CS533 – Midterm Progress Report

Apollo Ellis Russell Jones Sean Bartell Kathleen Chalas

May 15, 2014

Abstract

We combine wide-SIMD, scatter-gather, and PIM architecture to analyze whether the combination could produce a performance increase over traditional systems. Understanding when to widen SIMD, how much it helps, and why are important questions in the PIM space where bandwidth is suddenly much higher and latency is drastically reduced. We take existing algorithms and port their kernels to different widths of data-parallel execution and load/store units. We discuss performance results when adjusting parameters in our system using two applications. We conclude that combining wide-SIMD and PIM is not worthwhile.

1 Introduction

In the landscape of parallel computing and architecture, multicore CPUs and GPUs dominate. However, these processors meet different needs. Many-core CPUs, such as the Intel Larrabee, showcase high performance on irregular applications[6], while graphics processors demonstrate performance gains on regular, dense data-parallel applications[4].

These CPU and GPU platforms still fail to meet the performance needs of some applications, so we investigate an alternative way to improve performance for such applications. We place a small in-order co-processor next to memory and equip it with very wide SIMD registers and ALUs. We further equip this processor-in-memory (PIM) with a scatter-gather unit which alleviates the problems of keeping all of the SIMD lanes on the same execution path, problems well-known in graphics processors.

Our system is intended to accelerate programs that have a large bandwidth requirement, too large even for GPU memory systems. The target programs should have ample independent data-parallelism, but we still support highly irregular memory access patterns. Our PIM facilitates data-parallelism while the proximity to the memory system improves the performance of latency-critical and bandwidth-bound applications. Furthermore, SIMD is well-understood by many software engineers, unlike exotic designs such as dataflow architectures.

Our target applications get limited acceleration on GPUs, despite the incredible speed of the ALUs, because the inability to coalesce memory accesses makes memory bandwidth into a bottleneck. Furthermore, while CPUs can handle divergent threads and arbitrarily complex control flow, they are not optimized for massively data-parallel

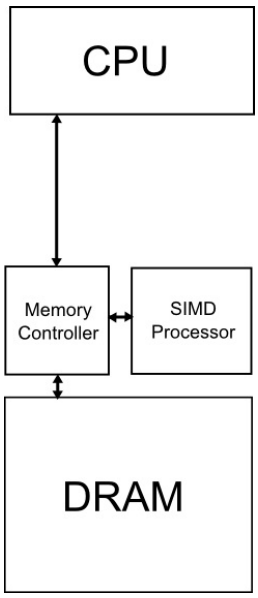

Figure 1: Our design.

codes or high bandwidth, limiting their performance in many cases. Certain types of code run up against both sets of limitations, and we expect our architecture to be much more efficient than CPUs or GPUs in these cases.

2 Related Work

Other projects have studied PIM architecture, wide SIMD, and scatter-gather capabilities in the interest of improving performance; however, they studied each element separately rather than fusing them together.

The Data-Intensive Architecture project (DIVA)[3] was closest in concept to our project, and it inspired some aspects of our design. PIM architectures are good for applications that require a high memory bandwidth to achieve good performance. Based on this, the researchers on the DIVA project created a design that combines PIM memories with some number of external host processors, with a PIM-to-PIM interconnect. Two mechanisms are primarily responsible for DIVA's increase in memory bandwidth. First, some amount of computation is selected to run on the PIMs, reducing the data that must transfer across the processor–memory interface. Second, in order to bypass the processor–memory bus, the PIMs communicate with each other over the interconnect. With these changes, DIVA improves speed even when running irregular applications, like sparse matrix and pointer-based computations, which are of particular interest to the high performance computing community. We noted that their test bed was composed of a sparse conjugate gradient computation, a natural-join database operation, and an object-oriented database query. They chose applications that would perform poorly on a traditional system.[2] Their overall goal with DIVA seems to have been the seamless integration of PIMs into traditional systems: irregular programs benefit from serious PIM computation when needed, but the PIM modules also work as a form of smart memory when intense computation is not necessary, for instance on regular, easily parallelizable programs that perform well on current processors.

Our system is greatly simplified and partially steered by a different goal. It only has one PIM module to which most if not all the work is offloaded. Thus,, we do not incur the complications or communication penalties of an interconnect. DIVA was meant to extend traditional systems while our architecture is intended to replace them.

