University of Maryland Technical Report UMIACS-TR-2019-01, July 2019.

Design and Evaluation of Monolithic Computers Implemented Using Crossbar ReRAM

Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Shang Li, Luyi Kang, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung

University of Maryland and Crossbar Incorporated

ABSTRACT

A *monolithic computer* is an emerging architecture in which a multicore CPU and a high-capacity main memory system are all integrated in a single die. We believe such architectures will be possible in the near future due to non-volatile memory technology, such as the resistive random access memory, or *ReRAM*, from Crossbar Incorporated. Crossbar's ReRAM can be fabricated in a standard CMOS logic process, allowing it to be integrated into a CPU's die. The ReRAM cells are manufactured in between metal wires and do not employ per-cell access transistors, leaving the bulk of the base silicon area vacant. This means that a CPU can be monolithically integrated directly underneath the ReRAM memory, allowing the cores to have massively parallel access to the main memory.

This paper presents the characteristics of Crossbar's ReRAM technology, informing architects on how ReRAM can enable monolithic computers. Then, it develops a CPU and memory system architecture around those characteristics, especially to exploit the unprecedented memory-level parallelism. The architecture employs a tiled CPU, and incorporates memory controllers into every compute tile that support a variable access granularity to enable high scalability. Lastly, the paper conducts an experimental evaluation of monolithic computers on graph kernels and streaming computations. Our results show that compared to a DRAM-based tiled CPU, a monolithic computer achieves 4.7x higher performance on the graph kernels, and achieves roughly parity on the streaming computations. Given a future 7nm technology node, a monolithic computer could outperform the conventional system by 66% for the streaming computations.

1. INTRODUCTION

In the post-Moore era, computer architects will no longer be able to rely on technology scaling. Instead, they will need to look for new technologies to continue fueling architectural innovation. This paper explores one such possibility: *monolithic computers*. A monolithic computer relies on new logic-memory integration technology to fabricate a CPU and a high-capacity main memory system all on a single die. Compared to conventional package-level integration involving multiple dies—*e.g.*, stacking DRAM dies on top of a logic layer or integrating DRAM and CPU dies over a silicon interposer—a monolithic computer achieves much higher integration of the CPU and memory system.

Whereas package-level integration can support thousands of wires between the CPU and main memory, monolithic integration will be able to support *millions of wires*, providing a much wider main memory interface than is currently possible. This will enable architects to deliver greater memory parallelism and bandwidth to data-intensive computations, and achieve superior bandwidth-per-watt. In addition, monolithic computers will reduce the physical distance that memory requests will need to travel. Because all memory requests can stay on the CPU die, there won't be a need to cross the silicon interposer, or worse, to traverse the system motherboard. This locality benefit can provide significant additional improvements in power efficiency.

Monolithic computers do not exist yet, but some researchers believe emerging non-volatile memories will change that in the future [1, 2]. Aly *et al* [1] argue that spin-transfer torque magnetic RAM (STT-MRAM) or resistive RAM (ReRAM) are such enabling memory technologies. Unlike conventional DRAM which requires special memory fabrication processes, *STT-MRAM and ReRAM can be fabricated in a standard CMOS logic process. Hence, they have the potential to be integrated into a CPU's die.*

The main hurdle, though, is identifying a suitable integration technology. Fine-grained 3D monolithic integration [1, 2] has been proposed as a possible solution, one that assumes a process technology in which multiple planar layers of silicon can be fabricated monolithically in 3D. This would allow compute logic and non-volatile memory to be integrated in alternating planar layers. While this results in extremely high logic and memory densities, unfortunately, it requires advanced process technology that is still at the developmental stages in research labs.

In our work, we assume a much simpler integration approach that exploits non-volatile memories with 3D crosspoint architectures. In crosspoint memories–examples include Intel's Optane [3] as well as the ReRAM technology from Crossbar Incorporated [4]–the memory cells are fabricated in between metal wires of a CMOS logic process, *i.e.* at the intersection of wires laid out perpendicularly in adjacent metal layers. Rather than isolate individual cells using access transistors, crosspoint arrays provide inter-cell isolation via selector devices integrated with the memory cells. So, there are no transistors within the core of the crosspoint arrays. Instead, *the bulk of the silicon area underneath the memory are free for implementing non-memory circuits.*

It is well known that such crosspoint memories can be layered on top of compute logic during back-end of line (BEOL) processing, for example in the top metal layers of a CPU's die. Although there are no per-cell transistors, some peripheral logic is still needed at each crosspoint array for access circuitry (decoders and sense amplifiers). These access circuits will impinge upon the CPU, but the majority of the die's area is still available for CPU logic. This will permit a large amount of memory (memory exists across the *entire* CPU die) to be fabricated extremely close to the cores (the memory sits directly on top of the cores). And, it uses standard CMOS logic processing that widely exists today.

This paper investigates computer architectures that will be enabled by monolithically integrated crosspoint memories, and evaluates their potential performance benefits. To do so, we assume a particular memory technology, Crossbar's ReRAM, and develop an architecture around its specific features. One of the key features driving our design is *massive parallelism in the integrated ReRAM*. Crossbar's ReRAM is extremely dense; and, it relies on current-based sensing that limits the extent of wordlines and bitlines within individual sub-arrays. This means the ReRAM sub-arrays are quite small. Hence, there can be a very large number of them, 10s of thousands, integrated into a CPU's die, with each providing an independent and parallel access point.

At the same time, ReRAM also exhibits a *fine access granularity*. Not only does current-based sensing make ReRAM sub-arrays small, it also limits the number of bits that can be sensed simultaneously. For example, each ReRAM sub-array may provide only 4 bits of data per access. So, multiple sub-arrays must be grouped together and accessed as a bank to provide larger access sizes. Since there is a fixed number of sub-arrays that can fit on a CPU die, there exists a tradeoff between granularity and parallelism: a coarser access granularity results in lower memory parallelism while a finer access granularity increases memory parallelism.

We propose to integrate a large number of memory controllers across the CPU die to support the massive memory parallelism in monolithically integrated ReRAM. The architecture we envision has up to 400 controllers. To maximize parallelism, we form banks that support the finest access granularity of interest to the CPU, which we assume to be 8 bytes. At 8-byte granularity, we estimate there will be 16,000 to 64,000 banks across the CPU die. Notice, there are 1–4 million wires emanating from these banks, eventually connecting to the controllers. Such an extremely parallel memory interface is possible given the monolithic integration of the controllers with main memory.

The fine-grained banks in our architecture will support programs with *irregular memory access patterns* very effectively. However, we must not overlook regular computations (*e.g.*, streaming) which prefer coarse-grained data transfers to amortize data movement overheads. We propose supporting a coarser granularity at the controllers by simply activating multiple fine-grain banks together to fetch at cache-block granularity whenever memory accesses exhibit spatial locality. While this does not reduce overheads at the bank level, it can increase efficiency upstream in the on-chip network and processor caches. Even when fetching 64-byte cache blocks, the on-die ReRAM can sustain 2,000–8,000 simultaneous requests, still quite a lot of parallelism.

In addition to the memory system, Crossbar's ReRAM also affects our design of the CPU. Besides massive parallelism, another key feature of ReRAM is higher access latency. Unfortunately, ReRAM is currently not as fast as DRAM, exhibiting latencies in the 100s of nanoseconds. The CPU must be able to tolerate these latencies by exploiting the available memory parallelism. We employ *tiled processors* [5, 6] to accomplish this. Our CPU consists of many cores interconnected by a scalable point-to-point on-chip network, with support for multithreading and SIMD vector units in each core. In particular, we leverage recent SIMD ISAs, such as Intel AVX-512 [7], that support wide scatter-gather memory operations. Scatter-gather can efficiently generate a large number of fine-grained requests to the memory system, which is a good match to the on-die ReRAM.

Despite having multiple cores, multi-threading, and wide SIMD, we find that fully exploiting all of the parallelism in the integrated ReRAM memory system is challenging. In particular, popular tiled CPUs, like Intel's Knights Landing (KNL) [8], do not exhibit enough parallelism. To scale up, our tiled CPU eschews some of the general-purpose features of CPUs like KNL. First, we use smaller in-order cores to enable higher core and thread counts, at the expense of single-thread performance. We envision having up to 400 cores supporting 2000 simultaneous threads. And second, we adopt a simple two-level cache hierarchy, omitting cache coherence to save on the large coherence directories that would be needed for 100s of cores. In that regard, our architecture resembles a tiled accelerator [9] more than a general-purpose multicore.

