by the programmer [11]. But, due to their peak performance benefits, GPUs have emerged as the computing substrate of choice for many scientific applications.

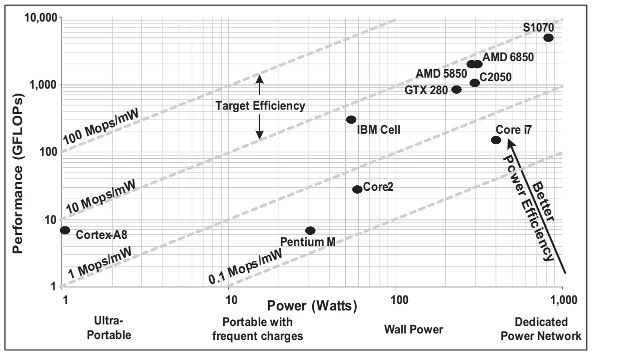


Fig. 1. Peak performance and power characteristics of several high-performance commercial CPUs and GPUs are provided: ARM Cortex-A8, Intel Pentium M, Core 2, and Core i7; IBM Cell; Nvidia GTX 280, Tesla S1070, and Tesla C2050; and AMD/ATI Radeon 5850 and 6850.

A natural question is, "what is the proper substrate for scientific computing – CPUs or GPUs?" This paper takes the position that the answer to this question is *neither*. CPUs are more focused on scalar program performance and do not have sufficient raw floating-point computation resources. Conversely, GPUs suffer from two major problems that limit scalability as well as the ability to deliver the promised throughputs for a wide range of applications: high power consumption and long memory access latencies. Rather, a new solution is required that offers high raw performance, power efficiency, and a tenable memory system.

Though power is not necessarily a significant drawback for video game graphics acceleration, requiring powerful cooling systems is a significant impediment for more portable platforms. Some of the algorithms that are commonly accelerated by GPUs are often deployed in systems where portability or power consumption is a critical issue. For instance, polynomial multiplication is used in advanced cryptographic systems, real-time FFT solving is required for complex GPS receivers, and low-density parity-check error correcting codes are used in WiMAX and WiFi. Monte Carlo Recycling or options-pricing algorithms for computational finance, are often deployed in dense urban areas where, while portability is not an issue, power and cooling certainly is an important cost concern. Even if the power consumption of a single GPU is not a concern, combining many of these chips to produce higher performance systems is untenable beyond a modest number (e.g., 1,000 Nvidia GTX 280s to create a petaFLOP system would require 200 kW).

To understand the trade-offs more clearly, Figure 1 presents performance-vs-power trade-offs for a variety of CPUs and GPUs. The GTX280 consumes over 200W of power while achieving a peak performance of 933 GFLOPs, resulting in a relatively low power-efficiency of less than 4 Mops/mW. Other solutions that push performance higher, like the Tesla S1070, can consume over 1kW. Other general-purpose solutions from IBM, Intel, and ARM, while consuming significantly less power, have similar or worse performance-per-Watt efficiency. The Core 2 and Core i7, while consuming less power than the GPU solutions also fall behind in terms of peak performance even when using the single-instruction multiple-data (SIMD) SSE instructions, leading to an overall loss of efficiency.

To overcome the limitations of CPUs and GPUs, we take the approach of designing a processor customized for the scientific computing domain from the ground up. We focus on dense matrix scientific applications that are generally data parallel. The *PEPSC* processor is designed with three guiding principles: power efficiency, maximizing hardware utilization, and efficient handling of large memory latencies. Our goal is one TFLOP performance at a power level of tens of Watts at current technology nodes. As shown in Figure 1, this requires increasing the energy efficiency of modern CPU and GPU solutions by an order of magnitude to approximately 20-100 Mops/mW. One solution would be to develop ASICs or hardwired accelerators for common computations [22]. However, we believe this approach is orthogonal and rather focus on a fully programmable SIMD floating point datapath as the starting point of our design. While CPUs and GPUs have economies of scale that make the cost of their chips lower than PEPSC could ever achieve, the design of PEPSC is useful to understand the limitations of current GPU solutions and to provide microarchitectural ideas that can be incorporated into future CPUs or GPUs.

This paper offers the following contributions:

- An analysis of the performance/efficiency bottlenecks of current GPUs on dense matrix scientific applications (Section II).
- Three microarchitectual mechanisms to overcome GPU inefficiencies due to datapath execution, memory stalls, and control divergence (Sections III-A, III-B, and III-C, respectively):
  1) An optimized two-dimensional SIMD datapath design which leverages the power-efficiency of data-parallel architectures and aggressively fused floating-point units (FPUs).
  2) Dynamic degree prefetching to more efficiently hide large memory latency while not exacerbating the memory bandwidth requirements of the application.
  3) Divergence-folding integrated into the floating-point (FP) datapath to reduce the cost of control divergence in wide SIMD datapaths and exploit "branch-level" parallelism.
- An analysis of the performance and power-efficiency of the PEPSC architecture across a range of scientific applications.

II. Analysis of Scientific Applications on GPUs

GPUs are currently the preferred solution for scientific computing, but they have their own set of inefficiencies. In order to motivate an improved architecture, we first analyze the efficiency of GPUs on various scientific and numerical applications. While the specific domains that these applications belong to vary widely, the set of applications used here, encompassing several classes of the Berkeley "dwarf" taxonomy [1] is representative of non-graphics applications executed on GPUs.

A. Application Analysis

Ten benchmarks were analyzed. The source code for these applications is derived from a variety of sources, including the Nvidia CUDA software development kit, the GPGPU-SIM [2] benchmark suite, the Rodinia [3] benchmark suite, the Parboil benchmark suite, and the Nvidia CUDA Zone:

- binomialOptions (`binOpt`): The binomial option pricing method is a numerical method used for valuing stock options.
- BlackScholes (`black`): A pricing model used principally for European-style options that uses partial differential equations to calculate prices.
- Fast Fourier Transform (`fft`): A high-performance implementation of the discrete Fourier Transform, used for converting a function from the time domain to the frequency domain.
- Fast Walsh Transform (`fwt`): The matrix product of a square set of data and a matrix of basis vectors that are used for signal/image processing and image compression.
- Laplace Transform (`lps`): An integral transform for solving differential and integral equations.
- LU Decomposition (`lu`): A matrix decomposition for solving linear equations or calculating determinants.
- Monte-Carlo (`mc`): A method used to value and analyze financial instruments by simulating various sources of uncertainty.
- Needleman-Wunsch (`nw`): A bioinformatics algorithm used to align protein and nucleotide sequences.
- Stochastic Different Equation Solver (`sde`): Numerical integration of stochastic differential equations.
- Speckle-Reducing Anisotropic Diffusion (`srad`): A diffusion algorithm based on partial differential equations that is used for removing the speckles in an image for ultrasonic and radar imaging.