SIMD requirements vary from application to application and even within an application. SIMD accelerators provide to be especially low powered and efficient in execution if the SIMD design is specialized to the application. AnySP[10] was a specialized system built to accelerate H.264 encoding, a highly important operation for smart phones and devices. In AnySP, the SIMD units are effectively 64-wide and all lanes operate in lock step; however, they can be dynamically partitioned into 8x8-wide, 4x16-wide, or 2x32-wide units, giving each H.264 kernel the best choice for efficiency. The lane partitions are actually 16-wide each, but glued together by a multi-SIMD shuffle network. The authors make various other improvements, such as adding a horizontal reduce add operation, and they fuse several commonly used instructions for further efficiency. The speed-ups are on the order of 3 times with power savings of 25%. Although our own system could support fusing instructions in the future, we do not intend to change SIMD width dynamically and we currently use a basic, common SIMD instruction set. Our processor is still more general-purpose and thus more programmable.

We also investigated prior work on scatter-gather operations, focusing in particular on the DistTree project[7]. DistTree sought to automate scatter and gather operations for applications like the Fast Fourier Transform that have predictable, yet complex, memory accesses. The paper gave us a means of phrasing how scatter-

	Main processor	**PIM coprocessor**
Frequency	3 GHz	500 MHz
Scheduling	Out-of-order	In-order
Architecture	32-bit ARM	32-bit ARM with wide SIMD
Cache	4 MB split L1; 16 MB L2	4 MB split L1
Memory Latency	18 ns	9 ns
Memory Bandwidth	24 GB/s	48 GB/s

Table 1: The processors in our proposed architecture.

gather would work in our own research. Their observation that most data-intensive programs follow a three step model of first loading old data, then computing new data in terms of the old data, and finally storing that new data helped us identify the application environment we would be working with in our data-intensive applications. Their observation that the first step is essentially a gather operation and the last step is a scatter operation indicated that we should pay attention to those targets when planning our implementation. The typical machine spends a significant amount of time computing the memory addresses prior to accessing data.

In DistTree, they have a scatter-gather system connecting the processor to memory and their performance gains were seen on applications that could predict their memory accesses before run time. Their binary-tree-like network was made to obviate the need for explicit memory addresses and run-time computations of addresses. In their paper, they shift the address calculation work to a special piece of hardware, allowing the processor to work on computation; on FFT, this resulted in a two-fold performance improvement. We kept their framing of the problem in mind when implementing our own scatter and gather operations.

3 Simulated Architecture

A modern Processor in Memory would be die stacked, and previous work has shown that die stacked DRAMs have more than double the bandwidth and approximately half the latency as off-chip DRAM[9]. However, die stacked chips have more thermal constraints than non-die-stacked chips, and therefore, the PIM coprocessor is in-order and clocked at a lower frequency. The PIM also only had an L1 cache, as a deep memory hierarchy would hide the benefits of being close to memory. The details of the processors are shown in the table.

Our wide SIMD architecture uses 16 SIMD registers, each with 64 64-bit elements. There are new SIMD units based on a standard set of ALU units.

3.1 ISA

The SIMD instruction set supported by the PIM is generally unremarkable except for the width of the vectors. Vectors of booleans, 32-bit integers, and 32-bit and 64-bit floating point values are supported, and the 64-bit elements are half as many in same vector. Addition, subtraction, multiplication, division, modulo, maximum, and less-than are supported with both vector–vector and vector–scalar operations. Square root, boolean "and", and element insert and extract are also supported.

Memory loads and stores are supported for both contiguous elements, using a start address, and scatter and gather operations, using an integer vector of addresses. To support control flow and uneven array sizes, loads and stores can be predicated using a length or a boolean vector. A few dedicated shuffle options are implemented, for instance to separate X, Y, and Z vectors where they are interleaved in memory.

4 Applications & Vectorization

To test whether our changes to the processor architecture make any significant improvements to performance, we ran benchmark applications representative of the kinds of workloads best suited for this architecture. PIM architectures have a reduced memory access latency because of the processor's proximity to memory. For SIMD, having larger lanes means there are more processing elements in the computer to perform a single instruction on a wide data set, thus requiring a great deal of data-level parallelism. In scatter-gather, the best workload also involves having a lot of data to read in uneven memory access patterns.

We selected the `fluidanimate` benchmark from the PARSEC suite and the `fft` benchmark from SPLASH-2X that is packaged with PARSEC 3.0[11]. They are trusted benchmarks used in countless other projects and they have remained relevant and representative of current workloads. They also meet the specifications of large bandwidth requirements, independent data-parallelism, and some irregularity in the access patterns with which to exploit the advantages of our architecture.