Tiled processors also provide a natural way to combine the memory and CPU architectures. Existing tiled CPUs employ off-chip memory systems, such as HBM DRAM, and provide memory access through a small number of controllers at the periphery of the on-chip network. As mentioned earlier, we foresee having a much larger number of controllers. We propose to *incorporate the controllers into the compute tiles themselves*, making them first-class citizens on the network on-par with the cores. Currently, we assume a single controller per tile, but different core-to-controller ratios are possible. This allows the controllers to scale with the compute tiles, much like in classical distributed shared memory (DSM) machines [10, 11, 12]. But now, the DSM nodes–including their main memories–are all integrated monolithically, eliminating chip crossings entirely.

The rest of this paper elaborates on our monolithic computer ideas outlined above. Specifically, we make the following contributions:

- We present Crossbar's ReRAM memory technology, describing its density, latency, integration with CPU logic, and potential for massive memory parallelism. While technology driven, the discussion is meant to inform architects on how Crossbar ReRAM could enable monolithic computers.

- We develop a CPU and memory system architecture that can effectively exploit the ReRAM's memory-level parallelism. The architecture employs a large tiled CPU with wide SIMD instructions, and incorporates memory controllers into every compute tile. It also supports a variable access granularity to maximize the ReRAM bank-level parallelism that is exposed to the CPU.

- Finally, we conduct an evaluation of monolithic computers on graph kernels and streaming computations.

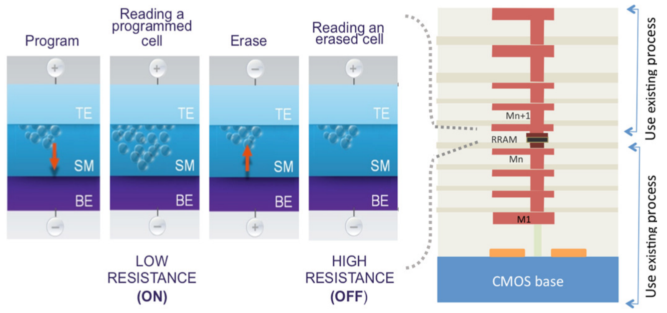

Figure 1: Crossbar ReRAM cell & close-up of the array

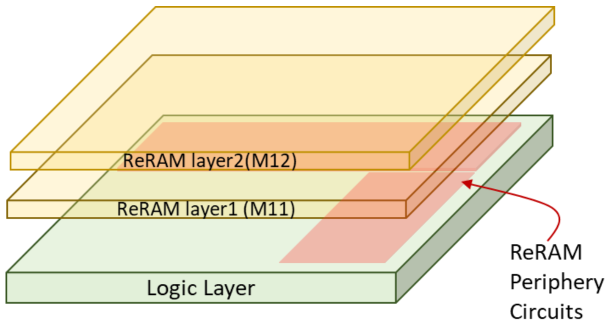

Figure 2: Monolothic CPU implementation with Crossbar ReRAM array above; Large portion of unblocked silicon area beneath the ReRAM bitcell array is available for general CMOS circuits

Compared to a DRAM-based tiled CPU, a monolithic computer achieves 4.7x higher performance on the graph kernels, and achieves roughly parity on the streaming computations. Given a future 7nm technology node, a monolithic computer could outperform the conventional system by 66% for streaming computations.

Sections 2–5 address these three contributions. Then, the paper concludes in Section 6.

2. CROSSBAR RERAM TECHNOLOGY

Crossbar ReRAM is a type of resistive memory cell [13] that can be operated and fabricated without per-cell access transistors. As shown in Figure 1, the cell elements lie between perpendicular wires, in an array that is fabricated up in the metal stack, in the back-end of line (BEOL). Because no per-cell access transistor is used, the silicon space beneath the memory array is not needed for access transistors, as it is in most memory-cell technologies, which means that the space beneath the array is actually *empty*, as shown in Figure 2. In fact, up to 75% of the area immediately under the array (the exact proportion depends on the array's organization) is empty and can be used for logic circuits having nothing to do with the array's operation. This presents a significant opportunity for integration, as one could use that empty space for core logic and SRAM caches. This is the fundamental integration technology for monolithic computers that we exploit in our work.

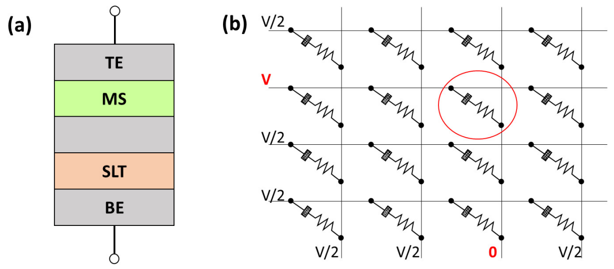

Figure 3: (a) 1S1R ReRAM bitcell cross-section showing Top Electrode (TE), Resistive element: Metallic Switch (MS), Selector device: Superlinear Threshold Layer (STL), and Bottom Electrodes (BE) (b) Crossbar array bias scheme, with selected cell circled

2.1 ReRAM Physical Characteristics

The Crossbar ReRAM bitcell is based on the creation of metallic filaments in a silicon-based switching medium for the resistive element [4, 14, 15]. Figure 3(a) shows the cross section of the 1S1R (1 selector per 1 resistive element) bitcell used. The design of the Crossbar bitcell enables a reliable memory operation without an access transistor. A proprietary FAST Superlinear Threshold Layer (STL) enables high selectivity (>10^6-10^{10}) and fast access times. This selector device is integrated with the resistive element to form the metallic switch (MS) layer for the bitcell as shown in Figure 1, sandwiched between the bottom electrode (BE) and the top electrode (TE). A voltage above a threshold (> V_{TH}) is required to select the cell to perform a read or write operation. For programming, a much higher voltage (>V_{PRG}) is applied to enable the formation or resetting of the metallic nano-filaments.

Figure 3(b) shows the bias scheme of the crossbar memory array for selection. All wordlines and bitlines are held at V/2, while the selected cell's wordline and bitline are biased to have a difference of V across it. The high selectivity of the selector device ensures minimal sneak path current on unselected cells on the same bitline–which have a potential of V/2 across their cells–thus permitting large arrays (2K x 2K cells is possible). But wordline drive current limitations bounds the number of bits that can be sensed in parallel to a small number, such as 8 bits per ReRAM sub-array layer.

Table 1 summarizes the key performance metrics of Crossbar's ReRAM array. Crossbar has successfully fabricated 2- and 4-layer stack memories, and is in the process of fabricating the 8-layer stack memory shown in Figure 1. Based on prior success, Crossbar predicts the feasiblity of 100GB to 200GB of ReRAM memory at 16nm on a 400mm^2 chip. Moreover, Crossbar expects their ReRAM to scale to 7nm in the foreseeable future, which would enable many 100s of GBs on a single die.

2.2 ReRAM Integration with Logic

When the ReRAM memory circuits are placed alongside other blocks, the peripheral circuits would be realized as blocked areas. In traditional digital implementation flow, we can think of these as placement blockage areas on which standard cells cannot be placed. For this study, we have chosen to use a 26% memory to periphery area ratio, as used

Table 1: Crossbar 1S1R ReRAM Memory Parameters

Key Parameter	Performance
Bitcell area for two-layer stack	$4F^2$
Read Latency	200-700 ns
Write Latency	≈ twice read latency
Cell Leakage	0.1 nA/cell
Program Energy	10-100 pJ/cell
Endurance	$> 10^5 - 10^8$ cycles
Retention	> 7-10 years
Scaling Potential	< 10 nm
Ron/Roff ratio	100
Selectivity (ΔI @V_R, $V_{R/2}$)	$> 10^6$ - 10^{10}

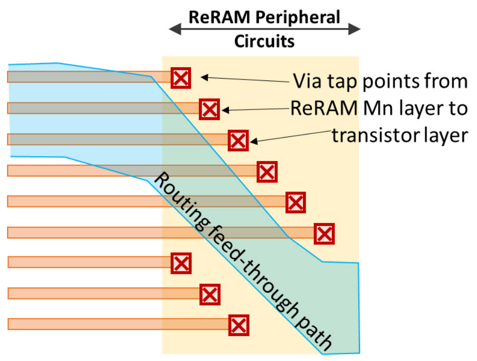
ReRAM Peripheral Circuits
Via tap points from ReRAM Mn layer to transistor layer
Routing feed-through path

Figure 4: Top-down view of via tap points from ReRAM metal layer to connect to ReRAM periphery circuits. Staggered via tap points allows for a routing channel for signals to feedthrough across blocked region.

by Crossbar for their 2-layer memory stack. This blocked region is denoted by the "L" shape in Figure 2.