The computation in these benchmarks is predominantly FP arithmetic. Other than that, the instruction breakdown of these benchmarks varies widely. Some benchmarks that access a number of different arrays have a higher number of integer arithmetic operations in order to perform address calculations. While most control-flow instructions in these applications are used for checking loop terminating conditions, a few benchmarks are quite control-flow intensive. These applications are all comprised primarily of parallelizable loops that will run efficiently on GPU-style architectures.

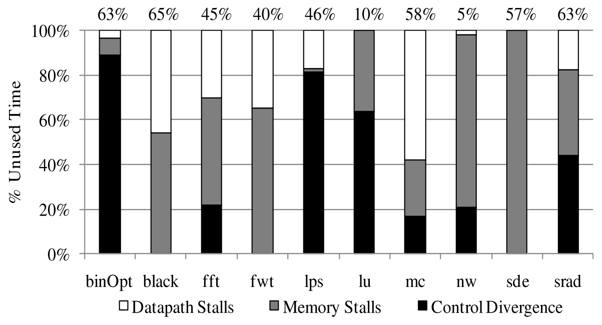


Fig. 2. Benchmark utilization on an Nvidia GTX 285 model. Utilization is measured as a percentage of peak FLOPs and is indicated as a number above each bar. The components of the bar represent a different source of stall cycles in the GPU. The mean utilization is 45%

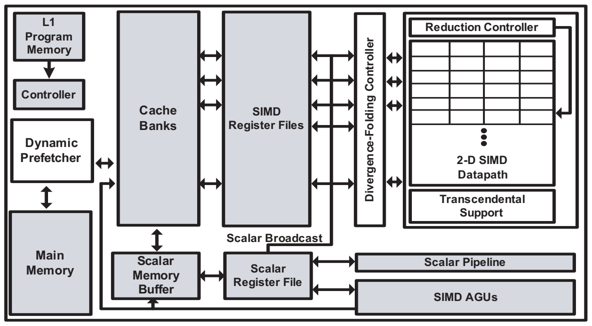


Fig. 3. PEPSC architecture template. The shaded components are part of conventional SIMD datapaths.

B. GPU Utilization

We analyze the benchmarks' behavior on GPUs using the GPGPU-SIM simulator [2]. The configuration used closely matches the Nvidia FX5800 configuration used in [2] and provided by their simulator. Modifications were made to make the simulated design very similar to the GTX 285 GPU, the most recent GPU that uses the FX5800's microarchitecture.

Figure 2 illustrates the performance of each of our benchmarks and the sources of underutilization. "Utilization" here is the percentage of the theoretical peak performance of the simulated architecture actually achieved by each of these benchmarks. Idle times between kernel executions when data was being transferred between the GPU and CPU were not considered.

On average, these applications make use of around 45% of the GPU compute power, with the extremes being BlackScholes, with a 65% utilization, and Needleman-Wunsch, with a 5% utilization. It is important to note here that the utilizations of individual benchmarks vary widely. Further, the extent to which specific causes lead to underutilization also varies from one application to another.

Figure 2 illustrates three principal reasons for underutilization in GPUs:

- Datapath stalls: This portion of the graph indicates the amount of time the shader core datapath itself is stalled for reasons such as read-after-write hazards. This is especially of concern on GPUs, which tend to have very deep floating-point pipelines.
- Memory stalls: The benefit from having numerous thread contexts in GPUs is the ability to hide memory latency by issuing a different thread-warp from the same block of instructions when one warp is waiting on memory. This is not always enough, however, and this portion of the graph indicates the amount of time that all available warps are waiting for data from memory. To check whether it is memory bandwidth or memory latency that is the limiting factor, a separate study was done using a machine configuration that had double the bandwidth of the GTX 285. The change in utilization was negligible.
- Serialization: The CUDA system collects 32 individual threads into a "warp" and executes them in a manner similar to a 32-wide SIMD machine. In sequential code, threads in a warp all execute the same series of instructions. However, in the event that some threads take a different execution path, the warp is split into the taken and not-taken portions and these two newly-formed warps are executed back-to-back rather than concurrently, reducing the overall utilization of the processor. The "control divergence" portion of the graph indicates the amount of time that is spent executing fractions of a warp rather than an entire warp at a given time.

Although a newer generation of Nvidia GPUs than the GTX 285 has been released, the major changes made to design – such as allowing multiple kernels to execute concurrently – do not affect the observations made in this analysis.

III. The PEPSC Architecture

An overview of the PEPSC architecture is presented in Figure 3. It has the following architectural components to fulfill the basic computational requirements for data-parallel scientific applications:

1) A wide SIMD machine to effectively exploit data-level parallelism.
2) A scalar pipeline for non-SIMD operations such as non-kernel code and incrementing loop-induction variables.
3) A dedicated address generation unit (AGU).
4) Special function units for math library functions such as sine, cosine and divide.

PEPSC employs several additional techniques to improve the efficiency of scientific computing, each addressing a different source of the current reduced utilization. These are:

- A two-dimensional design that extracts power efficiency from both the width and the depth of a SIMD datapath.
- Fine-grain control of the SIMD datapath to mitigate the cost of control divergence.
- A dynamically adjusting prefetcher to mitigate memory latency.
- An integrated reduction floating-point adder tree for fast, parallel accumulation with low hardware overhead.

These features are explained in more detail in the following sections.

