Scalable, Programmable and Dense: The HammerBlade Open-Source RISC-V Manycore

Dai Cheol Jung[1], Max Ruttenberg[1], Paul Gao[1], Scott Davidson[1], Daniel Petrisko[1], Kangli Li[1], Aditya K Kamath[1], Lin Cheng[2], Shaolin Xie[1], Peitian Pan[2], Zhongyuan Zhao[2], Zichao Yue[2], Bandhav Veluri[1], Sripathi Muralitharan[1], Adrian Sampson[2], Andrew Lumsdaine[1,3], Zhiru Zhang[2], Christopher Batten[2], Mark Oskin[1], Dustin Richmond[4], Michael Bedford Taylor[1]

[1]University of Washington, [2]Cornell University, [3]PNNL, [4]University of California, Santa Cruz

***Abstract*—Existing tiled manycore architectures propose to convert abundant silicon resources into general-purpose parallel processors with unmatched computational density and programmability. However, as we approach 100K cores in one chip, conventional manycore architectures struggle to navigate three key axes: scalability, programmability, and density. Many manycores sacrifice programmability for density; or scalability for programmability. In this paper, we explore HammerBlade, which simultaneously achieves scalability, programmability and density. HammerBlade is a fully open-source RISC-V manycore architecture, which has been silicon-validated with a 2048-core ASIC implementation using a 14/16nm process. We evaluate the system using a suite of parallel benchmarks that captures a broad spectrum of computation and communication patterns.**

***Index Terms*—manycore architecture, parallel programming, open-source hardware, RISC-V**

I. Motivation

Tiled manycore architectures propose to convert abundant silicon resources into parallel processors with unmatched computational density and programmability. Building such a processor involves stamping out identically shaped tiled processors interconnected by a Network-on-Chip (NoC). Due to their simplicity, tiled multicores can be designed, implemented and verified by a small engineering team, lowering the time and cost to deploy a new system [5], [43]. Manycore architecture resonates with a growing demand for flexible, rather than specialized, parallel hardware to accelerate innovative ideas in next-gen AI and ML [18], [25], [29], [32], [39], and in other emerging domains.

Modern manycore chips already pack hundreds, and occasionally more than a thousand cores into a single die [21], [27], [55]. Even with modest projected technology scaling over the next few years, 100K+ cores will fit on a full-reticle chip. However, general-purpose parallel manycore chips were conceived when 100 cores seemed far off, and scalability by another 1,000X is not assured. Current manycore architectures contain built-in scalability, programmability, and density limiters that must be re-examined and refined to make the next leap. This paper illuminates a path to 100K cores.

We outline key limitations of existing manycore designs:

Density. Many mechanisms in multicore designs are inherited directly from multi-chip processors, that were constructed by gluing together expensive single-core optimized chips.

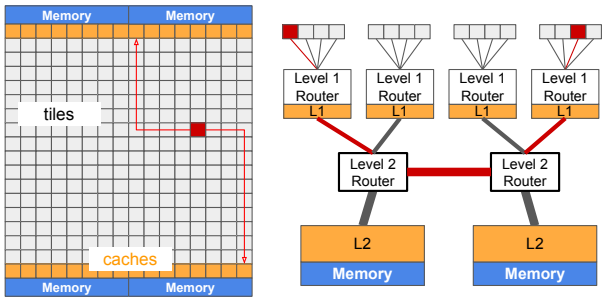

Fig. 1: **Resource organization in conventional manycore architectures**. High-diameter network topology in flat manycore is prone to massive network congestion. Hierarchical manycore provides fast communication between cores within a cluster, but the communication between clusters is complicated by the multi-level cache and hierarchical network topology.

These single-core chips focused on peak-performance, and multiprocessor support was a value-added feature that brought a high price and justified die area increases even if relatively large. Now that a chip holds many cores, total raw compute is inversely proportional to the area (or power) used by the compute tile, since it determines how many cores fit on-die.

Stripping down the cores so more of them fit has become the prime directive. Large private L2 caches are the easiest to let go, since the geomean performance impact of halving cache size is often a small percentage, and the resulting core count increase high. Beyond "right-sizing" the memory system, there are two other classes of optimizations: first, replacing expensive sequential processor mechanisms with cheaper ones, or throwing them out altogether; and second, revisiting parallel programming mechanisms we have for concepts like memory-level parallelism, synchronization, and communication, and replacing with more parsimonious yet equally effective ones. In this paper, we show how all three can be optimized.

Scalability. There are many kinds of scalability; and tiled manycores usually exhibit *physical scalability*; i.e. the ability to scale without frequency being disproportionately affected by growing logic depths or wire lengths. But *logical scalability* describes the degree to which resource consumption and

latencies of basic operations are controlled as core count grows. The first *flat manycores*, shown in Figure 1a, generally posited a large uniform tile array interconnected by 2-D mesh; with memory or I/O interfaces on the edge of the chip. For nearest-neighbor communication, this scales quite effectively, but for random/non-local communication, or all-to-edge communication as typical in DRAM accesses, the network quickly fills up, because each message must occupy $O(sqrt(N))$ cycle-hops of resources and time. In an $N \times N$ mesh, each tile can only inject packets at the average rate of $2/N$ per cycle before edge network channels become completely saturated [50]. And interestingly, mesh networks tend to underutilize available wiring bandwidth unless they assume very wide packets and thus block level memory accesses. *Hierarchical manycores*, shown in Figure 1b, a more recent phenomenon, seek to address this logical scalability problem by comprising clusters of cores in a rigid network hierarchy. Cores in the clusters are often connected with low-latency crossbars, designed to share some common resources. These clusters are then connected by the network in another level of hierarchy, usually with wider channel width. This works well for transferring cache-line sized data, but is inefficient for fine-grained, random access, which is common in graph and sparse data. Sharing data between clusters requires coordination of L1 and L2 caches, which adds cache-coherency complexities and overheads. Paradoxically, clustering reduces cross-network bandwidth, and to address this bandwidth shortage, will require larger ops per byte computational intensity at the node level; resulting in larger local memory, larger tile sizes, and reduced density.

Parallel Programmability. Parallel architecture choices that attempt to optimize density and scalability often come at a cost to parallel programmability. GPUs, for example, scale to reticle sizes using Streaming Multiprocessors (SM), but those SMs have extremely restricted communication between each other. GPUs seek to attain density through SIMT-abstracted SIMD units, which are infamous for their performance problems when memory accesses or control flow diverges. Message passing architectures are plagued by area-overheads of receive queues, challenges with deadlock and difficulty in expressing computations efficiently that pull rather than push data.

