

---

# THE MOVIDIUS MYRIAD ARCHITECTURE'S POTENTIAL FOR SCIENTIFIC COMPUTING

---

THIS ARTICLE DESCRIBES THE DESIGN AND IMPLEMENTATION OF DENSE MATRIX MULTIPLICATION ON THE MOVIDIUS MYRIAD ARCHITECTURE AND EVALUATES ITS PERFORMANCE AND ENERGY EFFICIENCY. THE AUTHORS DEMONSTRATE A PERFORMANCE OF 8.11 GFLOPS ON THE MYRIAD I PROCESSOR, AND A PERFORMANCE/WATT RATIO OF 23.17 GFLOPS/W FOR A KEY COMPUTATIONAL KERNEL. THESE RESULTS SHOW SIGNIFICANT POTENTIAL FOR SCIENTIFIC-COMPUTING TASKS AND INVITE FURTHER RESEARCH.

**Mircea Horea Ioniță**  
**David Gregg**  
Trinity College Dublin

..... Over the past decade, power and energy efficiency have become two of the main limiters of parallel-computing performance from the perspectives of absolute energy efficiency and thermal power dissipation, with mobile computing being the major driving factor behind this trend. However, the concern with energy efficiency is not limited to mobile computing; a system's energy efficiency (in the form of performance per watt) has become a key metric. GPUs, digital signal processors, and field-programmable gate arrays are typically significantly more energy efficient than conventional desktop or server CPUs; their deployment often leads to  $4\times$  increases in energy efficiency over traditional CPUs.<sup>1,2</sup> With mobile processor architectures becoming ever more versatile, the market is gradually moving away from custom silicon solutions toward energy-efficient CPUs.

The Movidius Myriad I processor can achieve 15.84 Gflops while dissipating only 0.35 W,<sup>3</sup> for a theoretical peak of around 45 Gflops/W. This level of power efficiency results from fundamental architectural choices: very long instruction word (VLIW), instruction-level parallelism (ILP), noninterlocked execution pipelines, and software managed on chip memory. However, the design choices leading to that level of performance make it an interesting target for a broader family of applications.

We investigate the possibility of using this architecture for typical scientific workloads; more specifically, we implement and evaluate the SGEMM primitive (dense matrix multiplication) on this platform. SGEMM is part of the Basic Linear Algebra Subroutines (BLAS) Level 3 family of functions, and is regarded as a major indicator of a system's computational potential.<sup>4</sup>

In order to achieve an efficient implementation, we exploit the platform's multiple kinds of parallelism, and we provide extensive details of the process, from which an application design methodology can be derived.

This article describes the design and implementation of SGEMM on the Myriad I mobile processor and shows how to use the Myriad instruction set to maximize single-core performance. We also describe our explicit data movement algorithm, which exploits data locality in the banked local memory, and we evaluate our implementation's performance in terms of both computational throughput and energy efficiency.

## The Myriad I mobile processor

Myriad I is a mobile processor designed from the ground up for power-efficient computation. It features eight highly specialized cores featuring single-instruction, multiple-data (SIMD) processing and instruction predication, known as streaming hybrid architecture vector engines (SHAVEs). A general-purpose reduced-instruction-set computing (RISC) core is used for task allocation, coordination, and peripherals management (Figure 1).

The chip's main memory comprises either 64 or 16 Mbytes of DRAM; it resides inside the CPU package on a separate die and is bonded internally to the main Myriad die. Additionally, the main die contains 1 Mbyte of directly addressable static RAM (SRAM), organized as eight physical 128-Kbyte slices, each associated with one SHAVE core. This SRAM is arranged in a connection matrix (CMX) topology, which forwards accesses between the slices. It is not a bus slave; instead, each SHAVE has dedicated memory access ports in its (physically) neighboring memory slice, which is described as being local to it. A SHAVE core always posts a CMX memory request to its local slice, which either services it or forwards it to the target slice through the CMX. This lets any SHAVE access any CMX slice at any time.