A. Two-Dimensional SIMD Datapath

The first dimension of any SIMD datapath is the number of SIMD lanes. The optimal number of lanes in a domain-specific architecture is generally decided by the amount of data parallelism available in the domain. In massively-parallel scientific computing applications, however, the prevalence of "DOALL" loops allows for an effectively infinite amount of

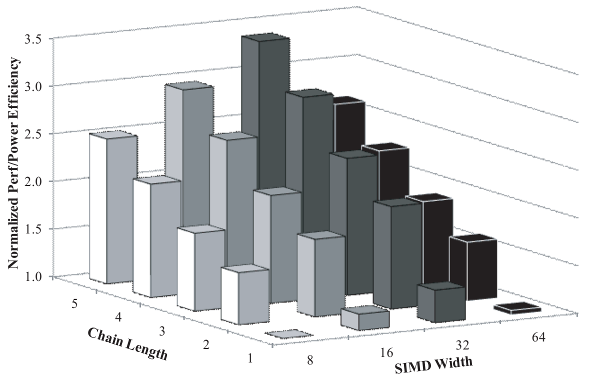


Fig. 4. Effect of length and width on the power efficiency of a 2D SIMD datapath, normalized to the 8-wide, 1-long case.

data parallelism. The deciding metric in such situations is the change in power efficiency of a SIMD datapath with increasing width due to factors such as control and memory divergence.

The PEPSC datapath introduces a second dimension with an operation chaining technique. The basic concept of operation chaining to efficiently execute back-to-back, dependent operations dates back to the Cray vector architectures. The architecture had separate pipelines for different opcodes allowing, for example, forwarding the result of an add instruction to an adjacent multiply unit to execute an add-multiply operation. In older, memory-to-memory datapaths with few registers, chaining was seen as an effective method to eliminate the need to write a value to a high-latency main memory only to read it back again.

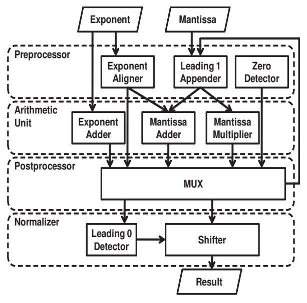
The PEPSC architecture uses a *different* style of chaining. It allows for several back-to-back, dependent floating-point operations to be executed on a novel, deeply-pipelined fusion of multiple full-function FPUs. This results in performance improvement due to fewer read-after-write stalls and power savings from fewer accesses to the register file (RF).

Figure 4 illustrates the power efficiency of varying SIMD widths between 8 lanes and 64 lanes when compounded on the power efficiency of 1- to 5-op FPU chaining. The efficiency is normalized to that of an 8-wide, 1-op FPU design, averaged across all benchmarks. These results indicate that a SIMD width of 32 lanes using a 5-op-deep chained FPU provides an optimal point, balancing the increasing efficiency of executing more operations per instruction with efficiency-reducing divergences. This 32x5 SIMD datapath has 3.4X the power efficiency of the 8x1 datapath.

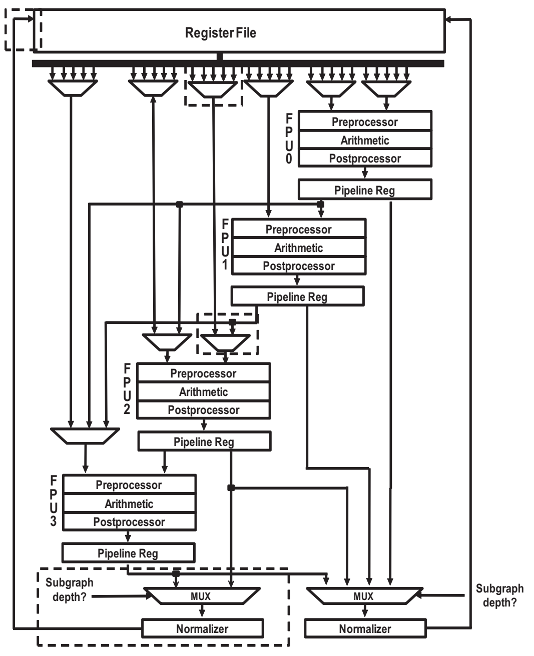
The remainder of this section explores the FPU microarchitecture in more detail.

1) Reducing FPU Latency: Scientific applications typically have chains of dependent FP operations longer than two operations. In the benchmarks studied in this work, nearly all have instances of 3 back-to-back floating point operations and some have instances of 5 back-to-back operations.

This work introduces generalized chains of FPUs. These chains allow for back-to-back execution of several floating-point operations, improving performance/power efficiency by eliminating redundant hardware and reducing accesses to the RF. All of the individual FPUs in the chained FPU retain full functionality and are able to execute any of the FP operations as a non-chained FPU. In order to achieve this, considerably



(a)



(b)

Fig. 5. (a) FPU internal structure. (b) Chained FPU design where the Normalizer stage may be removed for all but the last in a chain of FPUs

more hardware modifications have to be made to the FPU than what is traditionally done for a simple fused multiply-add/subtract FPU commonly seen in DSPs or GPUs.

Conventional FPU Architecture: A typical FP adder consists of a zero detector, an exponent aligner, a mantissa adder, a normalizer, and an overflow/underflow detector. The exponent aligner in the FP adder aligns the mantissa for the two operands so as to use the same exponent to compute the addition operation. Meanwhile, a FP multiplier generally consists of a zero detector, an exponent adder, a mantissa multiplier, a normalizer, and an overflow/underflow detector.

Chained FPU Design: As depicted in Figure 5(a), the FPU implementation used in this work is divided into four conceptual stages: preprocessor, arithmetic unit, postprocessor and normalizer. Typically, all FPUs operate only on normalized FP numbers and this is enforced by the normalizer. In general terms, the earlier parts of the FPU pipeline consist of components that expand the IEEE standard-form input operands into intermediate representations suitable for the main arithmetic units, and the later parts of the FPU pipeline compact the results of the computation back into the IEEE representation.

When operations are performed back-to-back, the intermediate values are never committed to architected state and, as such, need not be represented in the standard form, saving time on the FPU critical path and reducing the required hardware.

When the normalizer takes the result from the postprocessor, it primarily detects leading zeroes and shifts them as necessary so as to conform to the IEEE-754 FP format. If multiple FPUs are chained together and the value computed in the postprocessor is only an intermediate value, and not one committed to architectural state, the normalizing step may be removed and the next stage in the FPU chain can treat this result as a denormalized value. The normalizer consumes a significant amount of computation time – close to 30% – so its removal results in marked performance improvements. More details about the achievable improvements are presented in Section IV.