These programmability constraints, caused by density/scalability compromises in the architecture, add many people-years of effort to program, and even may lead the programmer to select suboptimal algorithms that happen to work within the constraints. This research shows that with the right combination of architectural techniques, all three can be attained – scalability, programmability, and density. The result is an ultra-optimized, more ideal general-purpose parallel architecture enabling programming models that can express a broad range of parallel algorithms in the most performant ways, for both regular and irregular data access and control flow [35].

II. Contributions & Insights

This paper has the following contributions:

- We present **HammerBlade**, a new kind of manycore architecture that explicitly optimizes the three key criteria for realizing 100K-core manycore processors: scalability, programmability, and density.
- We demonstrate the effectiveness of this architecture by drawing on data from our open-source, silicon implementation, which implements 2048 RISC-V cores in a manycore processor using a fraction of the reticle in a 14/16 nm process, and breaks world records for RISC-V performance. The hardware is up and running in our lab.
- We rigorously evaluate the system using a suite of parallel benchmarks that captures a broad spectrum of computation and communication patterns to demonstrate its parallel programmability. Our RTL simulation integrates DRAMSim3 [36] to accurately model a modern High-bandwidth Memory (HBM) 2.0 [26]. We analyze the hardware resource utilization and discuss the bottlenecks observed in each kernel (Figure 11).
- Using the post-APR gate-level power analysis, we demonstrate that our area-optimized RISC-V cores are 3.6–15.1$\times$ more energy-efficient than a prior leading manycore chip on a per-instruction, process-normalized basis (Figure 13).

This paper makes the following key insights:

- Relative to prior manycore designs, *aggressive densification* – i.e. focusing on more area-efficient realizations of single-threaded execution, parallel coordination mechanisms, and memory resources – stands out as a key contributor of performance uplift (Figure 10, 16).
- HammerBlade's Cellular Manycore approach provides an effective alternative to flat and hierarchical manycores, diffusing cache banks into a large manycore array. It provides (1) constant bandwidth and latency even as the system scales, (2) explicit optimization of data and computation placement, (3) chip-wide PGAS-style programming in a global memory system, and (4) coherence without need for coherence hardware.
- Ruche Networks [30] allow for increased logical scaling, enabling larger collections of cores to co-communicate with larger collections of cache banks without being impacted by bandwidth or latency (Figure 15), handily beating 2-D meshes (Figure 14).
- Networks oriented around single-word data messages can be implemented efficiently and are more effective at transferring high quantities of sparse data across clusters of cores than block-level messages that are endemic to hierarchical or mesh manycores (Figure 3, 16).

The rest of the paper is organized as follows: Section III describes the HammerBlade (HB) architecture. Section IV describes the programming model and the parallel kernels used for evaluation. Section V evaluates the key architectural optimizations, and explores different scaling strategies. Section VI surveys the related work.

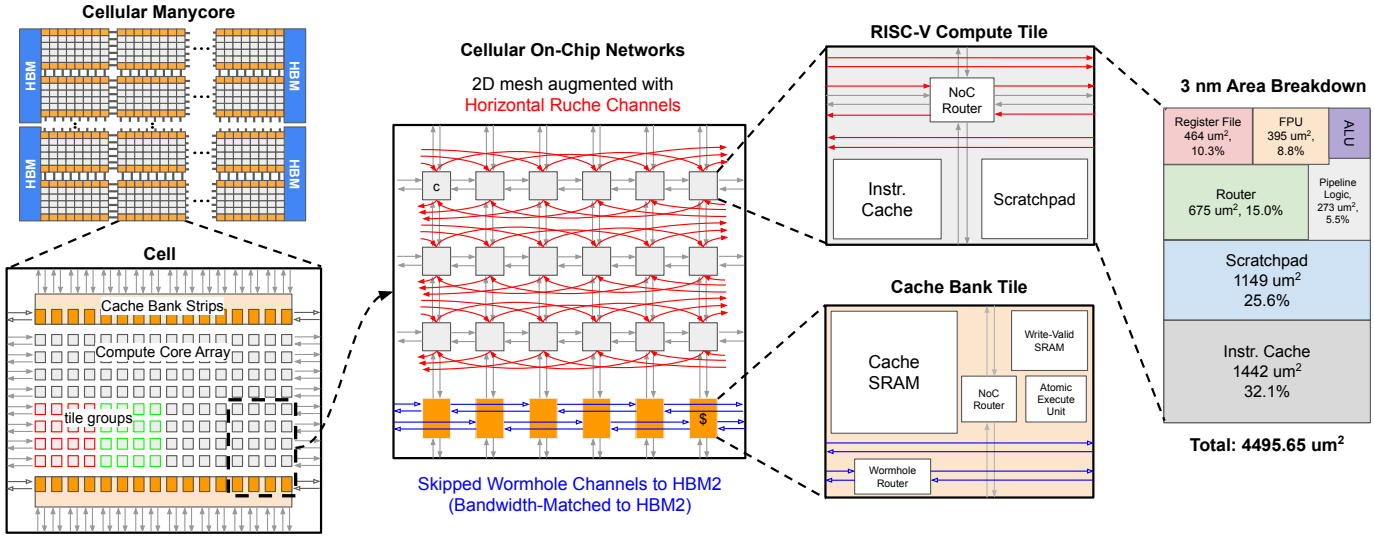

Fig. 2: **Overview of HB Cellular Manycore Architecture.** HB synthesizes a set of novel features that enable maximum scalability and density while maintaining programmable SPMD/PGAS abstractions. Cache banks are embedded in a homogeneously networked array of compute tiles, and associated with Cells, groups of nearby tiles, to provide locality as the system scales. The Ruche Networks provide 4× the bisection bandwidth over meshes and enable larger Cell sizes. The right side of the figure shows the area breakdown of a single HB tile scaled down to the 3 nm node [61]. With the tile area of 4496 um^2, 100K+ cores can fit on a 600 mm^2 die, orders of magnitude beyond any previous manycore designs in Section VI.

III. Architecture

This section describes the architectural features and mechanism of the HammerBlade Manycore.