We vectorized the kernels of each application for our architecture in order to test its performance. Although we could more fully exploit our architecture by fundamentally redesigning the applications, we avoided such changes in order to ensure our comparison is fair, and left code outside the kernels almost untouched. After all, if we redesigned an application to make it faster on our architecture, it would be difficult to tell whether any improvements came from the architecture itself or merely our redesign.

4.1 FFT

The discrete Fourier transform (DFT) has several applications, but it is most often used in digital signal processing to convert a time domain signal into the frequency domain, enabling many analyses and transformations. In practice, the DFT and its inverse are implemented with a fast Fourier transform (FFT) algorithm, which effectively takes the DFT matrix and factors it into a product of sparse matrices.

We are interested in FFT because of its use in the Robust SIMD project[8], which investigated the possibilities of dynamically adapting SIMD widths, clearly relevant to our SIMD width investigation. We used the same implementation of FFT, which is taken from the Splash-2X benchmark suite[1] included in PARSEC 3.0. Although there are multiple ways to implement FFT algorithms, Splash-2X uses a version specifically meant for the signal processing problem domain and runs on millions of data points, which is certainly desirable for our data-heavy workload specification. FFT also has a predictable and complex access pattern for which scatter-gather may be beneficial[7].

The FFT implementation we used contains two kernels that together take up about 85% of execution time. The first, FFT1DOnce, performs a computation over a few sequential arrays. This kernel was easily vectorized for our PIM, and indeed it should be easily handled on any SIMD architecture. The second kernel transposes a dense matrix by reading every row and storing it in every column. This would not easily benefit from traditional SIMD, but it was straightforward to modify to use our wide load and wide scatter, and will hopefully benefit from the memory access speed on the PIM.

4.2 Fluidanimate

From the animation domain, we chose the `fluidanimate` program from PARSEC[11], which simulates the flow of a fluid that behaves according to the Navier–Stokes equations. Fluid simulation is used in a broad variety of applications, from

weather prediction to smoke or water animation to blood flow simulation. The benchmark uses a working set of five frames and 300,000 particles that are processed in some 14 billion instructions total. Because of its significant data usage and large problem size, as well as its relevance to physics simulations in computer games and other platforms in need of real-time animations, we chose to include it amongst the programs we will run on our processor.

There are two kernels that together consume about 85% of execution time. For each particle, each kernel iterates over every particle in the same logical cell or one of the 26 neighboring logical cells. The first kernel calculates the density at each particle, while the second calculates the total acceleration from forces on each particle. While vectorized versions of the calculations themselves were straightforward, they require vectors containing the addresses of each combination of particles, which must be built by serial code. That code is likely to be a bottleneck, so we experimented with running it either on the PIM, to reduce communication, and on the out-of-order core. The overhead could most likely be reduced by redesigning `fluidanimate`, but that wouldn't be a fair comparison.

5 Evaluation

We evaluated our proposed architecture using ESESC[5], a modern, accurate, configurable processor simulator. ESESC is built of Qemu, which emulates the processor to determine which instructions are executed and what memory they access, combined with simulation code, which determines the timing and resource usage of each instruction. We modified both the emulator and the simulator to support our wide SIMD instructions.

Rather than modify the core of the code to fully implement the instructions, we took a less invasive approach. The source-level intrinsic that represents one SIMD instruction is shown below. The emulator executes the loop in order to calculate the result of the instruction, ensuring control flow that depends on the result will be correct, but it hides those instructions from the simulator. Only the first special assembly opcode is sent to the simulator, so that it treats this entire function as a single wide SIMD instruction. In this way, we ensured accurate emulation and simulation without actually needing to implement our new instructions or registers in the emulator. There are also special instructions that tell the simulator when to switch between the out-of-order processor and the PIM.

```
void vfloat_sqrt(vfloat in, vfloat out)
{
    asm volatile(...); // PRAGMA
    for (int i = 0; i < SIMD_WIDTH; i++)
        out[i] = sqrtf(in[i]);
    asm volatile(...); // PRAGMA
}
```

5.1 Emulation

An overview of the Qemu emulator is necessary to understand our changes. Qemu analyzes the binary code under emulation and creates internal structures called translation blocks, which are effectively basic blocks of machine instructions. For each translation block interpreted, Qemu loops over the program counters in the block and extracts the machine language instruction. The machine instruction is passed into the ESESC simulator, along with associated data such as addresses, data values, and other necessary information.