The ReRAM memory developed by Crossbar is CMOS compatible and back-end of line (BEOL) stackable, and it is organized so that the bitcells are stacked on higher metal layers. Because the peripheral area must allow for connection to the ReRAM layers, there will be restriction of the general interconnect routing over these blocked regions. The connections are depicted as via tap points in Figure 4. By staggering the position of the via tap points to each of the ReRAM wordlines, a feed-through path can be accommodated. This is critical when the blocked peripheral region is embedded with another block, that would likely need a limited number of interconnections signals across the region. In our study (Section 3.1.1), we found that the area penalty is extremely high when such interconnects are not allowed.

2.3 ReRAM Latency

While resistive RAMs are, in general, significantly faster than flash-based memories (hundreds of nanoseconds vs. hundreds of microseconds), they are still not as fast as DRAM (tens of nanoseconds), which calls into question their proposed use as main memory. There are two potential tradeoffs and one mitigating factor that we consider:

1. The actual latency of an array is highly dependent on its physical dimensions and organization. That is, the length of the wires matters, because one must drive that length of metal, and the number of cells per wire matters, because it adds significantly to the capacitive effects. Figure 5 shows the relationship between the efficiency of the ReRAM array's organization and its read access time for an existing memory fabricated at Crossbar in a 28nm process. One can see the tradeoff between the array's efficiency and its latency. To wit, putting more cells on the bitline (x-axis) leads to a denser array (lower value on the left-side y-axis) but a longer latency (higher value on the right-side y-axis).

2. The latency for a cell depends on its expected lifetime. That is, like many other nonvolatile cell technologies, one can trade off the expected longevity of data stored in the cell for reduced access times, both for write operations and read operations. Long latencies in nonvolatile memory technologies are coupled to the desired longevity of the data: one expectes nonvolatile memories to hold their data for decades. However, if the expectation is reduced to, say, minutes or hours, as would be the case were the technology to be used for main memory, then one can reduce the programming and read times significantly. We predict a redesign of Crossbar's current memory to tradeoff longevity for lower access time could result in read latencies of 200-700ns (Table 1), rather than 1-2+ usec, for the most area efficient (right-most) arrays in Figure 5.

3. One mitigating factor is that, while DRAM latency is indeed tens of nanoseconds, in all practical scenarios (i.e., access by multiple processor cores through a typical memory controller), the *real, observed* latency of the DRAM-based *memory system* is in fact hundreds of nanoseconds or more. This is as true today, even for high-performance DRAMs, as it was over a decade ago [16, 17]. So, to match the actual main-memory latency seen by software, one need only provide memory latencies in the low hundreds of nanoseconds. This is indeed possible with ReRAM, because with sufficient parallelism, one can reduce the queueing delays to a small fraction of the overall memory delay.

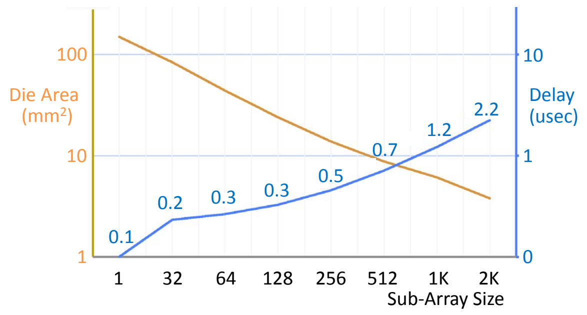

Figure 5: Trade-off between density (left-side y-axis: the die area required for the desired array capacity) and read latency (right-side y-axis: microseconds), vs. number of cells per bitline (x-axis)

While read latency is a critical parameter for most applications, high write latencies also impact the overall system performance. For ReRAM, due to its non-volatile nature, high write-energy is required to modify the data stored in the

ReRAM array. As indicated in Table 1, we expect write latency for ReRAM to be roughly twice the read latency, making writes 10-100x more costly than on DRAM-based main memory systems. Write buffers can mask much of the latency from writes, but the long occupany of banks can cause contention and in turn delay reads.

2.4 ReRAM Sub-Array Parallelism

One of the key features of ReRAM technology is the potential for massive memory-level parallelism due to the large number of ReRAM sub-arrays that can be integrated on a CPU's die. To gauge the amount of parallelism, we consider an existing 8GB memory from Crossbar implemented using 2-stack ReRAM. Assuming 2K $\times$ 2K sub-arrays (*i.e.*, the most area-efficient configuration in Figure 5), this memory contains 8,192 separate sub-arrays. In a 400 mm^2 die at 16nm, there would be 8 such memories if the entire die were completely covered with ReRAM (an overly optimistic assumption which we will address in Section 3). This memory chip would contain *65,536 sub-arrays*, each with its own access circuits, and thus, the ability to provide an independent access point. Assuming scaling down to 7nm, we could expect about a 4x increase in density, resulting in *256K sub-arrays* on a single die.

In addition, stacked ReRAM layers can support even more memory parallelism as the ReRAM cells from different layers could be sensed in parallel. Because adjacent ReRAM layers share wires, not every layer can be accessed independently. But, it could be possible to simultaneously access every other layer. (This would increase the complexity of the access circuits, and would add area overhead on top of the analyses from Sections 2.2 and 2.3. So, it comes at some cost.) In an 8-stack crosspoint array, the ReRAM layers could provide another 4x in memory parallelism. This would yield *256K independent access points at 16nm, and 1M independent access points at 7nm.*

Not all of these access points will be exposed to the CPU, in part, because the natural access size of ReRAM is too fine-grained. As discussed in Section 2.1, each crosspoint array only allows sensing a small number of bits per access, *e.g.* 8 bits (per ReRAM layer). To increase the access granularity, multiple sub-arrays must be grouped together and accessed as a *bank*; hence, the bank-level parallelism would be less than the total number of independent sub-arrays. Section 3 will discuss the amount of bank-level parallelism the CPU can see. Nevertheless, this analysis suggests integrated ReRAM has the potential to support massive levels of memory parallelism.

3. CPU AND MEMORY ARCHITECTURE

In this section, we develop an integrated CPU and main memory architecture around the specific features of Crossbar's ReRAM technology from Section 2. The main goal for the architecture is to support a large amount of parallelism in order to fully utilize the massive number of sub-arrays in the on-die ReRAM discussed in Section 2.4. Supporting as much parallelism as possible will also help tolerate the ReRAM's high access latency as discussed in Section 2.3.

Figure 6 illustrates the overall architecture. The starting point for the architecture is a tiled CPU consisting of a large number of compute tiles. Each compute tile contains a simple in-order core, a private L1 cache, and an L2 cache slice. All of the L2 slices form a physically distributed, but logically shared, last-level cache. ReRAM memory cells are implemented over all of the compute tiles in BEOL metal layers, with access circuits integrated into the CPU's logic as discussed in Section 2.2.

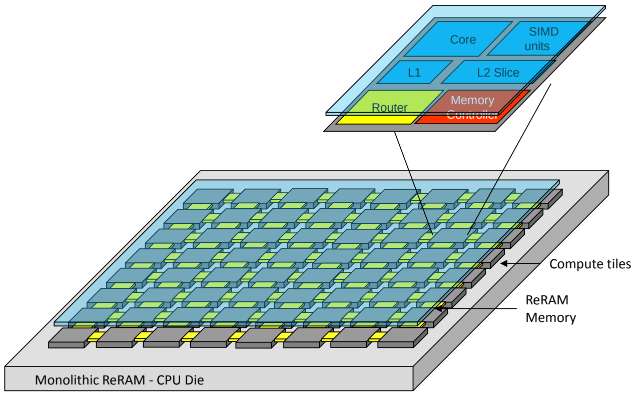

Figure 6: Tile-based monolithic computer.