Extra logic that is necessary to make chaining work properly includes the shifter placed in the postprocessor to process varying result widths, resolving temporary overflows, and detecting the true location of leading ones so that they may be correctly appended for the next stage of the computation. Control logic is also added to shorten the length of the datapath if an instruction with fewer than the maximum allowed number of operations is being executed. The conceptual operation of the chained design is illustrated in Figure 5(b).

Identifying FP chains: We use modifications made to the Trimaran [21] compiler to identify and select sequences of instructions for execution on the chained FPU. First, an abstract representation of the possible sequences of execution is created in a data-flow graph (DFG) form. In the DFG, nodes are used to represent each input, output and individual FPUs in the chain. Directed edges are used to represent all possible communication between these nodes. A greedy algorithm is then used to select the largest of these subgraphs that occur within the benchmark; e.g., a single, 4-long sequence of operations is preferred over two 2-long sequences.

The sub-graphs used in this work were sequences of dependent FP add, subtract, and multiply operations where intermediate values were not live-out of the DFG but were only consumed locally. The internal interconnection is illustrated in Figure 5(b).

Operating on floating-point values in this manner results in final answers that differ from those obtained by a fully IEEE 754-compliant FPU. Per our experiments, the results of the chained FPU datapath's execution diferred from those of a strictly IEEE 754-compliant one by less than 0.5%.

Chain coalescence: Using a 5-deep chained FP unit as suggested by Figure 4 will obviously result in under-utilization in some benchmarks. To combat this, a few hardware extensions to the chained FPU design allow for multiple independent 2- or 3-long FPU chains to execute in parallel on the FPU datapath. These extensions are indicated by dotted lines on Figure 5(b). To allow for this, the RF has to have an additional read port and an additional write port. The FPU datapath will require an additional MUX to allow for one of the FPUs to select between receiving its inputs from either a previous FPU or from the RF. A second normalizer stage will also have to be added since there will now be two two possible exit points from the chained datapath.

2) Hiding FPU Latency: While the total time taken to execute several back-to-back FP operations may be reduced using FPU chaining, it significantly increases the total pipeline depth and, consequently, the latency of any FP instruction. Traditional architectures use hardware multithreading to hide various sources of latency – control latency, computation latency, memory latency, etc. While hardware multithreading helps increase the overall performance, it has a few drawbacks. Firstly, the control when using multithreading is significantly more complicated as each thread has its own PC, machine status register, execution trace, etc. In addition, each thread must be presented with the same architectural state. This work takes a compiler-directed approach of hiding the long FPU pipeline latency by software pipelining the inner-most loops [14] and overlapping independent successive iterations. When using software pipelining, since more data is being processed in parallel, the RF must be increased in size to provide enough data for the required computation. Instantaneous power will also increase due to the increase in operations at any given time, but the overall energy of computation will decrease since the processor is spending more time doing useful work rather than idling and waiting to issue a new instruction.

B. Reducing Memory Stalls

There are a few different alternatives when trying to mitigate problems with off-chip memory latency. Large caches offer a dense, lower-power alternative to register contexts to store the data required in future iterations of the program kernel. Even though modern GPUs have very large caches, these are often in the form of graphics-specific texture caches, and not easily used for other applications. Further, many scientific computing benchmarks access data in a streaming manner – values that are loaded are located in contiguous, or fixed-offset, memory locations and computed results are also stored in contiguous locations and are rarely ever reused. This allows for creating a memory system that can easily predict what data is required when. Some GPUs have small, fast shared memory structures but they are generally software-managed and, as such, it is difficult to accurately place data in them exactly when it is required.

1) Stride Prefetcher: A conventional stride prefetcher [4], [7], [6], [18], [26] consists of the "prefetch table" – a table to store the miss address of a load instruction, the confidence of prefetching, and the access stride. The program counter value (PC) of the load instruction is used as a unique identifier to index into the prefetch table.

2) Dynamic Degree Prefetcher: Stride prefetchers often have a notion of degree associated with them, indicating how early data should be prefetched. In cyclic code, it is the difference between the current loop iteration number and the iteration number for which data is being prefetched. A traditional stride prefetcher uses a degree of one for all the entries in the prefetch table. With large loop bodies, degree-one prefetchers perform well as the time required for prefetching data is hidden by the time taken to execute one iteration of the loop. However, if the time taken to execute a single iteration of the loop is less than the time required for the prefetcher to get the next working data, the processor will stall.

Figure 6(a) shows the number of times all the loads of a particular degree are executed in one of our benchmarks, `lps`. In this figure, the degree of prefetching in different loops varies between 1 and 8. An experiment was conducted to determine

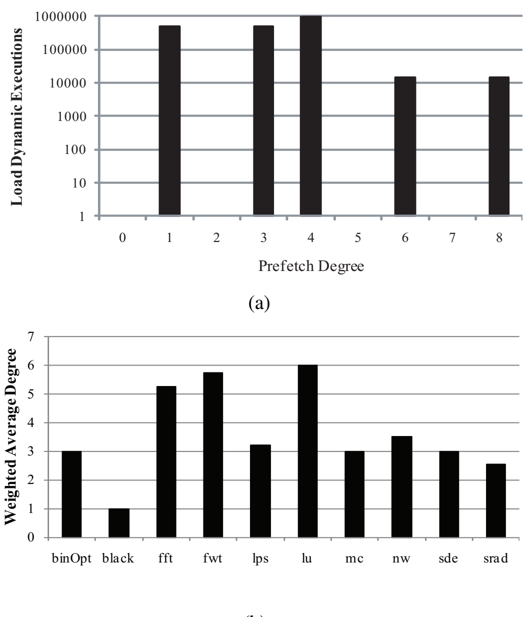


Fig. 6. (a) Varying prefetcher degrees in the `lps` benchmark (b) Varying weighted prefetcher degrees in different benchmarks