The CMX memory comprises single-port SRAM blocks; therefore, it has the performance of an L1 cache while being completely addressable. This makes the CMX the

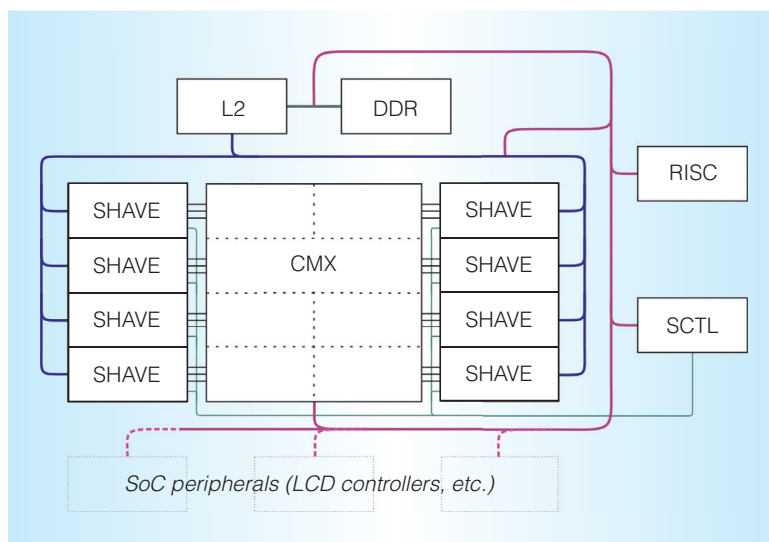


Figure 1. Movidius Myriad I processor, top-level view. The interconnect fabric is shown, as are the core components: the memory (CMX, DDR, L2 cache), the cores (SHAVEs, RISC), and the SHAVE control interface (SCTL).

preferred operating memory for both code and data.

Each SHAVE core has a direct memory access (DMA) engine for moving data between the CMX and main memory. This data movement is entirely under software control and requires the programmer to explicitly request the transfers.

The system features 128 Kbytes of L2 cache, which caches only the main memory, and can be used as a shared cache or can be partitioned into multiple independent caches.

A SHAVE core has a VLIW architecture that can start up to eight operations per cycle. It features two independent load/store units, a branch/repeat unit, a predicated execution unit, and three arithmetic units—integer, scalar, and vector. Additionally, a data comparison and moving unit handles data comparisons, data copying between register files, and format conversions. Every instruction can issue operations to any combination of the execution units.

Each of these units can begin executing one operation per cycle, which can lead to significant ILP. However, no data dependency checking is performed in hardware; it is entirely up to the compiler (or programmer) to find operations that can be issued in parallel, and to form instructions by scheduling these together.

```

float4 *pA, *pB;
float4 c0, c1, c2, c3, this_c;
// load 4x4 block of A
float4 a0 = *pA; pA += wA;
float4 a1 = *pA; pA += wA;
float4 a2 = *pA; pA += wA;
float4 a3 = *pA;
// load 4x4 block of B
float4 b0 = *pB; pB += wB;
float4 b1 = *pB; pB += wB;
float4 b2 = *pB; pB += wB;
float4 b3 = *pB;
// transpose B block
float4 bt0, bt1, bt2, bt3;
vector_transpose(bt0, bt1, bt2, bt3,
                 b0, b1, b2, b3);

// compute c0
float4 prod0 = a0 * bt0; // SIMD
float4 prod1 = a0 * bt1;
float4 prod2 = a0 * bt2;
float4 prod3 = a0 * bt3;
float sum0 = hsum(prod0);
float sum1 = hsum(prod1);
float sum2 = hsum(prod2);
float sum3 = hsum(prod3);
this_c.x = sum0;
this_c.y = sum1;
this_c.z = sum2;
this_c.w = sum3;
c0 += this_c; // SIMD
// compute c1
...

```

Figure 2. Kernel pseudocode fragment for matrix multiplication. Depicted is the code that reads the data from memory, as well as the code that computes one row of the result; the other three steps are identical. C matrix reading/writing happens before and after the kernel, rather than in it, and is not depicted.

The SHAVE cores fully implement single-precision IEEE754 floating-point arithmetic, as well as OpenCL half-precision (16-bit) arithmetic. The RISC core also features double-precision floating-point arithmetic, making mixed-precision algorithms feasible.

The Myriad I chip can run at a top frequency of 200 MHz, with its memory clocked no higher than 133 MHz.