After intense inspection of the Qemu code, we determined where to intercept the "pragmas," actually just unused ARM opcodes, corresponding to the operations in our ISA. Recall that some are used to delimit the offloaded PIM kernel and some delimit each intrinsic operation. First, we modified the disassemble function to ignore our opcodes. Once the blocks are being executed and their instructions are being passed to the simulator, we again check for our special opcodes. Between the start and end of an intrinsic, we have modified Qemu's execution function to stop sending instructions to the simulator, basically ignoring the remainder of the instructions that make up the intrinsic. We pass only the starting opcode to the simulator, which it will use to simulate the correct instruction.

When load and store intrinsics are encountered, our code also tracks the addresses in order to simulate cache miss and latency timing. This is accomplished as follows. When the initial opcode is detected, our code looks for another load opcode that accesses the first memory address of the vector. We pass this address into the simulator along with the intrinsic-starting opcode as usual. In the case of scatter and gather, we make the conservative assumption that they always miss in the cache and never pollute it, eliminating the need to send every address to the simulator.

Our modifications fit concisely and conservatively into the Qemu infrastructure. Our intrinsic-based design simplified the integration of our co-processor execution with the overall execution of programs in Qemu. While this modification was time consuming, the result is still cleaner and more complete than if we had tried to fully implement our special instructions and registers.

5.2 Simulation

After an instruction has been emulated by QEMU, it is passed to the simulator to do timing. At this point the instruction is intercepted and sent to the correct processor. The functional units in the PIM were adjusted to work at a lower frequency and wide SIMD units were added. The memory for the PIM was adjusted to have double the frequency and half the latency. Furthermore, the PIMs lower level caches were removed.

6 Results

We simulated each program in three configurations: without the PIM, with the existing serial code for the kernels run on the PIM, and with the vectorized kernels run on the PIM. The normalized CPI for `fft` and `fluidanimate` are shown in figures 2 and 3. In both cases, the PIM performs worse than the code running solely on the host processor. This is true regardless of whether the code was vectorized to gain benefit from the Wide SIMD units. This is primarily due to the slower frequency of the PIM. The PIM overall does not even gain in terms of memory accesses, since most of the memory accesses are cache hits, and the cache is clocked much slower for the PIM than for the host.

The compute and memory time both go down when each program is vectorized. The compute time goes down due to parallelization of the work. The memory time goes down because an entire cache line can be loaded with a single instruction. This reduces the cache latency, but does not affect the main memory latency. However, these benefits are not enough to make up for the slower frequency of the PIM.

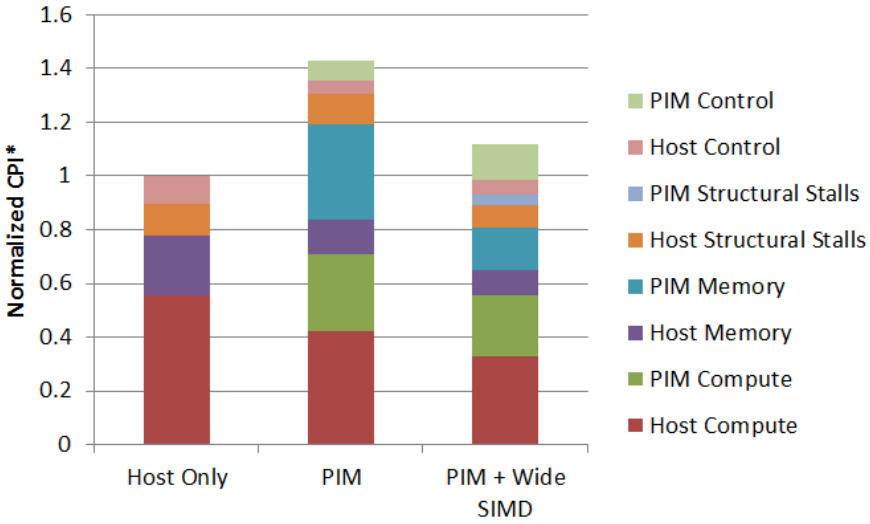

Figure 2: FFT cycles-per-instruction. The SIMD instructions count as 64 instructions in this chart.