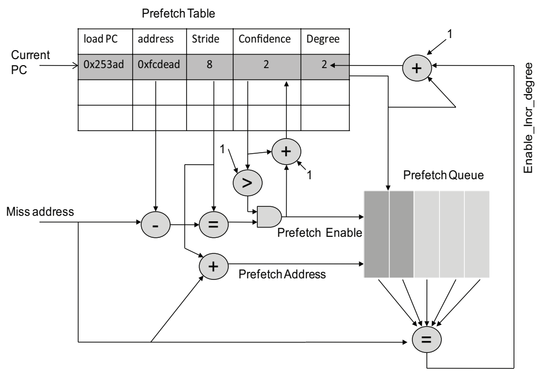
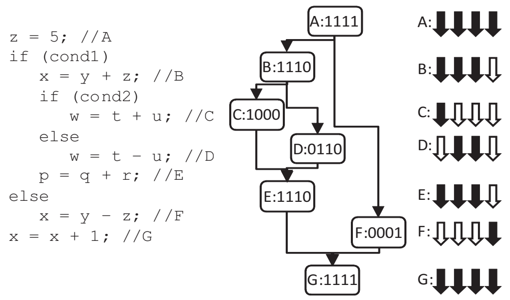


Fig. 7. Dynamic-degree prefetcher

the variance of degrees across all benchmarks as shown in Figure 6(b). This figure demonstrates that the prefetch degree in scientific applications is a characteristic of the benchmark itself. From Figures 6(a) and 6(b), it can be concluded that there is enough variance in degree within and across benchmarks that would offset the advantages of presetting a fixed degree. Having a lower degree leads to a performance penalty as the processor would have to stall for data. But if the degree is too large, more data will be fetched than is necessary, which leads to an over-utilization of bandwidth. Further, prefetched data may evict useful data. These factors indicate a dynamic solution is preferred. Loop-unrolling can decrease the degree of a load instruction and different amount of unrolling can be applied to the loops to change their degree to a fixed number. However, the performance benefits of this technique would vary between systems with different memory latencies. Loop unrolling also increases code size.

This work proposes a dynamic degree prefetcher (DDP) that varies the prefetch degree based on application behavior; higher degrees are assigned to cyclic code with shorter iteration lengths and lower degrees are assigned to code with longer iteration lengths. The computation of the degree is done at run-time, based on the miss patterns of the application. The DDP is illustrated in Figure 7. In the DDP, the initial degree of prefetching is set to 1. When an address is requested either by a load instruction or by a prefetch request, the prefetch queue is first checked to see if a prefetch request for data at that address has already been made. If so, it indicates that the prefetching for that load instruction is short-sighted. In response to this, the degree for the corresponding PC is incremented. By increasing the degree of the prefetcher, data is then prefetched from further ahead and, hence, by the time later iterations are executed, the required data will be present in the cache. As the degree is not fixed but dependent on the characteristics of the loop itself, different loops settle at different degrees.



```
z = 5; //A
if (cond1)
    x = y + z; //B
    if (cond2)
        w = t + u; //C
    else
        w = t - u; //D
    p = q + r; //E
else
    x = y - z; //F
x = x + 1; //G
```

Fig. 8. An example diverging code fragment and the associated control-flow graph. The black and white arrows indicate which SIMD lanes are used and unused, respectively, when using an immediate-post-dominator-reconvergence strategy.

C. Reducing Control Divergence and Serialization

SIMD architectures are normally programmed by explicitly specifying SIMD operations in high-level languages like C or C++ using intrinsic operations. In order to specify that only specific lanes in a SIMD operation should commit, a programmer has to express this explicitly using SIMD mask registers. Many SIMD architectures employ a “write mask” to specify whether or not a given lane commits its value. This write mask is specified as an additional operand. Destination registers for individual lanes are only overwritten by the values computed in the corresponding lane if the write mask for that lane is a ‘1’. While this gives the programmer control over the exact implementation of the code, handling such a low-level issue can become tedious.

The CUDA environment circumvents the issue of divergence in SIMD architectures by abstracting out the underlying SIMD hardware and allowing a programmer to essentially write scalar code and have the hardware assemble threads of scalar code into SIMD code. However, this now means that the hardware has to handle all control divergences. Figure 8 shows an example of divergent code. Divergence in Nvidia shader cores is handled using a technique called “immediate post-dominator reconvergence”, shown in Figure 8 which illustrates the way that post-dominator reconvergence works for a simple 4-wide SIMD [8]. Here, different lanes execute code on the “then” and “else” paths. The execution of the “then” and “else” basic blocks is serialized, and no computation is done on the unused lanes for each path, as shown by the white arrows, reducing the overall utilization of the datapath.

The AMD/ATI shader core architecture differs from the Nvidia one in its ability to extract intra-thread instruction-level parallelism using 5-wide VLIW "thread processors". These thread processors are in-turn clustered into 16-wide SIMD arrays with each element executing the same VLIW instruction word at any given time so this configuration, too, is vulnerable to reduced efficiency from divergence.

Solutions such as dynamic warp formation (DWF) [8] address the general problem of control divergence in SIMD architectures by grouping together CUDA threads that execute the same side of a branch. Such techniques, while effective at reducing divergence, are often limited by other effects; migrated threads must still execute in the same SIMD lane from which they originate, and mixing-and-matching threads from different warps reduces memory coalescence, reducing the effectiveness of the memory system.

1) Divergence-Folding: All lanes in a non-multithreaded SIMD architecture like PEPSC must apply the traditional technique of SIMD predicate masks and execute both sides of a branch for every lane. The per-lane masks are then used to decide whether the "then" or the "else" path of the branch for a given lane is finally committed to the RF. The performance penalty incurred by executing both sides of a branch due to the SIMD nature of the architecture can be reduced by the effective mapping of instructions on PEPSC's chained datapath.

Operations with complementary predicates can be executed simultaneously if two FUs are available. The 2D SIMD datapath allows such execution as there are multiple FPUs in each lane. The modified, dual-output chained FPU has sufficient register inputs and outputs to execute and commit two FPU chains concurrently. Therefore, independent FP operations on opposite sides of a branch can also be executed concurrently. A few modifications are made to the infrastructure to support this. First, the compiler is modified to allow FPU operations guarded by complementary predicates to be executed on the FPU datapath at the same time. Second, the control logic for the output selector MUXes of the FPU datapath is modified to select which of the two concurrently executing subgraphs should write their values to the RF based on a given SIMD lane's predicate mask bit. Since support for writing to two different registers already exists, this technique still works even if the two sides of a branch are writing to different registers.