Concerning power management, Myriad favors running at the maximum required frequency at all times, while allowing power to unused parts of the chip to be shut down dynamically. Given the chip's low thermal design power (TDP) and the significant contribution of static dissipation to this TDP, the frequency power correlation is rather weak, as predicted by previous research.<sup>5,6</sup>

## Matrix multiplication on a single core

In order to achieve good performance, two things are required: an efficient, single-core kernel and a suitable scheduling algorithm. This section describes the design of the kernel, which will be replicated on all the cores.

### Naive vector parallelism

Our matrix multiplication algorithm is built around a kernel that computes a single  $4 \times 4$  block of the result matrix  $C$ , by multiplying a  $4 \times 4$  block of each of the two input matrices,  $A$  and  $B$ . We use vector SIMD operations to implement it, and keep each of the  $4 \times 4$  blocks of  $A$ ,  $B$ , and  $C$  in registers; each  $4 \times 4$  block requires four vector registers, so all blocks fit comfortably in the existing 32 registers.

Figure 2 shows the pseudocode for a simplified version of the vector intrinsic code in our kernel. Note that we store the matrices in  $C$  (row major) format.

The algorithm comprises two loops:

- The inner loop walks simultaneously over four rows of  $A$  and four columns of  $B$ , fully computing a  $4 \times 4$   $C$  block.
- An outer loop walks across all of  $B$  and  $A$ , running the inner loop on each pair of blocks.

To compute a  $4 \times 4$  cell of  $C$ , we must multiply the rows of  $A$  by the columns of  $B$ , so in order to use SIMD, the  $B$  block must be transposed. The Myriad SHAVE supports a single-cycle  $4 \times 4$  transpose instruction, which avoids the cost of vector shuffling; the pseudocode uses a single intrinsic to show that.

Multiplying two  $4 \times 4$  blocks requires 128 floating-point operations (64 multiplications, 64 additions). The vector arithmetic unit (VAU) can start a vector multiply or addition per cycle, operating on four pairs of values, so by using the VAU we could expect the kernel to take at least 32 cycles.

However, the scalar arithmetic unit (SAU) features a horizontal-vector sum operation. It is therefore possible to initiate both a vector multiplication and a horizontal sum in the same machine cycle, reducing the problem of

implementing an efficient  $4 \times 4$  block matrix multiplication to performing the computation using vector multiplies and horizontal sums.

The second half of Figure 2 shows our solution: we multiply a row of  $A$  by a row of  $B^T$  (a column of  $B$ ), and we then use the horizontal sum to reduce the four products to the value of one element of the resulting  $C$  block.

The kernel accumulates the values of successive  $4 \times 4$  matrix multiplications to a single  $C$  block, which we represent as four vectors. To do this, we gather the sums from the scalar registers where they are produced into a vector register (`this_c`); once `this_c` is populated, it is added to the vectors representing the  $C$  block.

### Instruction-level parallelism

In addition to vector parallelism, the SHAVE cores offer significant ILP, allowing multiple operations to be executed each cycle. In order to overlap the execution of operations from successive iterations of our kernel loop, we applied modulo scheduling.<sup>7</sup> This allows a new iteration of the loop to begin before the previous one completes.

Using VAU multiplication operations and SAU horizontal summations, we need only 16 instructions to perform all 64 required multiplications and 48 of the 64 required additions. The remaining 16 additions can be performed using VAU (vertical) additions—accumulating the current result to the  $C$  buffer.

The VAU is thus the critical resource, being busy for at least 20 cycles in any instruction schedule, performing the multiplications and the  $C$  accumulation.

### Platform-specific improvements

To lower the required instruction count further, an alternative approach was sought. First, the SHAVE cores feature a SIMD multiply and accumulate unit, with a dedicated vector accumulator. Second, the SAU is idle much of the time, and some of the result accumulation can be performed on it. These changes complicate the loop, but with aggressive modulo scheduling, it can be compacted into 18 instructions, with an ILP of 3.39.

An iteration of the kernel runs in 19 cycles. Given that each iteration performs

128 floating-point operations, the kernel achieves a performance of 6.74 floating-point operations (flops) per cycle. The measured performance of the entire algorithm is 6.62 flops/cycle for CMX slice-sized workloads.

## Multicore algorithm

Myriad I does not have a threading system for executing code on the SHAVE cores; all the memory is shared, leaving it to the programmer to choose how best to use it. In these circumstances, we chose to program the Myriad as a bare metal system, allowing the code to fully control its behavior. We chose to implement all the scheduling code on the RISC core, and use the SHAVE cores only for computation.