A. The HammerBlade (HB) Cellular Manycore Architecture

In the broader picture, we envision that general-purpose chips of the future (and indeed the mobile phone chips of today) will have three classes of computational blocks, as envisioned by the tiered accelerator fabric (TAF) architectural pattern [19]. These chips will have general-purpose Linux-capable cores for hosting the OS, a manycore for general purpose parallel computation, and one or more highly specialized accelerators, such as tensor cores for ML. Pulling the OS-capable hardware, and support for single-threaded OoO execution out of the manycore array, and into separate Linux-capable cores is the first step of densification, eliminating the need to have privileged-mode and OS-support specific hardware as well as hefty BTB's, caches and reorder buffers in every tile in the manycore array. This paper focuses on the design and analysis of the manycore part of these systems.

HammerBlade, as is to be expected of a tiled manycore, has a physical design hierarchy which are the units of replication that allow the chip to be efficiently composed by repeatedly stamping out VLSI layout. In order to scale to large number of cores in reasonable CAD flow time, HammerBlade is assembled by VLSI CAD tools hierarchically as shown in the build-up diagram in Figure 2, using compute and cache bank blocks, a Cell block that replicates the compute and cache block, and then a top-level block which replicates the Cell block and off-chip I/O and memory resources. HammerBlade's logical hierarchy (i.e. the architectural or programmer's view), on the other hand, allows the programmer to name resources at different scales (i.e. within a tile group, within a cell, across a group of cells, and across the chip), facilitating both logical scalability (e.g., tile groups in a Cell can use load and store instructions to communicate through a contiguous region of address space that is striped across nearby cache banks to avoid the scalability challenges of flat manycores) and PGAS (Partitioned Global Address Space) simultaneously.

A HB Cell is a 2-D array of compute tiles and two 1-D arrays of cache banks, as depicted in Figure 2. Tiles and caches are all interconnected with the Ruche Network [30], [45], a 2-D mesh augmented with uniform, long-range links that pass through tiles. These extra links increase bisection bandwidth and reduce the network diameter without disrupting the mesh's ability to map easily onto a silicon substrate. As the distance of these links grows, more wires pass through the tiles. This can effectively utilize all VLSI wiring resources, which are largely unused by 2-D mesh-based architectures, without adding much hardware overhead. In HB implementation, the Ruche links that skip three tiles horizontally boost the peak bisection bandwidth by 4× greater than 2-D mesh. This increases the router area by 40%, and the overall tile area by 4%.

HB has a flat, uniform network hierarchy, so the Ruche Network extends beyond the Cell boundary to make connections with adjacent Cells. This allows network packets to reach any location on the chip without changing the packet format. By doing so, we avoid additional hardware for de/serialization and arbitrary network bottleneck. By software default, the cores within the same Cell work together on a common problem, using the Cell's shared cache banks. Later, we describe how PGAS is set up for this affinity (Section IV).

The HB cache hierarchy is also flat. Cache banks are the last-level cache to DRAM. Each cache bank is independent and mapped to an exclusive range of DRAM address space to avoid coherence problem. Cache banks implement a *write-validate* policy [28], as used in most GPUs [31]. This essentially eliminates unnecessary DRAM reads for cache write misses, which is helpful in typical workloads that write out results in large blocks. Cache banks are also *non-blocking*, and provide enough buffering to ensure that primary and secondary misses are drained out of the network to allow other hit requests to proceed. All miss status holding registers (MSHR) are consolidated in the last level of cache hierarchy to be shared by all tiles for better utilization, rather than scattered across the hierarchy. RISC-V cores can remotely perform atomic operations (e.g. atomic-swap, OR, add) on these cache banks with acquire/release semantics [59], which enables many synchronization primitives necessary for parallel programmability.

A strip of cache banks includes the 1-D wormhole flow-controlled channels, which are only used for the process of cache refill and eviction. As shown in Figure 2, each strip of cache banks contains multiple pairs of *skipped* channels, which improve fairness and latency for cache banks in the middle. The skip distance and the channel width can be adjusted to match the HBM2 bandwidth.

B. HB RISC-V Core

Each HB core is a high-frequency, area-optimized, single-issue, in-order, 5-stage, RISC-V processor with atomic and floating-point extensions, enabling SPMD execution. Each core contains 4 KB scratchpad memory (SPM) and 4 KB instruction cache (icache). Smaller SRAMs enable more cores but lower bit densities. We select the size that maximizes cores under the constraint of sufficient size to hold sizeable kernel inner loops. This is one of the major sources of the improved compute density. SPMD's independent program execution is its major advantage over SIMT, where every thread must execute in lockstep. HB cores try to minimize the associated cost in its frontend logic. The core implements a simple branch predictor with a 2-cycle miss penalty. The core predicts 'taken' for a backward branch and 'not taken' for a forward branch, which is sufficient for a wide range of data-parallel kernels. The icache is direct-mapped with 4-instruction cache line and 12-bit tags. This provides 16 MB of program space, which is practically 'unlimited' for many data-parallel kernels. The tag bits and instructions are combined in a single SRAM block with only 25% area increase over 4 KB IMEM used in Celerity [19]. When branch and jump instructions are stored in the icache, the lower part of the target address is pre-computed and stored in the immediate field, effectively serving as a zero-area branch target buffer.

HB provides more efficient mechanisms for MLP than other throughput-oriented architectures. HB cores are *explicitly interfaced* with the on-chip network, where each individual RISC-V memory operation addressing remote resources turns into a network packet. In most architectures, data transfer is implicitly done by cache refill and evict mechanism at block-level granularity. For fine-grained accesses, this may waste bandwidth resources by moving unwanted data as well. HB remote memory operations are *non-blocking*, enabled by a bit-vector styled scoreboard, which costs less than 1% of the tile area. GPU multithreading, on the other hand, requires a massive register file to keep every thread context readily available, which amounts to a significant portion of GPU area and energy [33]. A single HB tile can launch up to 63 outstanding requests, each of which may potentially generate a cache miss and a DRAM request in the shared LLC banks and provide an ample source of MLP. Remote loads can be pipelined in the network to create a load-use distance and hide latency (Figure 7). Unlike in HB, SIMD execution models create a restriction that every memory request generated by an instruction must complete before moving on to a next instruction. Uncoalesced accesses in GPU have to be replayed for multiple cycles, which further degrades the pipeline utilization [14], [44].