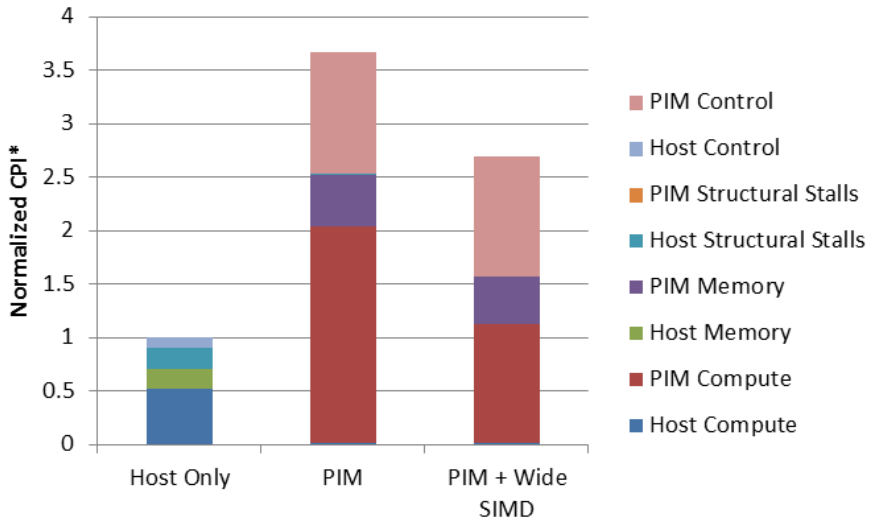

Figure 3: Fluidanimate cycles-per-instruction. The SIMD instructions count as 64 instructions in this chart.

7 Conclusion

Based on our results, a processor-in-memory with wide SIMD does not appear to be worth pursuing. This is because most applications suited for vectorization, such as `fft` and `fluidanimate`, exhibit a great deal of locality. Such programs are better suited for fast and deep memory hierarchies rather than moderately fast access to main memory. Furthermore, SIMD designs help compute-bound programs, while PIM designs help memory-bound programs. The two strategies target different types of programs and provide little benefit in combination. Wide SIMD in isolation appears to deserve further research, as it provided major performance benefits on the PIM.

References

[1] Christian Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: ACM, 2008, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128. URL: http://doi.acm.org/10.1145/1454115.1454128.

[2] Paul Dlugosch et al. "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing". In: *IEEE Transactions on Parallel and Distributed Systems* 99.PrePrints (2014), p. 1. ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.8.

[3] Jeff Draper et al. "The Architecture of the DIVA Processing-in-memory Chip". In: *Proceedings of the 16th International Conference on Supercomputing*. ICS '02. New York, New York, USA: ACM, 2002, pp. 14–25. ISBN: 1-58113-483-5. DOI: 10.1145/514191.514197. URL: http://doi.acm.org/10.1145/514191.514197.

[4] Kayvon Fatahalian and Mike Houston. "A Closer Look at GPUs". In: *Commun. ACM* 51.10 (Oct. 2008), pp. 50–57. ISSN: 0001-0782. DOI: 10.1145/1400181.1400197. URL: http://doi.acm.org/10.1145/1400181.1400197.

[5] Ehsan K. Ardestani and Jose Renau. "ESESC: A Fast Multicore Simulator Using Time-Based Sampling". In: *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. HPCA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 448–459. ISBN: 978-1-4673-5585-8. DOI: 10.1109/HPCA.2013.6522340. URL: http://dx.doi.org/10.1109/HPCA.2013.6522340.

[6] Cyril Kowaliski. *Intel Shows Larrabee Doing Real-time Ray Tracing*. Sept. 2009. URL: http://techreport.com/news/17641/intel-shows-larrabee-doing-real-time-ray-tracing (visited on 05/08/2014).

[7] Anderson Kuei-An Ku, Jingling Xue, and Yong Guan. "Gather/scatter hardware support for accelerating Fast Fourier Transform". In: *Journal of Systems Architecture* 56.12 (2010), pp. 667–684. ISSN: 1383-7621. DOI: http://dx.doi.org/10.1016/j.sysarc.2010.09.007. URL: http://www.sciencedirect.com/science/article/pii/S1383762110001153.

[8] Jiayuan Meng, J.W. Sheaffer, and K. Skadron. "Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth". In: *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. May 2012, pp. 107–118. DOI: 10.1109/IPDPS.2012.20.

[9] Moinuddin K. Qureshi and Gabe H. Loh. "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Vancouver, B.C., CANADA: IEEE Computer Society, 2012, pp. 235–246. ISBN: 978-0-7695-4924-8. DOI: 10.1109/MICRO.2012.30. URL: http://dx.doi.org/10.1109/MICRO.2012.30.

[10] S. Seo et al. "Customizing wide-SIMD architectures for H.264". In: *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium on*. July 2009, pp. 172–179. DOI: 10.1109/ICSAMOS.2009.5289229.

[11] Princeton University. *The PARSEC Benchmark Suite*. 2013. URL: http://parsec.cs.princeton.edu/parsec3-doc.htm (visited on 05/08/2014).