### Background

We found the major system bottleneck to be the main memory interface. In particular, using the same kernel on a 1-Mbyte dataset, we achieved 6.62 flops/cycle (in CMX) versus 0.14 flops/cycle (in DRAM)—a  $47\times$  decrease in performance. This difference prompted us to discard from the onset the possibility of using in-place DRAM buffers.

#### *Data movement and locality considerations.*

There is a total of 1 Mbyte of CMX memory, divided into eight 128-Kbyte slices, each paired to a specific SHAVE core (“local slice”). However, any SHAVE core can access any memory location in any of the CMX slices.

This design is rather novel; the existing literature on programming systems with per-core isolated memory (such as the Cell processor<sup>2</sup>) does apply, but it doesn’t address the possibilities opened by the CMX slices not being isolated. Furthermore, every SHAVE core has its own DMA engine, which operates in parallel with the regular instruction execution. This fact immediately points to the potential usefulness of double buffering.

To operate on larger-than-CMX-memory buffers, we used tiling to divide the matrices into blocks that fit the locally available CMX memory. On each SHAVE, we programmed its DMA controller to transfer the required blocks ( $A$ ,  $B$ , and the previous  $C$ ) into the local buffers, performed the computation

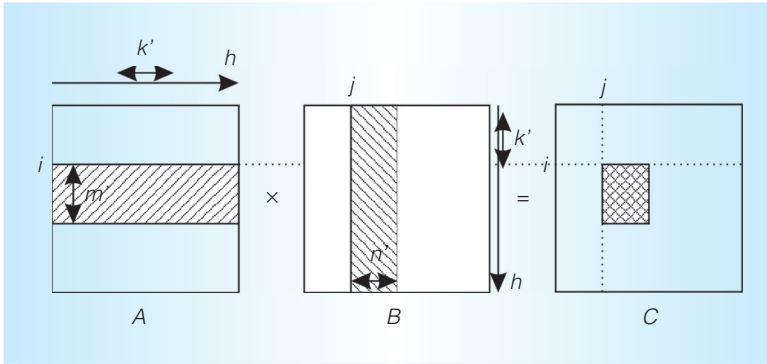


Figure 3. Basic CMX tiling: one  $(i, j)$  iteration.

locally, and then used the DMA controller to transfer the result (updated  $C$  block) back to its main memory destination. Also, because DRAM accesses are expensive performance-wise, even when using DMA transfers, it is desirable to reuse data already available in CMX, where possible.

### Multicore implementations

Our first implementation attempted to maximize the cached data reuse, as in Kodukula et al.<sup>8</sup> However, instead of mind-ing caches, our implementation used a DMA schedule—blocks of  $A$  and  $B$  get “cached” in the CMX memory, matrix multiplication is performed locally, and the result is copied back to the main memory (see Figure 3).

Let the matrices be  $A : m \times k$ ,  $B : k \times n$ , and  $C : m \times n$ . We define  $m' (\leq m)$ ,  $k' (\leq k)$ , and  $n' (\leq n)$ , such that a SHAVE core’s local CMX memory can accommodate the buffers  $a : m' \times k'$ ,  $b : k' \times n'$ , and  $c : m' \times n'$ . The  $\{m', k', n'\}$  triplet denotes the matrix block sizes.

A scheduler runs on the RISC core, dividing the matrices into  $\{m', k', n'\}$  blocks that are assigned to SHAVE cores for computation; the code on the SHAVE will DMA the input data to CMX, compute locally, and DMA the result to DRAM.

This approach resulted in a performance of up to 11.71 flops/cycle. The follow-up investigation showed that the performance was severely limited by the L2 cache thrashing; the obvious solution would be to maximize cache reuse, giving priority to  $B$ , which is not accessed sequentially. We modified the

algorithm by allocating a maximally sized  $B$  buffer in each slice, and transferring the same  $B$  block to all SHAVEs.