C. HB Network-on-Chip

HB's simplified NoC design plays a crucial role in maximizing compute density and network scalability. It is a major departure from the previous manycore designs, which often integrate multiple NoCs for various traffic types. Epiphany-V tiles [42] include routers for not only the on-chip traffic, but also the off-chip (DRAM) traffic. HB's separation of the off-chip routers from the compute tile keeps the tile area low. Raw [58] and Tilera [12] implement fast register-to-register scalar transfer networks, but their routing rules need to be pre-configured at compile-time, using a specialized compiler [34]. Raw adds a dynamic wormhole-routed message network, but it is prone to deadlock, so an extra identical network is added for deadlock recovery [56]. Tilera provides dedicated networks used only for I/O and kernel-level traffic to guard it from the user code [60]. Since most workloads may not make use of all these various networks, there are always some unutilized network resources. HB addresses these issues with the NoC design that provides only the fundamental functionalities.

HB Global Network All core traffic travels on two physically separate Half-Ruche Networks: one for requests, and the other for responses. Each RISC-V memory operation is injected into the network as a single-flit packet that contains the destination (X,Y) coordinate and the offset address, which is translated from the memory address using the PGAS mapping (described in Section IV). Because the frequently accessed cache banks are placed on the top and bottom sides of the Cell, using static X→Y dimension-ordered routing for the request network and Y→X order for the response network is best for the network throughput [4]. As the size of the Cell grows, horizontal channels that cross the bisection become the bottleneck. Half-Ruche Network [30], which adds additional channels in the horizontal direction, relieves this problem.

Unlike the wide channels that connect the clusters in hierarchical manycores, Cellular manycore's globally uniform network is efficient at transferring sparse, random data between Cells. In HB, the wiring density (e.g. *bit per mm*) is

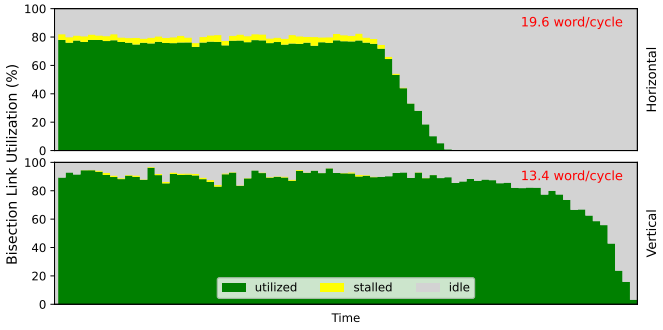

Fig. 3: **The utilization of bisection links between two 16×8 Cells over time, while 1 MB of sparse, random data is being transferred to the adjacent Cell (vertical and horizontal).** Cellular manycore's globally uniform network and *word-access per packet* allow high bandwidth utilization (80∼90%) even for completely sparse, random data transfer between Cells, which is not possible with 1024-bit wide channels used in a representative hierarchical manycore [49].

21.6× higher horizontally and 7.0× higher vertically than the representative hierarchical manycore, which has a 1024-bit wide 2-D mesh network [49]. Figure 3 plots the utilization of bisection links between two Cells, while 1 MB of sparse data is being transferred to the random locations in an adjacent Cell's cache banks (assuming no cache miss).

While *word-access per packet* is very effective for sparse, random data transfer, it can easily overwhelm the network, when every tile tries to read a large sequential block. HB implements *Load Packet Compression* to utilize the on-chip bandwidth more efficiently. When the processor detects consecutive remote loads in the instruction stream that are sequential and destined to the same location in the network, instead of sending four separate packets each with a register ID and an address, it sends one packet with four register IDs and one base address. This combined with the Ruche Network greatly improves the bisection bottleneck, enabling larger Cells.

Hardware (HW) Barrier Compute tiles can synchronize with low latency using the 1-bit wide network with the same Ruche Network topology (Figure 4). HW barrier has two configuration registers: (1) the input directions (if any) from which a tile must wait for the barrier signals before it can send out its own barrier signal, and (2) the output direction to send its barrier signal when the tile joins the barrier. A group of cores can form a tree-like structure [63], where the barrier signals converge at a single root node. Using these configurations, barriers with varying group sizes can be implemented. Once the signal has reached the root, it propagates a wake-up signal back to the leaf nodes. HW barrier costs extremely low area, and, as core count increases, its latency scales much better than the software-implemented barriers.

D. Open-Source Hardware

HammerBlade Manycore[1] is freely available under Solderpad Hardware License [3]. The synthesizable RTL source

[1]Available at https://bsg.ai/hammerblade/

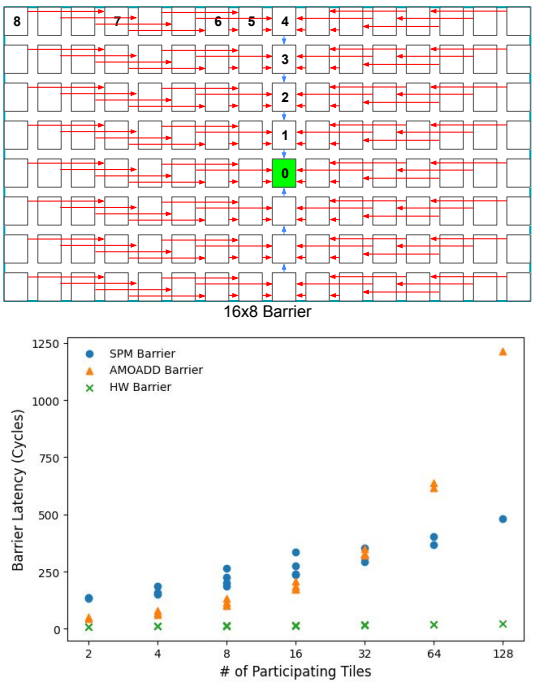

Fig. 4: **Using the Ruche links that skip three tiles horizontally, the barrier signal from the remotest tile can reach the root node in only 8 clock cycles.** HW Barrier is more scalable than the software-implemented barriers, which enables the scalability of larger Cells.

code is written in SystemVerilog with highly configurable and composable interfaces [57]. It has been tested for portability to various commercial and open-source EDA tools. Most importantly, it provides an extensive set of custom performance debugging and visualization tools, which analyze where and why the processors spend most of the time during the kernel execution and the utilization of DRAM, cache, processors, and network routers to identify the bottlenecks in the system.

IV. Parallel Programming Model

This section describes HB's programming model and PGAS mapping that brings logically-defined hierarchy to the flat hardware. We provide the basic programming primitives that higher abstractions can build upon. We introduce the parallel benchmark suite for evaluating the system in Section V.