The main distinguishing feature of the architecture is how the memory controllers (MCs) are incorporated into the tiled CPU. In conventional tiled CPUs, the memory system is off chip, and access to it is provided through MCs that are situated at the periphery of the network-on-chip (NOC). This means there are typically much fewer MCs compared to the number of compute tiles or cores. So, traffic to off-chip memory necessarily constricts as it funnels through the small number of MCs, leading to contention and queueing delays. As has been shown in previous work, queueing delays account for the lion's share of what is perceived to be "DRAM latency" in real systems [16].

In contrast, our proposed architecture has a much larger number of MCs that are distributed across the entire CPU die. A natural arrangement is to have one MC per compute tile, as shown in Figure 6. *This puts the MCs on par with the cores and caches*, instead of relegating them to the edges of the NOC. It provides scalability and eliminates the constriction of memory traffic at the MCs, mitigating the queueing delays that can occur in conventional architectures.

Furthermore, each MC receives memory requests on local L2 slice cache misses. The MC services such cache miss requests from the local ReRAM banks integrated above the compute tile to which the MC belongs. Hence, associating MCs with compute tiles allows each L2 slice to cache data from the physically closest ReRAM banks, thus localizing all communication between the L2 cache and the memory sub-system. The only long-haul communication occurs on L1 misses that need to access a remote L2 slice. But even then, the communication still stays on-chip.

3.1 ReRAM Banks

The numerous memory controllers in our architecture constitute one part of the solution for providing massive memory parallelism. The other major part are the ReRAM sub-arrays discussed in Section 2.4. The actual amount of memory parallelism the CPU will see depends on the number of sub-arrays that can be integrated into a CPU's die as well as the number of independent banks these sub-arrays are used to form. The following addresses both issues in greater depth.

3.1.1 *Sub-Array Area Efficiency*

In Section 2.4, we estimated the number of sub-arrays that could fit on a large CPU-sized die, but this estimate optimistically assumed that the ReRAM covers the entire CPU. The problem is that each ReRAM sub-array contains peripheral circuits (Figure 2) and present routing obstructions (Figure 4) that negatively impact the physical layout of the CPU with which it is integrated. So, in addition to the area for the peripheral circuits (about 26%), there is another area overhead incurred in the layout of the CPU itself. This CPU area overhead becomes more severe as the ReRAM sub-arrays are more densely packed. Thus, in practice, it is prohibitive to have 100% of the CPU die covered with ReRAM.

We conducted a study to quantify what fraction of the CPU die could be covered with ReRAM, and what cost to the CPU area would be incurred as a result. Our experiments integrate a standard-cell based synthesized RISC processor with ReRAM crossbar memory circuits. Specifically, the Berkeley VSCALE, which is a single-issue in-order 3-stage integer RISC-V processor, was used for the study. Because the baseline processor is quite small, we increased its size by scaling the datapath up to 256 bits.

We used the open-source NCSU FreePDK 45nm process design kit and the Nangate open source digital library for the 45nm process node. This design kit and library provides support for the Synopsys Design Compiler and Cadence Encounter. Synopsys Design Compiler is used for the synthesis step of the tool flow, and Cadence Encounter is used for the Automatic Place and Route (APR) step of the flow to produce a final GDSII layout.

To mimic the integration constraints from Section 2.2, two types of blockage layers are indicated in the Cadence Encounter setting. The first is for the placement blockage to prevent standard cells from being placed, and the second is routing blockage for the specific metal layers to limit routing. We mimic the restricted metal routing described previously by blocking metal layers 1–8 and allowing for the APR tool to route through the blocked region using metal 9 and 10. (The ReRAM memory layers are assumed to be in metal layers 11 and above).

For the placement blockage, we assume each sub-array's access circuits are situated along two edges of the sub-array, forming an "L" shape, as shown in Figure 2. We also assume the unit of replication for the ReRAM is four sub-arrays arranged in a 2×2 cluster, with rotations of 0°, 90°, 180°, and 270°, such that the peripheral circuits form a "cross" shape. This configuration allows for the peripheral blockage regions to be abutted with each other, resulting in a contiguous placement blockage region. At 45nm, and assuming $2K\times2K$ bit ReRAM sub-arrays, each 2×2 ReRAM cluster occupies $109\mu m\times109\mu m$. For our experiment, we assumed four such clusters are integrated into the CPU's logic. The question, then, is what is the optimal spacing between the cluster such that the CPU area overhead is minimized?

We iteratively tried different inter-cluster spacings to find the optimal separation. Figure 7 shows the layout of the best

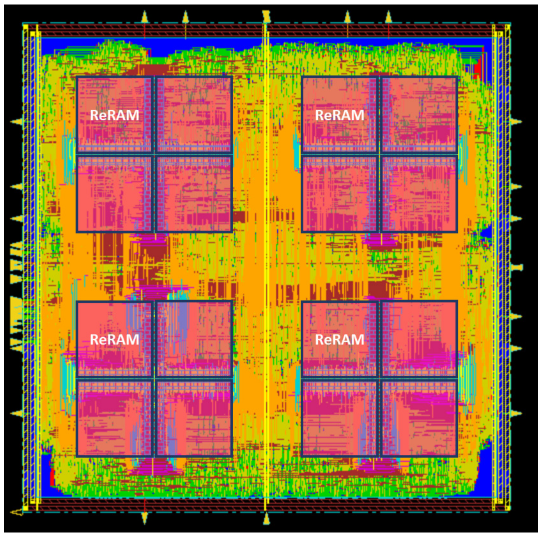

Figure 7: Four 2×2 ReRAM clusters integrated with a 256-bit RISC-V Processor.

configuration from our experiments. In the figure, the four instances of the 2×2 ReRAM clusters are clearly visible; the random logic underneath and surrounding these clusters are the placed and routed cells for the VSCALE processor. We found that an inter-cluster spacing of between $100\mu m$ to $150\mu m$ in both X and Y dimensions results in the lowest CPU area overhead. As Figure 7 shows, this allows about 50% of the CPU area to be covered with ReRAM. For this ReRAM density, the CPU area impact is a 20% increase.

Our area study only considers the impact of the ReRAM on CPU logic circuits, but CPUs also have cache memory. We envision that individual SRAM sub-arrays could be integrated underneath each ReRAM sub-array. This would require coordinating the selection of the SRAM sub-array size with the ReRAM sub-array size. While we believe this is feasible, we leave its evaluation for future work.

From this study, we conclude roughly half of the ReRAM sub-arrays that were estimated in Section 2.4 to fit over an entire CPU die could be feasibly integrated. In other words, on a 400 mm^2 die at 16nm, there are 32K sub-arrays (with 128K independent access points) on the CPU die. And, for the same die size at 7nm, there are 128K sub-arrays (with 512K independent access points).

3.1.2 *Access Granularity vs. Number of Banks*

As discussed in Section 2.4, individual ReRAM sub-arrays only provide a few bits per access. Multiple sub-arrays must be grouped into banks so as to provide a more useful data access granularity. Given that there is a fixed number of ReRAM sub-arrays on the CPU die, this gives rise to a *trade-off between the access granularity supported and the number of banks (and hence, parallelism) available to the CPU.*

To maximize bank-level parallelism, we support the finest access granularity of practical interest to the CPU, which we assume to be 8 bytes. Given 8-stack sub-arrays that can each provide 4 bytes of data per access (8 bits across 4 ReRAM layers), we would need two sub-arrays to implement each bank. On a 16nm technology node, we could thus implement

16,000 banks in total out of the 32K sub-arrays referenced in Section 3.1.1. And, on a 7nm technology node, we could implement 64,000 total banks out of the 128K sub-arrays referenced in Section 3.1.1.

As our results in Section 5 will show, supporting an 8-byte access granularity (along with massive parallelism) will benefit data-intensive computations with *sparse memory access patterns*. Doing so in an efficient manner has been extremely challenging for conventional DRAM-based systems, but it comes naturally to monolithic computers given ReRAM's fine-grain sub-arrays.