To determine the matrix block sizes, we use the following algorithm:

- find the largest  $k'$  such that a CMX slice has room for a  $(4 \times k')$   $A$  buffer, a  $(k' \times 4)$   $B$  buffer, and a  $(4 \times 4)$   $C$  buffer, where  $k' = k/2p$ ; find the largest  $n''$  such that the  $4 \times k'$ ,  $k' \times n''$ , and  $4 \times n''$ , buffers fit in the slice;
- find the smallest  $n'$  such that  $\lceil [(n/n')]/nShaves \rceil = \lceil [(n/n'')]/nShaves \rceil$ ; and
- find the largest  $m'$  such that  $m' \times k'$ ,  $k' \times n'$ , and  $m' \times n'$  buffers fit.

This approach minimizes the number of iterations, while maximizing the buffers.

We changed the flow to pick a  $B$  block, broadcast it to all the SHAVE cores, and then dispatch all the  $A$  blocks to SHAVE cores; once  $A$  was exhausted, we moved on to the next  $B$  block. With this approach, we achieved a performance of 16.98 flops/cycle, which fell short of expectations.

The intuitive way to improve this figure is to add double buffering; however, that would be useful only for small  $B$  matrices, because for larger ones  $k'$  would be halved, and this would double the number of dispatched tasks and negate the benefit of double buffering. Moreover, the  $A$  and  $C$  blocks are small enough that double buffering them would provide little benefit.

*Tightly coupled SHAVE cores.* Because there was no obvious way to significantly improve the previous result, we turned to the Myriad architecture for a different approach. We chose to exploit the fact that any SHAVE core can directly access any CMX slice.

We use the same blocking strategy described earlier, with the difference that multiple  $C$  buffers ( $c_{0 \dots nShaves-1} : m' \times n'$ ) are allocated in each slice.

The algorithm transfers to each SHAVE core unique  $B$ ,  $A$ , and  $C$  blocks (see Figure 4), then computation is performed, with every SHAVE core multiplying its local  $A$  block by one of the  $B$  blocks,



accumulating the result in one of its local  $C$  buffers.

However, when multiple SHAVE cores attempt to access the same CMX RAM block, the local one always wins the arbitration; this is handled by grouping tasks into rounds and synchronizing these such that no two cores access the same  $B$  block in any round. Additionally, we chose a slice memory layout that minimizes the chance of  $A/B$  RAM access conflicts.

The algorithm proceeds as follows:

1. DMA individual blocks of  $B$ ,  $A$ , and  $C$ , respectively, to all the SHAVES.
2. Let  $i = 0$ ; for each SHAVE, select output buffer  $c_i$  and run all of them on their local  $A$  buffer, and the  $i$ th (modulo  $nShaves$ ) next SHAVE's  $B$  buffer, storing the results in the local  $c_i$  buffer; proceed to the next step only after all are finished computing (see Figure 5a).
3. Iterate at Step 2 for  $i = 1 \dots nShaves - 1$  (Figure 5b depicts Step 2 when  $i = 1$ ).
4. Here, all  $A$  and  $B$  buffer combinations have been processed, and every SHAVE's ( $c_0 \dots nShaves - 1$ ) buffers are full.
5. DMA all the  $c$  buffers, from all CMX slices, back to DRAM.
6. For each SHAVE, in sequence, load only the next  $A$  horizontal blocks, and iterate at Step 2.
7. Once no more  $A$  blocks are available, DMA again the first  $nShaves$  blocks of  $A$  and the next  $nShaves$  blocks of  $B$ , and iterate at Step 2.

Once no more  $B$  blocks are available, the algorithm is complete.

We achieved 37.43 flops/cycle, a significant improvement over the classical approach. Also, because every SHAVE core receives its unique block of  $A$  and  $B$ , double buffering is likely to increase the performance further. Since  $B$  blocks change  $nShaves \times$  less often than  $A$  or  $C$  blocks, we chose to use double buffering only for the latter. We achieved this by modifying the blocking strategy to allow for two  $A$  buffers ( $a_{0 \dots 1} : m' \times k'$ ), several  $C$  buffers ( $c_{0 \dots 1, 0 \dots nShaves - 1} : m' \times n'$ ), and a  $B$  buffer