A. HB Programming Model

HB's programming model is SPMD at the Cell-level, which is more well-suited for irregular data access and control flow. A tile array in a Cell can be logically divided into smaller rectangular groups, called *tile groups* (Figure 2), for finer-grained thread management than in SIMT architecture, where threads are managed in a coarse-grained unit (e.g. warp, wavefront). There is a trade-off between latency and throughput when choosing the tile group size. On certain workloads, latency of a task can be reduced by adding more tiles to the group. If latency is less of a concern, many smaller tile groups can work on individual tasks in parallel to improve

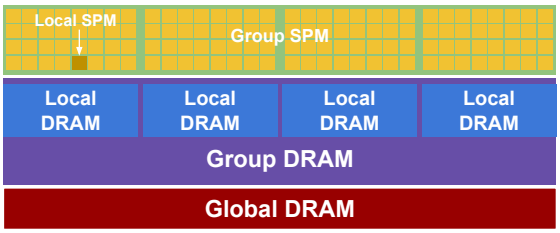

Fig. 5: **HB's PGAS mapping allows programmers to map data into the best place in the memory hierarchy to exploit physical locality.** PGAS mapping logically defines how tiles group together and share data in HB's physically flat cache and network hierarchy.

throughput. However, some workloads, such as graph and sparse applications, may not benefit linearly from more tiles in a tile group. In such case, smaller tile groups can be used to enable task-level parallelism. A *task*, in this context, refers to a coarse-grained unit of computation with data-dependent control flow that shares a common data structure and can be executed in parallel. For example, tasks can be different queries on a common graph or multiplying a stationary sparse weight matrix with different activation matrices. Tile groups can leverage the reconfigurable HW Barrier to synchronize in smaller groups. At the chip-level, every Cell could be running the same kernels, each working on a portion of a large problem, or running different kernels in a producer-consumer fashion (Figure 6).

The kernel code can be written in C/C++ and compiled with RISC-V GNU/LLVM toolchain [2] and with the support of domain-specific languages [13], [16], [17]. The host runtime code is responsible for memory management and data transfer. The host launches the kernel on the Cells by passing the pointers to the data. Tiles have distinct IDs like the block and thread IDs in CUDA [41].

Kernels execute in the PGAS, which is divided into five major address spaces to reflect the different levels of memory hierarchy in the HB system (Figure 5). This allows programmers to have full control of data management to exploit physical locality. Translation from a virtual address to a network address is done with a low-cost combinational logic without involving any expensive address translation hardware, such as a TLB. A few upper bits of an address determine which major address space it belongs in. Depending on address space: X,Y coords are either directly encoded in the address or produced by an address hashing scheme.

(1) *Local SPM* gives access to the local 4 KB scratchpad. Addresses (`0x0` $\sim$ `0xfff`) are private to each tile. Stack is often allocated here. Tiles can copy a block of data from DRAM to Local SPM for fast local processing.

(2) *Group SPM* can be used to remotely access other tiles' scratchpads. This is a shared address space for all tiles. The X,Y coordinates of the target tile and the SPM address offset is encoded in the address. This is particularly useful for spatially distributing data, when communication is structured and well-defined (e.g. nearest-neighbor and systolic-array patterns).

```
// HOST PROGRAM;
// Instantiate two Cells at (X,Y) location (0,0) and (1,0)
auto cell0 = new Cell(0,0);
auto cell1 = new Cell(1,0);
// Load different kernels on two Cells;
cell0.load_kernel("producer.riscv");
cell1.load_kernel("consumer.riscv");
// Allocate memory on Cells' Local DRAM;
float* input0, input1, output2;
cell0.malloc(4096*sizeof(float), &input0);
cell1.malloc(4096*sizeof(float), &input1);
cell1.malloc(4096*sizeof(float), &output2);
// Launch kernels;
// group_dram() produces a pointer to
// Cell1's Local DRAM in Group DRAM space;
cell0.launch(input0, cell0.group_dram(cell1,input1));
cell1.launch(input1, output0);
```

Fig. 6: **An example of a host program launching producer-consumer kernels on two Cells.** Using Group DRAM pointer, Cell0 can directly write the results in Cell1's Local DRAM space to minimize data movement.

(3) *Local DRAM* can be used to access DRAM memory space exclusively allocated for each Cell, where the majority of computation is done. This address space is private to a Cell, but shared by its constituent tiles. The access is done via the local cache banks within the Cell. *Regional IPOLY hashing* [47] pseudo-randomly distributes the address space among the cache banks at cache-line granularity. This prevents the partition camping problem with 2^n-stride access patterns prevalent in many parallel applications [6], [31].

(4) *Group DRAM* can be used to access another Cell's Local DRAM space. This is useful for broadcasting or gathering new results among Cells before the next program phase. As shown in Figure 3, Cellular Manycore's inter-Cell connections allow efficient transfers of both sparse and sequential data. This address space can be also used in a producer-consumer model, where one Cell produces and writes the results directly into another Cell's Local DRAM space. Figure 6 shows an example of a host code setting up this model.

(5) *Global DRAM* is shared by all tiles on the chip, and distributed on all cache banks on the chip using a custom hash function. This provides a convenient space for all Cells to combine the partial results at the end of kernel execution. The host can use this space to transfer a large block of data to the chip using the full DRAM bandwidth. In a 100K-core chip, all-to-all communication can become unsustainable. In such case, the chip can be divided into *grids*, a rectangular group of Cells to add a layer of locality. The most significant bits of the address are then used to select the grid, and the remaining bits are used for hashing within the grid.

B. Parallel Benchmark Suite

In order to demonstrate parallel programmability of HB, we introduce a parallel benchmark suite, inspired by Berkeley's parallel computing dwarfs [7]. *Dwarfs* represent a broad set of parallel computing and communication patterns, which has persisted through time and will remain relevant in the foreseeable future. Covering these bases lends confidence that HB can adapt to rapidly evolving workloads as a general-purpose parallel architecture. Our benchmark suite provides

better coverage of these dwarfs than the previous suites. For instance, GAP Benchmark Suite [11] specializes in graph processing, but lacks sparse and dense LA. Parboil [53] lacks spectral and N-body methods. Table I summarizes the benchmarks and the corresponding dwarfs. These benchmarks generally fall into one of three categories:

Benchmarks (Abbrev.)	*Dwarfs*	Input Data
AES (AES)	Combinational Logic	16384×1KB messages
Barnes-Hut (BH)	N-Body	16K, 32K, 64K bodies
Black-Scholes (BS)	MapReduce	10M options
Breadth-First Search (BFS)	Graph Traversal	See Table Ib
2-D FFT (FFT)	Spectral Method	16K/32K points (64/512×)
Jacobi (Jacobi)	Structured Grid	256/512×512×64
PageRank (PR)	Graphical Model	See Table Ib
Smith-Waterman (SW)	Dynamic Programm.	64K sequences
MatMult (SGEMM)	Dense LA	512×512×512 (256×)
Sparse MatMult (SpGEMM)	Sparse LA	See Table Ib

(a) List of benchmarks and their corresponding *Dwarfs* from [7].