That said, it is important to not overlook regular access patterns (*e.g.*, streaming) that prefer larger data transfers to amortize data movement overheads. We propose to support a coarser granularity at the controllers by simply activating multiple fine-grain banks together to fetch a cache-block whenever memory accesses are expected to exhibit spatial locality. While this does not reduce overheads at the bank level, it can increase efficiency upstream in the NOC and processor caches. Assuming 64-byte cache blocks, we would need to activate 8 banks together for such coarse-grained fetches. These cache-block-sized fetches would incur an 8-fold decrease in the number of simultaneous requests that the memory system could support.

3.2 Parallelism vs. Locality

As mentioned earlier, the massive number of banks in our architecture will help tolerate the higher ReRAM access latencies. The degree of latency tolerance is maximized when the CPU's accesses are spread evenly across all the banks. Bank conflicts are especially costly in our architecture since they serialize accesses whose latencies are high to begin with. To reduce bank conflicts, we employ an interleaved memory addressing scheme. In particular, we interleave cache blocks across compute tiles, with adjacent cache blocks mapped to adjacent tiles (*i.e.*, the tile ID is chosen from the lowest bits of the memory block address). Within a tile, we interleave cache blocks across banks, with spatially closest cache blocks mapped to adjacent banks (*i.e.*, the bank ID is chosen from the next lowest bits of the memory block address).

This memory interleaving scheme sacrifices locality since memory accesses will tend to spread across all of the compute tiles and banks on the CPU die. But for a monolithic computer, even the worst-case locality is still quite good. Regardless of which bank we access, that memory access is guaranteed to be satisfied on-die. For a conventional computer, *all* memory accesses are guaranteed to travel off-die.

3.3 CPU Support

As eluded to earlier, the CPU must be able to generate a large amount of memory parallelism commensurate with the memory system. To do so, we eschew many of the general-purpose features found in commercial tiled CPUs in favor of scalability that is more in line with accelerators [9].

Specifically, we use simple in-order cores to allow for higher core count at the expense of single-thread performance. Each of our cores is multi-threaded, and switches between its resident hardware threads on long-latency memory operations. As discussed above, Crossbar's ReRAM can scale to 7nm. Given ReRAM's good scaling characteristics, we assume 400 in-order cores with 5-way multithreading per core could fit on the CPU die. This allows for up to 2000 threads to be resident in the CPU simultaneously. Our experiments in Section 5 will also consider smaller thread counts, but this is the maximum we simulate in our experiments.

In addition to support for thread-level parallelism (TLP), each core also has vector units for executing SIMD instructions. The vector units not only support traditional SIMD memory operations that fetch contiguous blocks of data from memory, they also support scatter-gather memory operations found in recent SIMD ISAs such as Intel's Advanced Vector Extensions 512 (AVX-512) [7]. Every word fetched during a scatter or gather operation can be destined to a distinct memory location, which increases memory parallelism by a factor equal to the SIMD width. In particular, AVX-512 allows for eight 8-byte accesses in a single scatter-gather operation. Thus, across 2000 threads, our tiled CPU is capable of generating 16,000 8-byte memory requests, which matches the number of ReRAM banks we expect at 16nm.

Besides TLP and SIMD, our CPU also employs mechanisms in the cache hierarchy to support the access granularity features of our memory architecture. As mentioned in Section 3.1.2, our ReRAM memory system supports two access granularities: a fine-grained 8-byte access, and a cache-block-sized 64-byte access. We employ sectored caches [18, 19] in both the private L1s and the logically shared L2 slices to handle variable fetch sizes. Each sector is 64 bytes, and is filled entirely by a cache-block-sized memory access. Sectors are split into 8 sub-blocks of 8 bytes each. Fine-grained accesses only fill a single sub-block within a sector.

Supporting variable access granularity also requires determining the best memory access size on a per cache-miss basis. We adopt a very simple heuristic. If a cache miss is triggered by a scatter-gather memory operation, we assume the access pattern is sparse, and request a double word only. (There may be multiple misses from the same scatter-gather operation, but each only requests a double word). For all other cache-miss triggering memory operations, we request a full cache block. This simple approach works well for the accelerator-like benchmarks evaluated in our study.

Lastly, a noteworthy aspect of our CPU lies in what it *does not support*: cache coherence. We omit coherence on the CPU's private L1 caches to save on the large coherence directories that would be needed at 100s of cores. The lack of cache coherence is in keeping with the massively parallel accelerator-like nature of our design, and also helps to make room for the CPU's large core count.

4. EXPERIMENTAL METHODOLOGY

4.1 Simulator

We created an architecture-level simulator that models the monolithic computer described in Section 3. Our approach follows recent simulators of tiled CPUs capable of running 1000s of threads [20, 21, 22]. Like these previous simulators, we use an Intel Pin-based front-end [23] to feed a parallel instruction trace to a cycle-accurate back-end. The main difference is that for the front-end, we employ Intel's Software Development Emulator (SDE) [24]. Intel SDE can execute x86 binaries with 1000s of threads. But it can also

Cores (in-order, single-issue)	400
Threads per core	5
Clock rate	1 GHz
L1 Cache	32 KB, 4-way, 1 cycle
L2 Cache Slice	256 KB, 8-way, 5 cycles
Sector Size	64 bytes
Sub-Block Size	8 bytes
ReRAM memory controllers	400 (1 per tile)
ReRAM banks	16,000 (40 per tile)
ReRAM read latency	500 cycles, 200 cycles
ReRAM write latency	1000 cycles, 400 cycles
ReRAM access granularity	8 bytes
On-chip Network	20 x 20, 2D-Mesh
Network Channels	2 x 64 bytes

Table 2: Simulation parameters for the experiments.

emulate AVX-512 instructions, a feature that is absent from existing simulators that we are aware of. This added capability allows us to further increase the CPU's memory parallelism with wide scatter-gather operations.

Our back-end receives the parallel instruction trace from SDE, and performs cycle-accurate simulation of both the CPU and on-die main memory system. Table 2 lists the simulation parameters which reflect the accelerator architecture from Section 3: a tiled CPU with 400 in-order cores, each with 5-way multi-threading. The front-end executes 2000 threads in SDE and maps them in groups of 5 onto the 400 back-end cores. (We also simulate smaller configurations which will be discussed in Section 5). Each core has access to a non-coherent private L1 cache backed by a distributed logically shared L2 cache. All caches are sectored to allow cache-miss fills at either 64-byte or 8-byte granularity. For the main memory system, we assume a single memory controller is integrated into every compute tile that is responsible for accessing the ReRAM directly over the tile. In total, there are 16,000 ReRAM banks split into 40 banks per tile. We faithfully model all bank conflicts and queuing at the controllers. Once a read request is issued to a bank, the data is returned in 500 cycles. We also consider a more aggressive 200-cycle read latency. Writes are assumed to take twice as long as reads.

The compute tiles are interconnected via a 20 $\times$ 20 two-dimensional mesh network. This NOC carries the traffic between the private L1s and the L2 slices on L1 cache misses. It employs dimension-ordered routing and supports 2 virtual channels (one for requests and one for replies), and is deadlock free. We assume each channel between any pair of NOC routers contains two uni-directional 64-byte point-to-point connections, resulting in a network bisection of 20,480 wires. While this is quite wide, it is feasible given all the routers are integrated on the same die.

In our study, we compare monolithic computers to tiled CPUs with high-performance DRAM memory systems resembling HBM DRAM. The DRAM memory system uses 8 memory controllers–4 at the corners and 4 in the middle of each edge of the NOC–that control 8 DRAM channels. (Some of our experiments vary the number of MCs and channels which will be discussed in Section 5). We assume 128 independent banks are distributed equally across these 8 DRAM channels. At large thread counts, we find row buffer hits are almost non-existent, so we assume a closed-page policy. Under this assumption, each MC access incurs

Graph Kernels		
All Pairs Shortest Path (apsp)	scale = 22	1.3
Betweeness Centrality (bc)	scale = 22	1.1
Connected Components (cc)	scale = 22	2.2
Page Rank (pr)	scale = 22	1.7
Single Source Shortest Path (sssp)	scale = 22	1.4
Streaming Computations		
daxpy	1.47B elements	1.3
K-Means Clustering (kmeans)	256K pts, 20 features	1.7
Nearest Neighbor (nn)	204.8M hurricanes	1.0
Needleman-Wunsch (nw)	32K x 32K	1.0
Pathfinder (pf)	15M cols, 100 rows	1.6

Table 3: Benchmark names, input sizes, and number of instructions simulated (in billions).

a 32-cycle latency followed by 2 cycles of data transfer. All DRAM accesses occur at cache-block granularity, which for the baseline system is 128 bytes. The peak DRAM bandwidth is 512 GB/sec. Lastly, because the DRAM MCs are situated at the periphery of the NOC, each L2 slice must additionally communicate with a remote MC on an L2 miss.