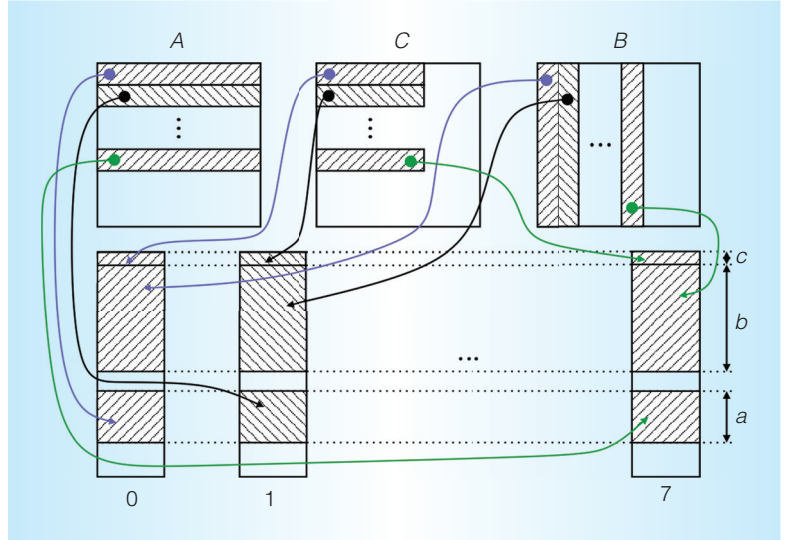


Figure 4. Mapping of matrix blocks on CMX memory slices. The numbers represent SHAVE cores; the boxes represent their local CMX memory slices.

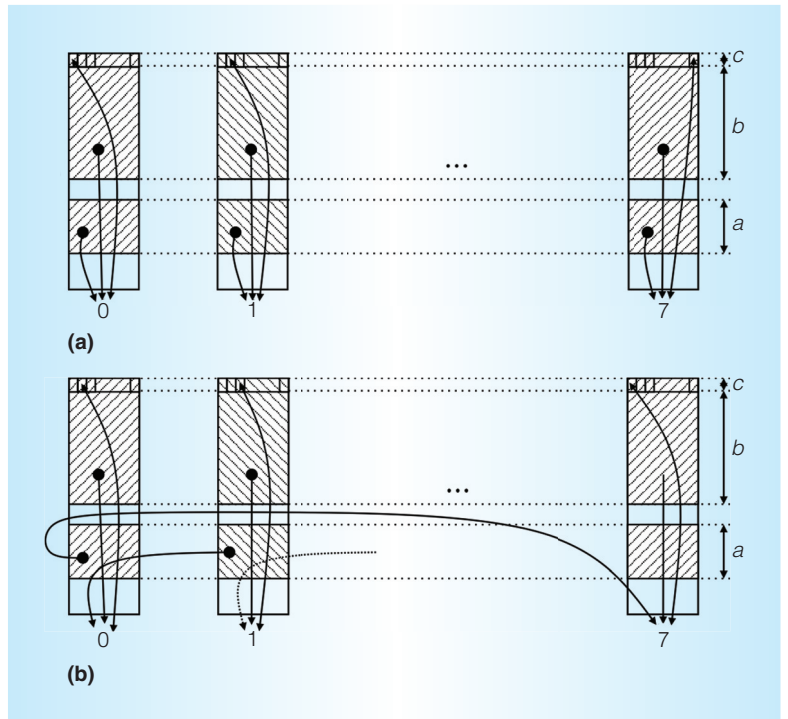


Figure 5. Step 2 of the algorithm: (a) when  $i = 0$ ; (b) when  $i = 1$ . It shows how every SHAVE's  $B$  and  $C$  buffers change depending on the iteration, while the  $A$  buffer is retained at all times.

( $b : k' \times n'$ ). The algorithm itself flows as described previously, with one change:  $A$  and  $C$  refills and  $C$  flushes happen in the background.

**Table 1. Estimated performance of existing architectures for SGEMM.**

| Architecture                             | Gflops      | Gflops/W    |
|--|-------------|-------------|
| Core i7 960                              | 96          | 1.2         |
| Nvidia GTX280                            | 410         | 2.6         |
| Cell                                     | 200         | 5.0         |
| Nvidia GTX480                            | 940         | 5.4         |
| Stratix IV field-programmable gate array | 200         | 7.0         |
| TI C66x digital signal processor         | 74          | 7.4         |
| Xeon Phi 7120D                           | 2,225       | 8.2         |
| Tesla K40 (+CPU)                         | 3,800       | 10.0        |
| <b>Myriad I</b>                          | <b>8.11</b> | <b>23.2</b> |
| Tegra K1                                 | 290         | 26.0        |

Double buffering improves performance to 45.05 flops/cycle, equivalent to 5.63 flops/cycle per core. Considering that the kernel's peak performance is 6.62 flops/cycle per core, we are achieving 85 percent of the peak performance for SGEMM. Given the system frequency of 180 MHz, this translates to a performance of 8.11 Gflops.