Name (Abbrev.)	Type	Edges	Vertices
wiki-Vote (WV)	Social	103689	8297
offshore (OS)	Scientific	4242673	259789
roadNet-CA (CA)	Road	5533214	1971281
road-central (RC)	Road	33866826	14081816
road-usa (US)	Road	57708624	23947347
ljournal-2008 (LJ)	Social	79023142	5363260
hollywood-2009 (HW)	Social	113891327	1139905

(b) List of sparse matrix, graphs in Compressed Sparse Row format used in the evaluation (Source: [20])

TABLE I: **Ten parallel benchmarks used to demonstrate HB's parallel programmability.**

(1) *Compute-intensive, Low-communication*: AES, BS, and SW have high operational intensity and require very little memory access. In these kernels, it is crucial to take advantage of the local scratchpad for frequently accessed data. In AES, tiles keep their own copies of S-box in Local SPM. BS is characterized by the heavy use of the FP divider and square-root unit, which is included in each HB tile. SW is an example of dynamic programming, which tends to have a high branch-miss rate. These kernels are easy to accelerate by adding cores.

(2) *Compute-intensive, Sequential-access*: SGEMM, FFT, and Jacobi are characterized by different phases of the program, where all tiles initially load large, sequential blocks of data, compute for a long time, and then dump out the results. Load Packet Compression helps the initial loading, and the write-validate cache helps write out the result. Jacobi stencil method makes use of the Group SPM space most effectively. Each tile loads $1\times1\times512$ vector on its Local SPM, and synchronize by using the memory fence, followed by the fast HW barrier. Tiles can access neighboring pixels from the nearest tiles (even the ones in another Cell) with low latency using the Group SPM pointers (Figure 7). This can be difficult in hierarchical manycore, where the bandwidth between clusters is limited, and accessing another cluster may involve another level of network or memory hierarchy.

(3) *Memory-intensive, Irregular-access*: SpGEMM, PR, BFS, and BH operate on sparse and irregular data structures that are difficult to partition. Each Cell replicates the whole or part of the data structure in its Local DRAM for faster access. There are usually multiple iterations in the algorithm, and the Cells need to synchronize at the end of each iteration

```
// Each tile has 1x1x514 buffer on local SPM;
#define Z 512
float data[Z+2];
// Pointers to neighbor SPMs using group SPM pointer;
// __tile_x,y is this tile's coordinate;
float *p_left  = group_spm(__tile_x-1,__tile_y,&data[1]);
float *p_right = group_spm(__tile_x+1,__tile_y,&data[1]);
float *p_up    = group_spm(__tile_x,__tile_y+1,&data[1]);
float *p_down  = group_spm(__tile_x,__tile_y-1,&data[1]);
for (int i = 0; i < Z; i+=4) {
  // Load 22-points in register file;
  register float self[6];
  register float left[4];
  register float right[4];
  register float up[4];
  register float down[4];
  // Remote loads; unrolled by compiler...
  for (int j = 0; j < 6; j++) {
    self[j] = data[i-1+j];
  }
  // Remote loads; unrolled by compiler...
  for (int j = 0; j < 4; j++) {
    left[j] = p_left[i+j];
    right[j] = p_right[i+j];
    up[j] = p_up[i+j];
    down[j] = p_down[i+j];
  }
  // Compute and store 1x1x4 output...
}
```

Fig. 7: **Jacobi kernel snippet demonstrating the use of Group SPM pointers to remotely access nearby scratchpads.** if the communication pattern is well-defined, as in Jacobi (nearest-neighbor), data can be spatially distributed in Group SPM for fast access and persistent storage. Remote loads are non-blocking and can be pipelined in the network to create a load-use distance and hide latency.

to exchange the partial results necessary for the next iteration. Cells can either broadcast the data using the Group DRAM pointers, or combine the data in the Global DRAM space.

Workload imbalance is a major challenge for these types of kernels [37], [51], [52], [54], [62]. Although this is beyond the scope of this paper, we describe the basic mechanisms used in the evaluation that more advanced methods can build upon. To split the work among the Cells, the nodes to process are first statically split among the Cells. For example, in the direction-switching BFS [10], each Cell is assigned a subset of frontier nodes for the forward direction, or a subset of unvisited nodes for backward direction. In SpGEMM, implementing Gustavson's algorithm [24], each Cell is given a subset of output rows to compute. Then, the tiles can use the atomic-add to implement parallel for-loop (Figure 8).

V. Evaluation

In this section, we evaluate the HB architecture using the parallel kernels, described in Section IV.

A. Experimental Setup

HB's performance has been measured by the cycle-accurate simulation of its silicon-validated open-source RTL. We simulate four 16 GB stacks of HBM2 [26] operating at 1.0 GHz for 1 TB/s peak bandwidth. DRAMSim3 [36] has been integrated with the RTL model using the SystemVerilog DPI interface to accurately model HBM2 pseudo-channel timing. From the 2048-core ASIC, we measure that it can achieve 1.35 GHz at

```
#define NUM_FRONTIER 1000000
// These point to buffers in Local DRAM;
int* q0; // Initialized to 0 by host;
int* offset, nonzeros, distance;
int* curr_frontier, next_dense_frontier;
// Parallel for-loop using amoadd;
for (int i=amoadd(1,q0); i<NUM_FRONTIER; i=amoadd(1,q0)) {
  int src = curr_frontier[i];
  int start = offset[src];
  int end = offset[src+1];
  for (int j = start; j < end; j++) {
    int nz = nonzeros[j];
    if (distance[nz] == -1) {
      // node hasn't been visited;
      int word_idx = nz / 32;
      int bit_idx = nz % 32;
      amoor(1<<bit_idx, &next_dense_frontier[word_idx]);
    }
  }
}
```

Fig. 8: **BFS kernel snippet demonstrating the use of atomic add to implement a simple parallel for-loop.** The runtime of each subtask in graph applications can vary greatly depending on the graph input. SPMD model's independent thread execution is its major advantage, which is well-suited for kernels with severe data access and control flow irregularity.

the nominal voltage with passive air cooling. The parameters are listed in Table II. Due to excessively long simulation time, a multi-Cell simulation has been modeled by using multiple single-Cell simulations running in parallel and conservatively estimated data transfer time between program phases based on data transfer size and network bandwidth.