4.2 Benchmarks

We drive our simulations using 10 benchmarks, which are listed in Table 3. Half of our benchmarks are graph kernels, while the other half are streaming computations. All but two are from standard benchmark suites–either CRONO [25] for the graph kernels or Rodinia [26] for the streaming computations. The exceptions are CC, which identifies connected components using the Awerbuch-Shiloach [27] algorithm and is part of a DARPA streaming graph challenge problem, and daxpy, which computes $y[i] = a * x[i] + y[i]$. Both CC and daxpy were written by the authors.

All of the kernels were explicitly parallelized to create threads for multiple cores. The threaded code was then vectorized by hand using AVX-512 intrinsics to generate SIMD instructions for vector units. We found there is ample vector parallelism in our benchmarks, leading to significant memory parallelism. Specifically, unit-stride array traversals occur in all of the benchmarks, and are opportunities for packed vector load/store instructions. In the graph kernels, memory indirection through adjacency or edge lists are ubiquitous, and are opportunities for scatter-gather memory instructions. (AVX-512 allows masking which can dynamically disable portions of individual scatter-gather operations, thus supporting traversal of variable numbers of edges). There are also opportunities for scatter-gather in one of the streaming benchmarks, NN. (We will discuss this in Section 5).

The second column of Table 3 specifies the inputs for each benchmark. For the graph kernels, we created an input graph using SSCA2 [28] which is based on the Recursive MATrix (R-MAT) scale-free graph generator [29]. The number of vertices in the generated graph is 2^{22} (*i.e.*, scale=22). The input graph exhibits a power law degree distribution with community structure, and resembles social network graphs. For the Rodinia benchmarks, the input sizes in Table 3 are scaled-up versions of the inputs provided with the benchmarks [26]. This was necessary to accommodate the 2000 thread and 8-way SIMD parallelism used in our experiments.

We functionally execute the serial initialization portions of each benchmark in SDE only, and then turn on the cycle-accurate back-end during the parallel region. The last column of Table 3 reports the number of instructions (in billions) simulated in the parallel region. For cc, daxpy, nn, and pf, these represent the entire parallel region. For the other 6 benchmarks, the instruction counts represent a portion of the parallel region. All of the benchmarks exhibit the same behavior throughout the parallel region, with the exception of cc (which is why it was one of the benchmarks run to completion). So, we expect the simulation windows for the 6 partially simulated benchmarks to be representative.

5. PERFORMANCE EVALUATION

5.1 Overall Results

Figure 8 presents our main performance results. The graph kernels are shown on the left side of the figure, while the streaming computations are shown on the right side. For each benchmark, we plot the throughput achieved on a monolithic computer with either 200ns or 500ns read latencies (write latencies are double that), as well as the throughput achieved on a conventional tiled CPU with a DRAM memory system. These are labeled ReRAM-200, ReRAM-500, and DRAM, respectively, in Figure 8. Not only do we simulate the tiled CPU with 2000 threads specified in Table 2, we also simulate tiled CPUs running 200, 400, and 1200 threads. For 400 and 1200 threads, we utilize all 400 cores, but run with 1 and 3 threads (instead of 5 threads) per core. For 200 threads, we utilize only half the cores–*i.e.*, every other core in a "checker pattern" across the 2D mesh. All bars are normalized against the DRAM bars with 200 threads.

At 200 threads, the conventional tiled CPU outperforms monolithic computers. Averaged across five streaming benchmarks, DRAM throughput is 4.5x higher than ReRAM-500 throughput, and 2.3x higher than ReRAM-200 throughput. The conventional tiled CPU's advantage at 200 threads also extends to the graph kernels. Averaged across five graph kernels, DRAM throughput is 2.2x higher than ReRAM-500 throughput. The only time when monolithic computers come out on top at 200 threads is in the ReRAM-200 bars for the graph kernels. With 200ns ReRAM, the graph kernels achieve slightly higher throughput on the monolithic computer, about 4%, compared to the conventional tiled CPU.

In these experiments, there is little to no contention in the memory system (for both DRAM and ReRAM) because the thread counts are not high enough to saturate the memory-level parallelism. When contention is low, memory access latency is the main determiner of performance. DRAM holds a significant advantage over ReRAM in terms of access latency, hence its superior performance at 200 threads.

But the situation changes as we increase thread count. That change, though, is different for the graph kernels versus the streaming computations. In the graph kernels, monolithic computer performance increases dramatically as we scale parallelism, far faster than the traditional tiled CPU's performance which appears almost flat-lined in comparison. Even at 400 threads, the left side of Figure 8 shows monolithic computers over-take the conventional tiled CPU in many cases. At 2000 threads, the monolithic computer is far superior in all cases–up to 3.1x faster assuming 500ns ReRAM, and up to 7.5x faster assuming 200ns ReRAM (for sssp). On average, the monolithic computer's throughput is 2.0x and 4.7x higher than the conventional tiled CPU assuming 500ns and 200ns ReRAM, respectively, for the graph kernels.

The vast majority of cache misses for graphs is incurred by indirect memory references that are supported through scatter-gather operations. The combination of 2000 threads and 8-way scatter-gather generates an enormous number of sparse (8-byte) memory requests. The DRAM memory system is incapable of handling this parallelism. It only has 128 banks, and worse, fetches 128-byte cache blocks on each sparse request, consuming bandwidth to transfer mostly unused data. This leads to extreme contention at the DRAM's banks. In contrast, our monolithic computer uses the mechanisms from Sections 3.1.2 and 3.3 to adapt the access granularity, employing fine-grain fetches for the scatter-gather memory requests. This opens up the maximum 16,000-way bank-level parallelism to support the indirect memory references. Despite the massive number of scatter-gather memory requests, the ReRAM-based memory system is capable of keeping up without incurring significant bank contention. This enables the highly scalable monolithic computer performance visible in Figure 8 for the graph kernels.

Figure 9 further illustrates this difference in scalability. The figure plots the average round-trip read latency–*i.e.*, from the time a read is initiated by a core until it returns from the memory system–for the graph kernels from Figure 8. These results show latency always increases as we scale thread count. But there is a major difference between the monolithic computer and the conventional tiled CPU. For the ReRAM-500 and ReRAM-200 bars, the increase is noticeable, almost doubling in some cases from 200 to 2000 threads. However, the increase for the DRAM bars grows much faster. By 2000 threads, DRAM's round-trip latency shoots up to 2137 cycles on average, and as much as 3972 cycles for sssp. This is due to queueing and serialization at the DRAM controllers. The queueing delays in the monolithic computer are far smaller thanks to the ReRAM's massive parallelism. When including these queueing delays, DRAM's overall latency is much higher than ReRAM despite having a lower baseline access latency, illustrating the point in Section 2.3.

For the streaming computations on the right side of Figure 8, monolithic computer performance also increases as parallelism is scaled. But instead of surpassing the performance of conventional tiled CPUs, as is the case for the graph kernels, the monolithic computer simply approaches it and at best achieves parity. (NN is an exception which we will discuss shortly). Most of the streaming computations perform dense traversals of large linear arrays. Our monolithic computer adapts the access granularity to support these streaming memory operations, favoring large cache-block accesses over the fine-grained fetches suited for the graph kernels. This reduces the amount of memory parallelism supported across the on-die ReRAM. Rather than request parallelism, the most important characteristic becomes total data bandwidth, which makes DRAM much more competitive with ReRAM.