#### Energy efficiency

Our experimental results show that our matrix multiplication on eight cores offers overall performance of 8.11 Gflops, and the published TDP for Myriad I is 0.35 W.<sup>3</sup> On the basis of these figures, we can expect that Myriad I will achieve 23.17 Gflops/W.

Table 1 summarizes the energy efficiency of several general-purpose and special-purpose processors when running SGEMM. These figures are estimates compiled by Igual et al.<sup>9</sup> and Pedram et al.,<sup>10</sup> as well as data presented by Intel<sup>11</sup> and Nvidia<sup>12</sup>; although these estimates might not be perfect, they have been used in a recent similar study of a digital signal processor.<sup>9</sup>

Because SGEMM is computationally bound, we expect that for most of these platforms the systems would run at full speed, regardless of whether they could be more efficient at a different frequency or voltage point. Additionally, it makes us expect that the energy efficiency of most of these systems would peak alongside the maximum performance.<sup>5</sup>

## Discussion

The energy efficiency achieved by Myriad I for SGEMM is remarkable, considering it is exceeded only by that of a 28-nm GPU, whereas Myriad is a 65-nm processor. A single algorithm is insufficient to draw overall conclusions; however, comparing its architecture with existing systems could let us infer more about its potential.

The most similar existing top-level architecture is that of Cell, the performance profile of which is well-known. There are, however, outstanding differences. Synergistic processing elements (SPEs) are purely vector cores, whereas the SHAVEs are more similar to bundles of specialized RISC cores. SHAVEs have scalar registers, and the memory is granular at byte level. Also, their instruction encoding permits twice the ILP possible on an SPE. Furthermore, the memory model is more flexible than that of Cell, which lacks a unified address space.

We can draw two conclusions. First, the similarity of Myriad and Cell leads to the expectation that, maintaining proportions, their performance curves would be largely similar for the same application. Second, the SHAVE design lets us leverage the developments in VLIW compiler research alongside those in the RISC field.

A major issue concerning scientific computing is the lack of double-precision floating-point support on SHAVEs. We can partially avoid this by using mixed-precision algorithms, running the bulk of the computation on the SHAVEs and the refinement step on the RISC, which is double-precision capable. In this respect, we believe Myriad is in a similar position to GPUs of 10 to 15 years ago: if the potential were demonstrated, hardware support would follow.

Scientific-computing techniques have important applications outside the traditional scientific establishment. Linear algebra is used in game physics, computer vision, and other applications that were once confined to large-scale computing but are now increasingly found in mobile systems. As computing moves onto mobile devices, the need for energy-efficient high-performance platforms will only increase.

In this field, the Myriad architecture is unique, mainly because it was designed from scratch with the goal of providing specifically that. We believe that Myriad represents a new class of ultra low-power processors; this article aims to show the Myriad architecture's potential for scientific computing, so that parallel-computing researchers can start to consider these types of processors in future machines. However, as with GPUs a decade ago, some practical problems remain to be solved. Nonetheless, history suggests that many of the most important lessons are learned at the boundaries of the technology, and that hardware and software techniques developed for the most power-efficient systems will influence other systems where power efficiency is important.

In this article, we have focused on Myriad's architectural features. However, given their benefits for both power efficiency and computational throughput, we believe that some of these features will become mainstream in the next generations of computing platforms.

MICRO

## Acknowledgments

This work was supported in part by Science Foundation Ireland grants 10/CE/I1855 and 12/IA/1381 to Lero, the Irish Software Engineering Research Centre ([www.lero.ie](http://www.lero.ie)). We also thank Movidius for the continued support that made this work possible.