Configurations	**16×8**	**16×16**	**32×8**	**2×16×8**
Area (mm^2)	311	539	620	620
Cell Array	8×8	8×8	8×8	16×8
Core Array	16×8	16×16	32×8	16×8
Scratchpad Size (KB)	4			
Cache Sets	64			
Cache Ways	8			
Cache Block Size (B)	64			
Cell Cache Banks	32	32	64	32
Cell Cache Size (MB)	1	1	2	1
Total On-chip Storage (MB)	96	128	192	192
Core Frequency	1.35 GHz			
Memory Frequency	1.0 GHz			
Core / mm^2	26.4	30.3	26.4	26.4

TABLE II: **HB Machine Configurations.** We explore different strategies to double the hardware resources, while the HBM2 bandwidth remains constant (Figure 9).

Our baseline HB design has 8×8 Cell array, and 16×8 core array, totaling 8192 cores. In this setup, each Cell is mapped to one HBM2 pseudo-channel. Based on our 14/16 nm implementation, Baseline HB area is measured at 310 mm^2, which is less than half of the contemporary GPGPU die area [40]. We explore different strategies to double the compute resources, while keeping the HBM2 bandwidth constant (Figure 9):

(1) *Doubling the size of each Cell vertically* (16×16): this simply doubles the number of compute tiles, but reduces the cache capacity per tile by half. Larger Cell generally increases the average hop latency to the cache banks.

(2) *Doubling the size of each Cell horizontally* (32×8): this doubles the number of compute tiles, the cache bank capacity, and also the cache bandwidth. The increased cache capacity allows working on larger datasets more efficiently by reducing the cache miss rate. However, a wider Cell dimension puts more pressure on the bisection bandwidth.

(3) *Doubling the number of Cells* (2×16×8): this achieves the similar effect as 32×8, but the key difference is that it creates two distinct Local Cell address spaces, and it avoids the pressure on the bisection bandwidth. This is not a problem for embarrassingly-parallel kernels, where data can be easily split; however, data structures like graphs or octrees, which are difficult to partition, may need to be duplicated in the Local DRAM space of each Cell.

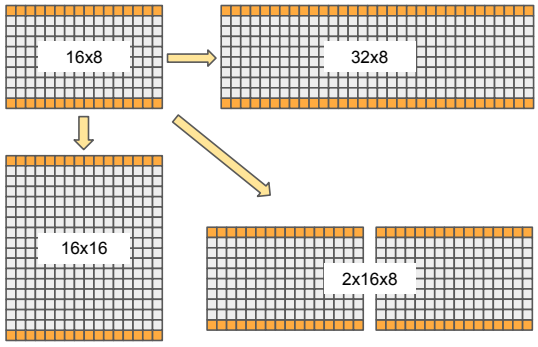

Fig. 9: **Exploring different strategies to double the hardware resources, while the HBM2 bandwidth remains constant.** The comparison is shown in Figure 15.

B. Incremental Feature Analysis

In Figure 10, we evaluate 16×8 single Cell performance as we incrementally apply hardware features to our baseline architecture (Cellular Baseline). We also compare against the "Baseline Manycore", which has the cache capacity, core density, and NoC router bandwidth normalized to [12]. We incrementally improve each parameters (router, cache, density opt) to match the "Cellular Baseline". We then add the architectural features in the following order: Non-blocking loads, Ruche Network, Write-validate policy, Load Packet Compression, Regional IPOLY, and finally Non-blocking cache. Figure 10 shows these optimizations and features improve the performance for a variety of kernels without greatly hurting any particular kernel. Applying all optimizations yields 5.2× geomean speedup vs. Baseline Manycore.

Looking at the progression of the geomean speedup, it is worth noting that the higher core density is most beneficial for all kernels. For the memory-intensive kernels, there has to be an enough number of cores to generate memory requests and keep the HBM2 channels saturated. The compute-intensive kernels are easy to scale with more cores, arguing for maximizing compute density with area-optimized RISC-V cores.

BH has benefited from Regional IPOLY the most, because each tile is given 4 KB private stack space allocated on Local DRAM to keep track of the nodes to visit during the tree-traversal; without the hashing, every tile would initially hit the same cache bank, which would cause a massive traffic jam. Jacobi has improved by 17–48×, which highlights the strength

of using Group SPM for the well-defined traffic patterns (e.g. nearest-neighbor, systolic-array) in HB.

C. Core and HBM2 Utilization

Figure 11 analyzes the core and HBM2 utilization of the most optimized HB Cell from Figure 10. The core utilization graph shows the percentage of cycles that cores are either executing (Int or FP instruction) or stalling. Int instructions also include memory-access and control instructions. The stall percentage is broken down into the stall types in Table III.

Stall Type	Description
MemorySysStall	Stalled on remote load responses from DRAM
NetworkStall	Outgoing request packet is stalled due to network congestion
BypassStall	Pipeline interlock due to back-to-back dependent instructions
BranchMiss	Bubble cycles incurred by branch miss
DivStall	Stalled on the iterative FP divide and square-root unit
FenceStall	Stalled on all outstanding remote memory operations to complete
BarrierStall	Stalled on the HW barrier to complete

TABLE III: **List of core stall types used in Figure 11**.

HBM2 utilization graph shows the % of cycles that HBM2 channel is either reading, writing, busy (i.e. one or more queue requests exist, but HBM2 cannot accept any commands due to DRAM timing parameters), or idle (i.e. queue is empty). Refresh cycles have been subtracted from the denominator.

In Figure 11, kernels are ordered from memory-intensive to compute-intensive. The broad spectrum suggests that the parallel kernels exercises different bottlenecks in the architecture, which motivates building a balanced parallel architecture.

PR, BFS, and SpGEMM are memory-bound, mostly waiting for the remote load responses from HBM2. If the HBM2 is fully utilized, it is usually a good sign; performance cannot improve further without more HBM2 bandwidth. Those with low HBM2 utilization may be improved by having each core generate more memory requests by unrolling the loop further. Graphs like wiki-Vote (WV) that are small and have high variance on degrees do not perform very well. BFS for the road networks have relatively lower HBM utilization because the frontier size remains relatively small throughout the entire search. One remedy for this is to leverage the task-level parallelism by dividing the Cell into two or more Tile Groups, as mentioned above. Each tile group uses the same graph data structure, but independently runs different graph queries to generate more memory requests. High barrier stall is usually indicative of high tail-latency problems, which may be improved by better load balancing.