In fact, with 500ns ReRAM, our monolithic computer delivers a peak bandwidth of 256 GB/sec (16,000 banks $\times$ 8 bytes / 500ns), which is only half the peak bandwidth of the DRAM memory system. With 200ns ReRAM, the monolithic computer improves to 640 GB/sec. Unfortunately, these

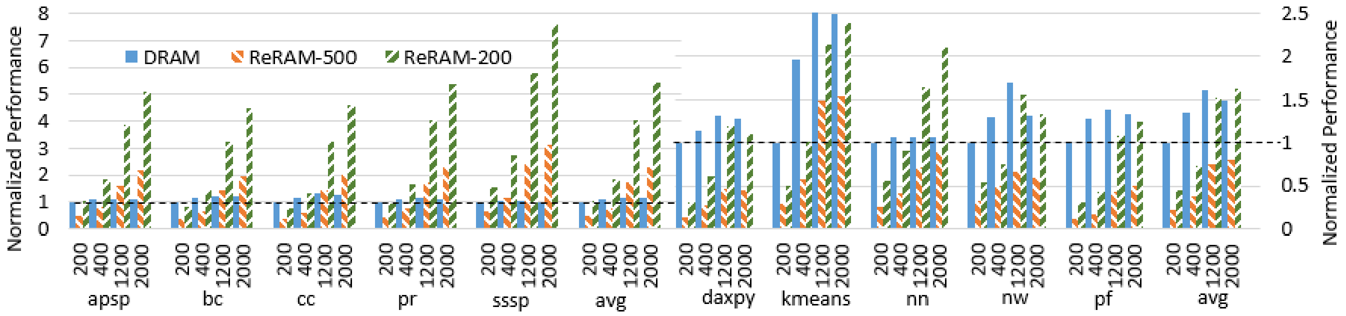

Figure 8: Normalized throughput on monolithic computers with 200ns or 500ns read latencies, and on high-performance DRAM. Results are shown for 200, 400, 1200, and 2000 threads.

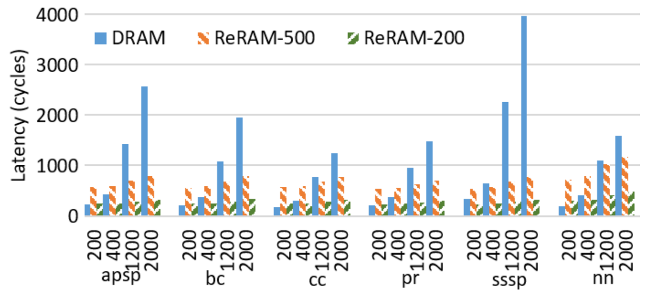

Figure 9: Average round-trip latency for benchmarks with fine-grain accesses.

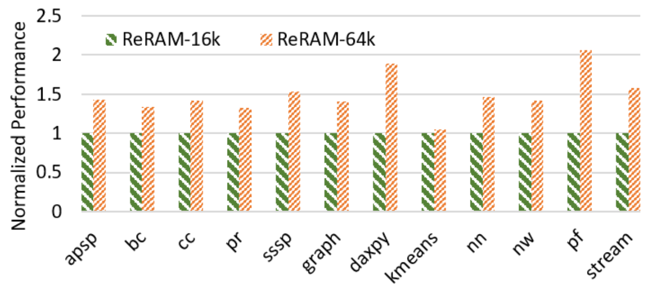

Figure 10: Performance gain from increasing the number of ReRAM banks from 16K to 64K.

data rates are only for reads. Because writes are twice as slow, the effective data bandwidth can be considerably lower. (Writes don't usually stall the CPU, but they occupy banks that can delay reads which do stall the CPU). As Figure 8 shows, monolithic computers are always slower than the conventional tiled CPU on the streaming computations (though it is close at 2000 threads with 200ns ReRAM). For streaming benchmarks other than NN, the monolithic computer is 48.1% as fast as the tiled CPU for ReRAM-500, and almost achieves parity with 94.5% of the performance for ReRAM-200 at 2000 threads.

While the Rodinia benchmarks are dominated by dense streaming computations, one exception is NN. Instead of unit-stride array traversals, the main array in NN is traversed with a large stride. In each 64-byte cache block that is fetched during this traversal, only 16 bytes or 25% of the data is referenced. We used a gather memory operation to fetch this sparse data precisely. This enables NN to achieve better performance than the other streaming computations, though not as good as the graph kernels. At 2000 threads, the ReRAM-200 bars show a performance advantage of over 2.0x for monolithic computers compared to the conventional tiled CPU. Even for the ReRAM-500 bars, the monolithic computer almost matches the conventional tiled CPU.

Like the graph kernels, NN benefits from the higher parallelism that ReRAM affords sparse accesses. This is confirmed by the last set of bars in Figure 9 which show the average access latency for NN on ReRAM is much better than on DRAM, similar to the behavior in the graph kernels. But NN is not as sparse as the graph kernels where each cache block usually brings in only 8 bytes of useful data. For DRAM that employs 128-byte cache blocks, this results in a fetch efficiency of only 6.25% for graphs as compared to 25% for NN.

5.2 Technology Scaling Impact

In the monolithic computer experiments from Section 5.1, we have assumed 16,000 ReRAM banks in the memory system, which is our estimate for the 16nm technology node (see Section 3.1.2). This is a lot of memory parallelism. But, ReRAM is also highly scalable. In future technology nodes (*e.g.*, 7nm), it is conceivable to have up to 64,000 banks. An important question is how might technology scaling impact our performance results?

Figure 10 reports the throughput on a monolithic computer when the number of ReRAM banks is increased to 64,000. For each benchmark, the scaled-up performance is shown in the bars labeled ReRAM-64K, which are plotted next to the baseline configuration in the bars labeled ReRAM-16K. For all of the experiments, we run the benchmarks with 2000 threads, and employ a read latency of 200ns (and a write latency of 400ns). So, the ReRAM-16K bars in Figure 10 are identical to the last set of ReRAM-200 bars for each benchmark in Figure 8. All bars in Figure 10 are normalized to the ReRAM-16K bars.

As Figure 10 shows, scaling up the number of ReRAM banks has a positive impact on performance across the board. The reason for the improvement, though, depends on the type of benchmark. For the graph kernels, performance improves primarily because of a reduction in bank conflicts. Even though the baseline configuration provides 16,000 ReRAM banks, the tiled CPU we assume is capable of completely saturating all of these ReRAM banks, especially when using scatter-gather memory operations. We find the scatter-gather requests in the graph kernels are uncorrelated, so they tend to spread across all of the banks. Nevertheless, the sheer

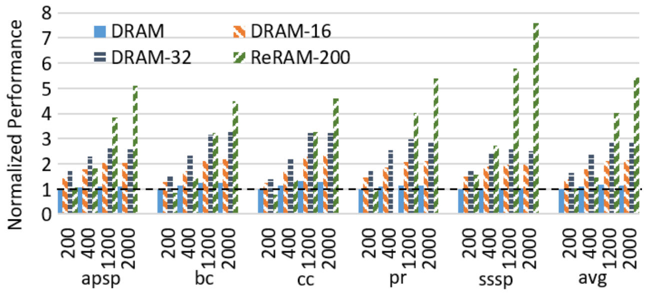

Figure 11: Performance with larger numbers of narrower DRAM channels.

volume of memory requests means there still exists a significant probability for accesses to collide. The left half of Figure 10 confirms this: when the number of banks is increased from 16,000 to 64,000, the graph kernels' performance improves by 40.8% on average.

In contrast, for the streaming computations, performance improves primarily because of an increase in total memory bandwidth. As described above, the peak read bandwidth for the baseline configuration is 640 GB/s (and is lower with writes factored in). When the number of banks increases to 64,000, this peak read bandwidth quadruples to over 2 TB/s. The streaming computations running on our tiled CPU cannot utilize this amount of memory bandwidth, so we don't expect performance to quadruple. As the right half of Figure 10 shows, the streaming computations exhibit a 57.6% improvement in performance on the scaled-up configuration. Compared to the DRAM bars in Figure 8, monolithic computers can outperform conventional tiled CPUs by 66.0% on the streaming computations if given 64,000 banks. Kmeans, being less memory intensive, does not utilize all the bandwidth in the 16,000 bank case, and therefore does not increase in performance when the banks are scaled up.