## References

1. M. Ali et al., "Level-3 BLAS on the TI C6678 Multi-Core DSP," *Proc. IEEE 24th Int'l Symp. Computer Architecture and High Performance Computing*, 2012, pp. 179–186.
2. S. Williams et al., "The Potential of the Cell Processor for Scientific Computing," *Proc. 3rd Conf. Computing Frontiers*, 2006, pp. 9–20.
3. D. Moloney et al., "1TOPS/W Software Programmable Media Processor," *Hot Chips 23*, 2011; [www.hotchips.org/wvp-content/uploads/hc\\_archives/hc23/HC23.19.8-Video/HC23.19.811-1TOPS-Media-Moloney-Movidius.pdf](http://www.hotchips.org/wvp-content/uploads/hc_archives/hc23/HC23.19.8-Video/HC23.19.811-1TOPS-Media-Moloney-Movidius.pdf).
4. J.J. Dongarra et al., "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Mathematical Software*, vol. 16, no. 1, 1990, pp. 1–17.
5. D.C. Snowdon, S. Ruocco, and G. Heiser, "Power Management and Dynamic Voltage Scaling: Myths and Facts," *Proc. Workshop Power Aware Real-Time Computing*, 2005; [www.ssrn.nicta.com.au/publications/papers/Snowdon-RH.05.abstract.pml](http://www.ssrn.nicta.com.au/publications/papers/Snowdon-RH.05.abstract.pml).
6. A. Miyoshi et al., "Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling," *Proc. 16th Int'l Conf. Supercomputing (ICS 02)*, 2002, pp. 35–44.
7. B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proc. 14th Ann. Workshop Microprogramming (MICRO 14)*, 1981, pp. 183–198.
8. I. Kodukula et al., "An Experimental Evaluation of Tiling and Shackling for Memory Hierarchy Management," *Proc. 13th Int'l Conf. Supercomputing (ICS 99)*, 1999, pp. 482–491.
9. F.D. Igual et al., "Unleashing the High-Performance and Low-Power of Multi-core DSPs for General-Purpose HPC," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC 12)*, 2012, pp. 26:1–26:11.
10. A. Pedram, R. van de Geijn, and A. Gerstlauser, "Codesign Tradeoffs for High-Performance, Low Power Linear Algebra Architectures," *IEEE Trans. Computers*, vol. 61, no. 12, 2012, pp. 1724–1736.
11. *Intel Xeon Phi Product Family: Performance Brief*, Intel, Dec. 2013; [www.intel.com/content/www/us/en/benchmarks/xeon-phi-product-family-performance-brief.html](http://www.intel.com/content/www/us/en/benchmarks/xeon-phi-product-family-performance-brief.html).
12. S. Oberlin, *Accelerating Exascale: How the End of Moore's Law Scaling is Changing the Machines You Use, the Way You Code, and the Algorithms You Use*, Nvidia, Mar. 2014; [www.siam.org/meetings/ex14/02-oberlin-slides.pdf](http://www.siam.org/meetings/ex14/02-oberlin-slides.pdf).

**Mircea Horea Ioniță** is a PhD student in the School of Computer Sciences and Statistics at Trinity College Dublin. His research interests include multicore computing and




embedded systems. Ionică has a BSc in systems engineering and computer sciences from Universitatea Politehnica in Timișoara. He is a member of Lero, the Irish Software Engineering Research Centre. Contact him at [ionicam@tcd.ie](mailto:ionicam@tcd.ie).

**David Gregg** is a senior lecturer in the School of Computer Sciences and Statistics at Trinity College Dublin. His research interests include compilers, software development tools, and program optimiza-

tion with a focus on multicore architectures. Gregg has a DTech in computer science from the Technische Universität in Vienna. He is a member of Lero. Contact him at [dgregg@tcd.ie](mailto:dgregg@tcd.ie).



*Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.*

IEEE  computer society

# ROCK STARS OF 3D PRINTING

**JESSE HARRINGTON AU**  
Chief Maker Advocate,  
Autodesk

**BRIAN GAFF**  
Partner, McDermott Will  
& Emery, LLP

**PAUL BRODY**  
VP & Global Industry  
Leader of Electronics, IBM

**3D Printing Will Actually Change the World! Are you Ready?**

**17 March 2015**  
The Fourth Street Summit Center  
San Jose, CA

**REGISTER NOW**  
Early Discount Pricing Now Available!  
[computer.org/3dprinting](http://computer.org/3dprinting)