There are a few caveats to this technique – the "then" and "else" paths should both be comprised of floating-point operations and the operation chain length should be limited to 2- or 3-long chains for these paths at most in order for both sides to fit on the FPU datapath.

Where the traditional predicate mask technique would require sequential execution of both sides of the branch, this divergence-folding technique exploits "branch-level" parallelism and executes both sides of the branch in parallel.

2) Reduction Tree: Another source of serialization are "reduction" operations such as an accumulation of all the values computed so far. Since such operations only involve addition to a single value, it is inherently serial and, as such, not SIMD-izable via a traditional SIMD machine.

In order to minimize the amount of time spent performing these serial reduction operations, an "adder tree" can be used to sum up all the values in a SIMD register in logarithmic time. For a 32-wide SIMD machine, this requires the use of a total of 31 floating-point adders, arranged 5 adders deep.

Normally, the hardware cost of such an addition would be quite substantial – on the order of doubling the number of FUs. However, due to the FPU chaining techniques presented in Section III-A1, this hardware cost can be reduced. Essentially, adding some interconnect logic to half of the two-dimensional datapath creates an FPU tree. For a SIMD width of n, the first level of the FPU chain (FPU0 in Figure 5(b)) sums together $\frac{n}{2}$ pairs of adjacent elements, the second level (FPU1) sums together the resulting $\frac{n}{4}$ pairs, etc. until the last element adds one pair. For a 5-deep FPU chain, this technique can only be used to add together 2^5, or 32, values.

IV. Results

The major components of PEPSC were designed in Verilog and synthesized using the Synopsys Design Compiler and Physical Compiler tools. Power results were obtained via VCS and Primetime-PX, assuming 100% utilization. Power characteristics for regular memory structures like dual-ported RFs and caches were obtained through an Artisan memory compiler while RFs with more than 2 read ports were designed in Verilog and synthesized.

The M5 simulator system was used to study the cache design and the DDP. The main memory latency was set at 200 cycles. The L2 size was 4kB per SIMD lane with 4-way associativity and a delay of 20 cycles. The L1 size was 512B per SIMD lane with 4-way associativity and a delay of 1 cycle.

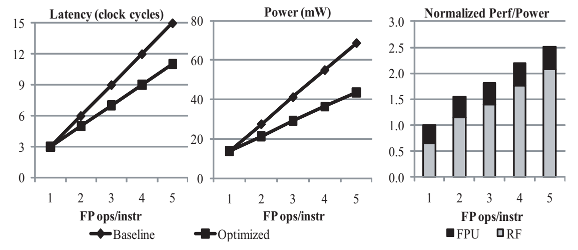
Sections IV-A, IV-B and IV-C present speedup information for the specific problem solved in isolation. Combined results showing the aggregate affect of all the modifications are presented in Sections IV-D and IV-E.

A. Datapath optimizations

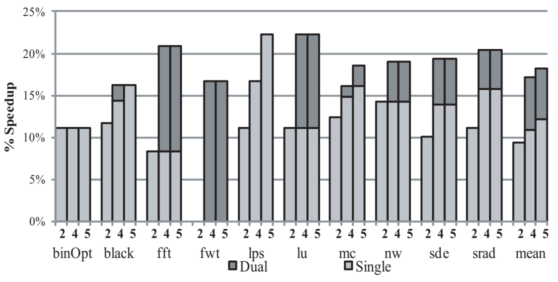
Figure 4 indicates that a number of the applications in this domain have several long sequences of back-to-back FP operations. Based on this data, the FP datapath in PEPSC was designed with an FPU consisting of 5 back-to-back operations. Figure 9 shows the effects of varying the number of operations in the chained FPU.

In Figure 9(a), the x-axis for all the graphs, "FP ops/instruction" is the number of successive, dependent FP operations executed in the chained FPU. The "latency" graph shows the time (in clock cycles) taken to execute an input subgraph. The baseline 3-cycle FPU takes 3 cycles for each operation and thus has a latency that increases by 3 cycles for every added operation. The removal of redundant hardware in the optimized FPU chain results in significantly less overall latency – a savings of 4 cycles when 5 FPUs are connected back-to-back.

The "power" graph in Figure 9(a) illustrates the power savings obtained from optimized normalizing. Here, the baseline is multiple un-modified FPUs executing operations back-to-back. In this graph, too, the gap between optimized and unoptimized FPUs widens quite dramatically as the number of operations per instruction increases. The power measurement in this graph is the sum of the RF access power and the



(a)



(b)

Fig. 9. (a) Datapath latency, power and efficiency effects of varying the number of FPUs when increasing FPU chain length. (b) Speedup with increasing chain length. The chain lengths are indicated in bold numbers below each bar; the dark portion, “Dual”, above the “4” and “5” bars indicate the additional performance improvement from allowing multiple subgraphs to execute concurrently.

FPU execution power to better reflect the power penalty from increasing the number of RF read-ports.

The “normalized perf/power” graph in Figure 9(a) addresses the efficiency of the different solutions. Here, too, the power consumed for the amount of work done steadily reduces as the number of FP operations per instruction increases. While the access power for an RF increases for every added read port, the improvement in the efficiency of the RF shown in the graph indicates that this is amortized by the reduction in overall accesses and by the performance improvement achieved by chaining together FPUs.

Figure 9(b) shows the reduction of stall cycles observed for the different benchmarks when using 2-, 4-, and 5-operation chained FPU datapaths. The speedup varies based on how frequently the various chained FP operations occur in each benchmark and the latency penalty incurred when issuing back-to-back dependent FP operations. The benchmarks that had the most to gain from chaining FPUs were, in general, the ones with the most number of 5-long chains of FP operations – `black`, and `lps`, for example. The `fwt` benchmark had no operation patterns that could be accelerated using the chained FPU and, therefore, had no performance improvement. The “dual” bars indicate the additional performance improvement from allowing multiple independent subgraphs from executing on the FPU datapath. Significant performance improvement is observed in the majority of benchmarks, the most notable being `fwt` which previously could not exploit the improved datapath design. On average, the speedup for a 5-long chain increased from 12% to 18%.

Speedup saturates at 5 operations and adding a sixth operation in the chain only reduces the overall utilization of the FPU. Using a 5-op chain is, therefore, the best solution, from both a performance and an efficiency perspective.