5.3 DRAM Channel Scaling

In the conventional tiled CPU experiments from Section 5.1, we have assumed a high-performance DRAM memory system resembling HBM DRAM. Specifically, we modeled 8 memory controllers attached to 8 channels of DRAM die stacks, each providing 128 bytes of data per memory request. In HBM DRAM, however, the controllers are implemented outside the DRAM die stacks, providing flexibility to have different organizations. In particular, the internal DRAM banks in the die stacks are narrower than the 128-byte interface we assumed at the controllers. It is possible to have a larger number of narrower channels, at the expense of having more memory controllers. This provides greater request parallelism and supports a finer access granularity, both of which can benefit the graph kernels.

In addition to the baseline 8 $\times$ 128-byte channels, we also simulated 16 $\times$ 64-byte channels (16 memory controllers attached to 16 channels, each providing 64 bytes per memory request), and 32 $\times$ 32-byte channels (32 memory controllers attached to 32 channels, each providing 32 bytes per memory request). In each case, we changed the cache block size to match the request size, but kept cache sizes the same. Notice, total memory bandwidth doesn't change across all three configurations (512 GB/s)–only request parallelism changes.

Figure 11 reports the performance of the different DRAM configurations for the graph kernels. (The streaming computations perform best on the baseline configuration, so we omit those results). Four bars are plotted for each benchmark. The first set of bars, labeled DRAM, represent the baseline 8 $\times$ 128-byte configuration and is identical to the corresponding bars in Figure 8. The next two sets of bars, labeled DRAM-16 and DRAM-32, represent the 16 $\times$ 64-byte and 32 $\times$ 32-byte configurations, respectively. These DRAM configurations are compared against our monolithic computer, labeled ReRAM-200, which is identical to the corresponding bars in Figure 8. All bars are normalized to the DRAM bars with 200 threads.

As Figure 11 shows, the scalability of the conventional tiled CPU running the graph kernels indeed improves as the number of DRAM controllers increases. Although the 8-controller bars are flat-lined, the DRAM-16 and DRAM-32 bars continue to increase with thread count. By 2000 threads, on average, DRAM-16 is 1.9x better than the baseline DRAM configuration while DRAM-32 is 2.5x better. Nevertheless, our monolithic computer still exhibits better scalability. At 2000 threads, we are still 2.6x better than DRAM-16 and 87% better than DRAM-32, averaged across the five graph kernels.

5.4 Memory Capacity

It is important to note our experiments make assumptions about memory capacity that heavily favor DRAM. In particular, we have not penalized DRAM for its much lower capacity. While in-package DRAMs like HBM exhibit extremely high memory bandwidths, they also have very low capacities, in the 16–32 GB range, whereas ReRAM can achieve many 100s of GBs. Given larger inputs that don't fit in high-speed DRAM, a conventional tiled CPU would be forced to access off-package DRAM (*e.g.*, DDR4) at far lower bandwidths. If we were to perform those experiments, monolithic computers would hold a much more significant performance advantage over DRAM than what we report for the streaming computations. And, the gains we report for the graph kernels would also increase.

6. CONCLUSION

Researchers have forecasted the existence of monolithic computers before [1, 2]. But to our knowledge, this paper is the first to provide a detailed design of a monolithic computer, and to perform a simulation-based evaluation of its capabilities. We integrate Crossbar ReRAM over CPU logic via BEOL processing steps in standard CMOS, enabling a large amount of main memory to be placed right on top of the CPU's cores. Our physical design study estimates that half the CPU die could be feasibly covered with ReRAM. This would enable a large number of ReRAM sub-arrays to be integrated with the CPU, each providing a fine access size. We propose to implement a variable granularity memory system on top of these ReRAM sub-arrays that supports both a fine-grain 8-byte access (to maximize bank-level parallelism for irregular access patterns) and a traditional cache-block access (to efficiently support streaming access patterns). To sustain the request rates that our memory system can support, we investigate a tiled accelerator-like CPU with simple in-order cores, multithreading, and wide SIMD capable of scatter-gather memory operations. Our results show that such a monolithic computer architecture can provide unprecedented performance on irregular computations such as graph kernels, outperforming conventional DRAM-based systems by 4.7x. Assuming a 16nm technology node, monolithic computers will merely achieve parity with conventional systems on streaming computations due to the high memory bandwidths that DRAM can attain for regular memory access patterns. But at a 7nm technology node, our results show monolithic computers could outperform DRAM-based systems by 66% on streaming computations.

7. REFERENCES

[1] M. M. S. Aly, M. Gao, G. Hills, C.-S. Lee, G. Pitner, M. M. Shulaker, T. F. Wu, M. Asheghi, J. Bokor, F. Franchetti, K. E. Goodson, C. Kozyrakis, I. Markov, K. Olukotun, L. Pileggi, E. Pop, J. Rabaey, C. Re, H.-S. P. Wong, and S. Mitra, "Energy-Efficient Abundant-Data Computing: The N3XT 1,000x," *Computer*, December 2015.

[2] M. M. Shulaker, T. F. Wu, M. M. Sabry, H. Wei, H.-S. P. Wong, and S. Mitra, "Monolithic 3D Integration: A Path from Concept to Reality," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[3] Intel, "Intel Optane Technology." 2017.

[4] Crossbar, "ReRAM Memory, Crossbar." 2017.

[5] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff, "Tile Processor: Embedded Multicore for Networking and Multimedia," in *Proceedings of the 19th Symposium on High Performance Chips*, (Starford, CA, USA), 2007.

[6] Y. Hoskote, S. Vangal, S. Dighe, N. Borkar, and S. Borkar, "Teraflop Prototype Processor with 80 Cores," in *Hot Chips*, 2007.

[7] Intel, "AVX 512 Instruction Extensions, http://software.intel.com/en-us/blogs/2013/avx-512-instructions." 2017.

[8] Intel, "Intel Xeon Phi Product Family, http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html."
2014.

[9] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator," in *Proceedings of the International Symposium on Computer Architecture*, (Austin, TX), pp. 140–151, June 2009.

[10] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 2–13, ACM, June 1995.

[11] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, (Chicago, IL), IEEE, April 1994.

[12] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 241–251, June 1997.

[13] L. O. Chua, "Memristor—the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[14] S. H. Jo, K.-H. Kim, and W. Lu, "High-Density Cross-bar Arrays Based on a Si Memristive System," *Nano Letters*, 2009.

[15] S. H. Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian, "3D-stackable crossbar resistive memory based on Field Assisted Superlinear Threshold (FAST) selector," *IEEE International Elec-tron Devices Meeting*, 2014.

[16] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, and Disk*. Morgain Kaufmann, 2007.

[17] P. Siegl, R. Buchty, and M. Berekovic, "A bandwidth accurate, flexible and rapid simulating multi-hmc modelling tool," in *Proceedings of the Third International Symposium on Memory Systems, MEMSYS 2017, Washington, DC, USA, October 2-5, 2017*, pp. 71–82, ACM, Oct 2017.

[18] D. Burger, "Hardware Techniques to Improve the Performance of the Processor/Memory Interface," tech. rep., University of Wisconsin-Madison, December 1998.

[19] S. Kumar and C. Wilkerson, "Exploiting Spatial Locality in Data Caches using Spatial Footprints," in *Proceedings of the 25th International Symposium on Computer Architecture*, (Barcelona, Spain), June 1998.

[20] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An Evaluation of High-Level Mechanistic Core Models," *ACM Transactions on Architecture and Code Optimization*, April 2014.

[21] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, January 2010.

[22] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th International Symposium on Computer Architecture*, (Tel-Aviv, Israel), June 2013.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[24] Intel, "Intel Software Development Emulator, http://software.intel.com/en-us/articles/intel-software-development-emulator."
2012.

[25] M. Ahmad, F. Jijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, (Atlanta, GA), October 2015.

[26] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, (Austin, TX), October 2009.

[27] B. Awerbuch and Y. Shiloach, "New Connectivity and MSF Algorithms for Ultracomputer and PRAM," in *Proceedings of the International Conference on Parallel Processing*, August 1983.

[28] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse, "HPCS Scalable Synthetic Compact Applications #2 Graph Analysis." August 2006.

[29] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Minig," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004.