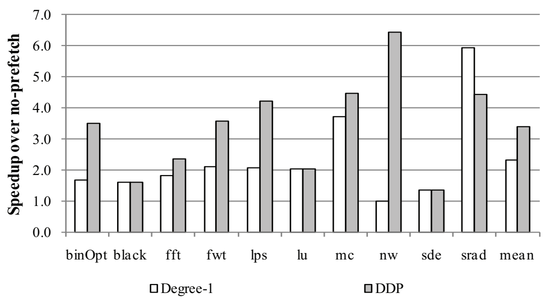


Fig. 10. Comparison of DDP and degree-1 prefetching.

B. Memory System

To ascertain the effectiveness of our prefetching techniques, we measured the performance of a baseline no-prefetcher system, the performance with an added degree-1 prefetcher and the performance with the DDP. These results are shown in Figure 10. The baseline degree-1 provides, on average, a speedup of 2.3X, resultant from a 62.2% reduction of average D-cache miss latency. The more sophisticated and adaptive DDP shown in Figure 7 provides a speedup of 3.4X on average, due to an additional 53.1% reduction in average D-cache miss latency over the degree-1 prefetcher. This amounts to a speedup of 1.5X speedup over the traditional degree-1 prefetcher. The `srad` benchmark presents a pathological case; the main inner-loop has 5 separate, equally-spaced loads from the same array, leading to higher-than-expected conflict and pollution in one particular set in our 4-way associative cache. To confirm this, we repeated the experiment but with an 8kB, 8-way associative L2; for this configuration, the performance of the DDP was better than that of the degree-1 prefetcher.

The memory system used for the results in Figure 10 prefetched data from main memory into the L2 cache, but not from the L2 cache into the L1 cache. While there was a performance improvement in some benchmarks by the addition of a prefetcher from the L2 cache to the L1 cache, this was offset by the performance loss observed in others, resulting in a negligible average performance difference between the two cases. The performance reduction observed was primarily caused by the pollution of the L1 cache by data prefetched too soon, resulting in the eviction of needed data such as local variables on the stack.

A common concern when using prefetchers is the transfer of unnecessary data. The total amount of data transferred from main memory to L2 was analyzed for the no-prefetcher, degree-1 and dynamic degree prefetchers. The degree-1 prefetcher transferred, on average, only 0.8% more data than the no-prefetcher case, and DDP transferred 1.3% more than the no-prefetcher case.

C. Control Divergence and Serialization

Four of the benchmarks studied demonstrated speedup using the control-flow techniques discussed in Section III-C: `lps`, `lu`, `mc` and `srad`. The source of the improvement between them varies.

The benchmark `lps`, `lu` and `srad` benefit primarily from the use of the divergence-folding technique. All these benchmarks are control-flow intensive, and have predominantly symmetrical control-flow graphs, performing similar, short

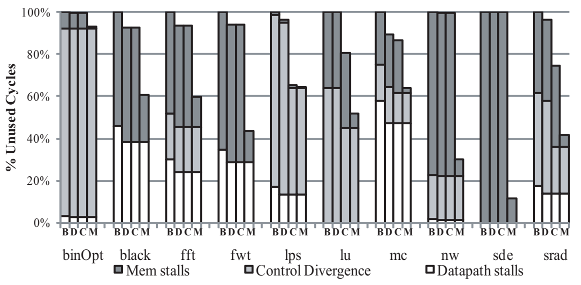


Fig. 11. Reduction in GPU overheads after cumulatively applying various techniques. "B" is the baseline bar, "D" is after adding the chained FPU datapath, "C" is after adding divergence-folding to mitigate control overhead and "M" is after adding the prefetcher to reduce memory latency.

computation on either side of a divergence – the perfect use-case for the technique. These benchmarks saw reductions of 38%, 30% and 50% in control-flow overhead, respectively.

The `mc` benchmark employs reduction operations at the end of its loops which are not amenable to SIMD-ization. Making use of the reduction adder tree proposed in Section III-C2 rather than serially summing values provides a 16% reduction of control-flow overhead.

D. Application to GPU

Rather than creating an entirely new architecture, another option would be to augment existing GPUs with the hardware introduced in this work. Our estimates for the impact this would have are illustrated in Figure 11. The first bar for each benchmark is the baseline underutilization breakdown from Figure 2 and each subsequent bar is the revised breakdown with cumulatively adding the FPU chaining, divergence folding and a reduction tree, and the dynamic-degree prefetcher. This average utilization increases to around 73% – a significant improvement over the 45% utilization baseline.

E. PEPSC Specifications

The PEPSC processor's design characteristics and power consumption breakdown are shown in Figure 12(a). The top table in Figure 12(a) shows the specifications of each individual core. The efficiency of 56.9 Mops/mW is approximately 10X that of the Nvidia GTX 285 and over 10X that of the Nvidia GTX 280. The bottom table in Figure 12(a) shows a component-by-component breakdown of the power consumed in the PEPSC processor. The FPU power includes the power consumed by the additional control logic required by the op-selection scheme presented in Section III-C1 and also the extra routing and FP adder required by the reduction tree presented in Section III-C2. Since there are components to this FPU datapath that go across lanes, the per-lane power consumption number is an approximation. The "Memory system" power includes the power consumed by the two levels of caches and the dynamic-degree prefetching engine.

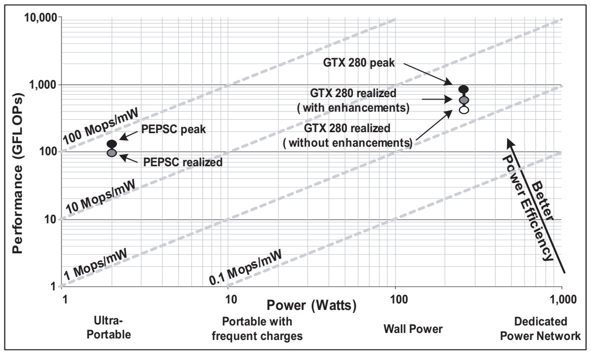
Figure 12(b) is a modified version of Figure 1. It illustrates the power consumption and performance of PEPSC relative to other current designs. The white dot under the GTX 280 dot is the average realized performance for the baseline design. The gray dots below the PEPSC and GTX 280 dots indicated the average realized performance, as opposed to the peak performance.

For both peak and average use-cases, the PEPSC design is over 10X as power-efficient as modern GPUs.

Frequency	750 MHz
SIMD Lanes	32
Peak Performance	120 GFLOPs
Peak Total Power	2.10W
Efficiency	56.9 Mops/mW
Technology Node	45nm

Component	**Power**
5-op 32-bit FPU with reduction + divergence-folding	1.18W (∼37mW/ln)
16-element 32-bit RF	9.28 mW/ln.
Memory system	580mW
Etc. datapath (scalar pipe, AGU, control)	59 mW

(a)



(b)

Fig. 12. PEPSC specifications. (a) Overall per-core specifications and power breakdown of individual components. (b) **(Modified Fig. 1)** Power-efficiency of PEPSC. Black points are peak performances, and the gray and white points represent different points of underutilization.

V. Related Work

There are several other examples of low-power, high-throughput SIMD architectures like SODA [23] for signal-processing but it, being a purely integer architecture, is unsuitable for this domain space. VTA [10] has a control processor with a vector of virtual processors, which are far more complex and power-consuming than the SIMD lanes of PEPSC. There are also high-throughput floating point architectures such as Merrimac [5], TRIPS [16] and RAW [19] but these are more focused on general-purpose computing and do not have the same power efficiency as PEPSC. The streaming buffers in Merrimac are a potential alternative to the memory system employed in this paper, but are less general than using a cache hierarchy. There has also been recent work on image-processing [9] and physics simulations [24], targeting domains traditionally addressed by commercial GPUs but these, too, do not address power to the extent that this work does. Parallax [24] is a heterogenous domain-specific architecture for real-time physics with coarse-grain and fine-grain cores coupled together to distribute the workload to achieve high performance.

Some Cray systems use a "chained FPU" design. However, these are essentially just forwarding paths across different FUs. While this connectivity reduces register accesses, the FPU itself was not redesigned the way the PEPSC FPU is. In [25], the authors create a compound functional unit used to accelerate a variety of different applications but do not optimize the functional unit by removing overlapping hardware the way that this work's FPU chaining implementation does; further, it provides a solution that is more amenable for use as a co-processor rather than a within-datapath FU.

An alternative technique to mitigate the serialization problem of modern GPUs was presented in [8]. This "dynamic warp formation" technique collects warps of SIMT threads in Nvidia GPUs that branch in the same direction. This technique is quite effective but is limited in that the relative position of a thread within a warp can never change in order to maintain program correctness.

Hardware based stride prefetchers have been in existence for some time [4], [7]. Even though the current trend in data prefetching [6] is to look for correlation in data accesses, we feel that a simple strategy is more effective for scientific applications which have more regular access patterns. Variation of degree has been studied by [18]. They take periodic samples of various parameters of prefetching and change the aggressiveness of prefetching at the end of every sampling interval. Furthermore, they change the aggressiveness of the prefetcher in quanta, rather than reaching a general sweet spot. Increasing degree one at a time is more effective for scientific applications as the application execution times are quite high and the initial delay in ramping up the stride degree is only a small factor. In [18] they fix the aggressiveness of the prefetcher for a region which is effective for the SPECint benchmarks, whereas in the benchmarks that we have used, precise prefetching on a per-stream basis is possible because of the regularity in memory accesses. TAS [26] also increments the degree of prefetching but in that work interval state is computed from global time and on its basis the degree is increased in discrete preset quanta.

VI. Conclusion

The PEPSC architecture – a power-efficient architecture for scientific computing – is presented in this work. When running modern scientific, throughput-intensive applications, it demonstrates a significant improvement in performance/power efficiency over existing off-the-shelf processors. This architecture was designed by addressing specific sources of inefficiencies in the current state-of-the-art processors. Architectural improvements include an efficient chained-operation datapath to reduce register file accesses and computation latency, an intelligent data prefetching mechanism to mitigate memory access penalties, and a finely controllable SIMD datapath to exploit data-level parallelism while mitigating any control divergence penalty. The PEPSC processor provides a performance/power efficiency improvement of more than 10X over modern GPUs. While a single domain-specific design is presented here, many of the architectural techniques introduced are general enough to be deployed on current designs.

Acknowledgments

We thank the anonymous referees who provided excellent feedback. We also thank Gaurav Chadha and Wade Walker for providing feedback on this work. This research was supported by National Science Foundation grant CNS-0964478 and ARM Ltd.

References

[1] K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, Dec. 2006.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[4] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[5] W. Dally et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 35–42, 2003.

[6] P. Diaz and M. Cintra. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 81–92, 2009.

[7] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, 1992.

[8] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007.

[9] V. Govindaraju et al. Toward a multicore architecture for real-time ray-tracing. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 176–197, Dec. 2008.

[10] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, , and K. Asanovic. The vector-thread architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 52–63, 2004.

[11] 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 *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, 2010.

[12] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7, Apr. 2008.

[13] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa. Fine-grain parallelism using multi-core, Cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function. In *Proc. of the 2009 International Conference on Parallel Processing*, pages 9–17, 2009.

[14] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[15] D. Rivera, D. Schaa, M. Moffie, and D. Kaeli. Exploring novel parallelization technologies for 3-D imaging applications. In *Symposium on Computer Architecture and High Performance Computing*, pages 26–33, 2007.

[16] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[17] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2009.

[18] S. Srinath et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 63–74, Feb. 2007.

[19] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[20] P. Trancoso, D. Othonos, and A. Artemiou. Data parallel acceleration of decision support queries using Cell/BE and GPUs. In *2009 Symposium on Computing Frontiers*, pages 117–126, 2009.

[21] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[22] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–218, 2010.

[23] M. Woh et al. From SODA to scotch: The evolution of a wireless baseband processor. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 152–163, Nov. 2008.

[24] T. Yeh et al. ParallAX: an architecture for real-time physics. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 232–243, June 2007.

[25] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 277–288, 2009.

[26] H. Zhu, Y. Chen, and X.-H. Sun. Timing local streams: improving timeliness in data prefetching. In *Proc. of the 2010 International Conference on Supercomputing*, pages 169–178, 2010.