AES, SW, SGEMM, and BS are compute-bound, which could easily benefit from having more tiles. SW is affected by high branch miss rate, which can be fixed with RISC-V integer min-max extension [1] or the fused add min-max functions recently added in GPUs [22]. BH and BS could benefit from the faster iterative FP divide and square-root unit, especially for the back-to-back inverse square-root operation. SGEMM has high core utilization, but does not fully utilize the HBM2 bandwidth, suggesting more cores, more speedup. BS has high bypass stalls because of the FP polynomial calculation.

Some kernels (FFT, Jacobi, SGEMM) are stalled due to the network congestion. Ruche Networks and Load Packet Compression help by increasing the bisection bandwidth and reducing the network load (Figure 14). Regional IPOLY also helps by distributing the network traffic more randomly.

D. Scaling Irregular Workloads using Tile Group

Figure 12 highlights the use of tile groups to scale highly irregular workloads, which result in poor resource utilization. Using eight tile groups of 4×4 (vs a single tile group of 16×8) improves the throughput of SpGEMM (WV) by 4× and the HBM2 utilization by 7.8×. Dividing further into smaller tile groups has a diminishing return, since it increases working set size, which then increases network traffic and cache miss rate.

E. NoC Bisection Utilization

Figure 14 shows the utilization of horizontal channels that cross the bisection of the 16×8 Cell. Because IPOLY hashing evenly distributes network traffics among all cache banks, the horizontal bisection bandwidth becomes the bottleneck. Even for a modestly sized 16×8 Cell, the bisection links on 2-D mesh could be stalled as much as 50% of the time. HB employs Ruche Networks and Load Packet Compression to mitigate this problem. Ruche Networks boost the bisection bandwidth by creating extra channels with unused wires. Load Packet Compression reduces the network load by compressing sequential accesses.

Figure 14 compares the bisection utilization of 16×8 Cell with (1) 2-D mesh, (2) Ruche Network, and (3) Ruche Network + Packet Compression. For 2-D mesh, PR (HW), Jacobi (DRAM), FFT (64K) have particularly high stall percentages. Ruche Networks significantly reduces the amount of time packets are stalled at the bisection across all kernels, except Jacobi ($), in which the nearest-neighbor communication is majority. Load Packet Compression helps with most kernels except SpGEMM, which has fewer sequential accesses.

F. Doubling HW Resources

Figure 15 evaluates three different ways to double the number of cores, while the HBM2 bandwidth is held constant (as described in Figure 9). 16×16, 32×8, and 2×16×8 achieve geomean speedup of 1.25×, 1.39×, 1.34× over Baseline HB, respectively. Compute-intensive kernels are generally easy to accelerate with more cores. Doubling the core without doubling the cache capacity and bandwidth is not as effective. The benefit of having larger Cells versus having more Cells is evident in BH. In 2×16×8, the octree structure, which has good temporal and spatial locality, is duplicated in the Local DRAM space of both Cells, which wastes the HBM2 bandwidth and cache capacity. However, for data structures with less locality, such as graphs in BFS or sparse matrices in SpGEMM, duplicating the data does not impact performance.

G. Comparison with Hierarchical Manycore

Figure 16 compares 32×8 HB Cells against a manycore model (ET), whose thread density, cache capacity, and network bandwidth are based on [21]. Both designs have equal HBM2 bandwidth. The comparison is done on irregular workloads,

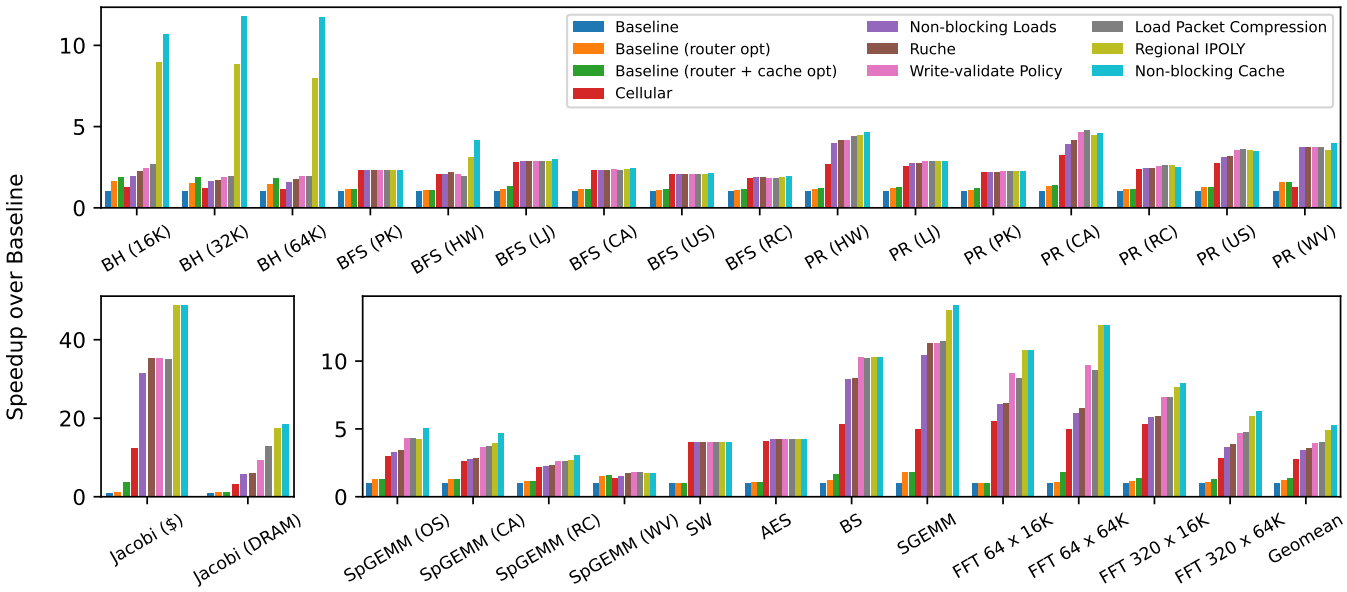

Fig. 10: **HB physical and architectural optimizations improve single-Cell performance by 5.2× over Baseline Manycore.** These optimizations improve a wide range of kernels without hurting the performance of any particular ones. Improving the core density stands out as the key contributor to performance uplift, which is why HB has taken the route to optimize the tile area as much as possible. Regional IPOLY improves programmability by eliminating the partition camping problem of 2^n-stride access.

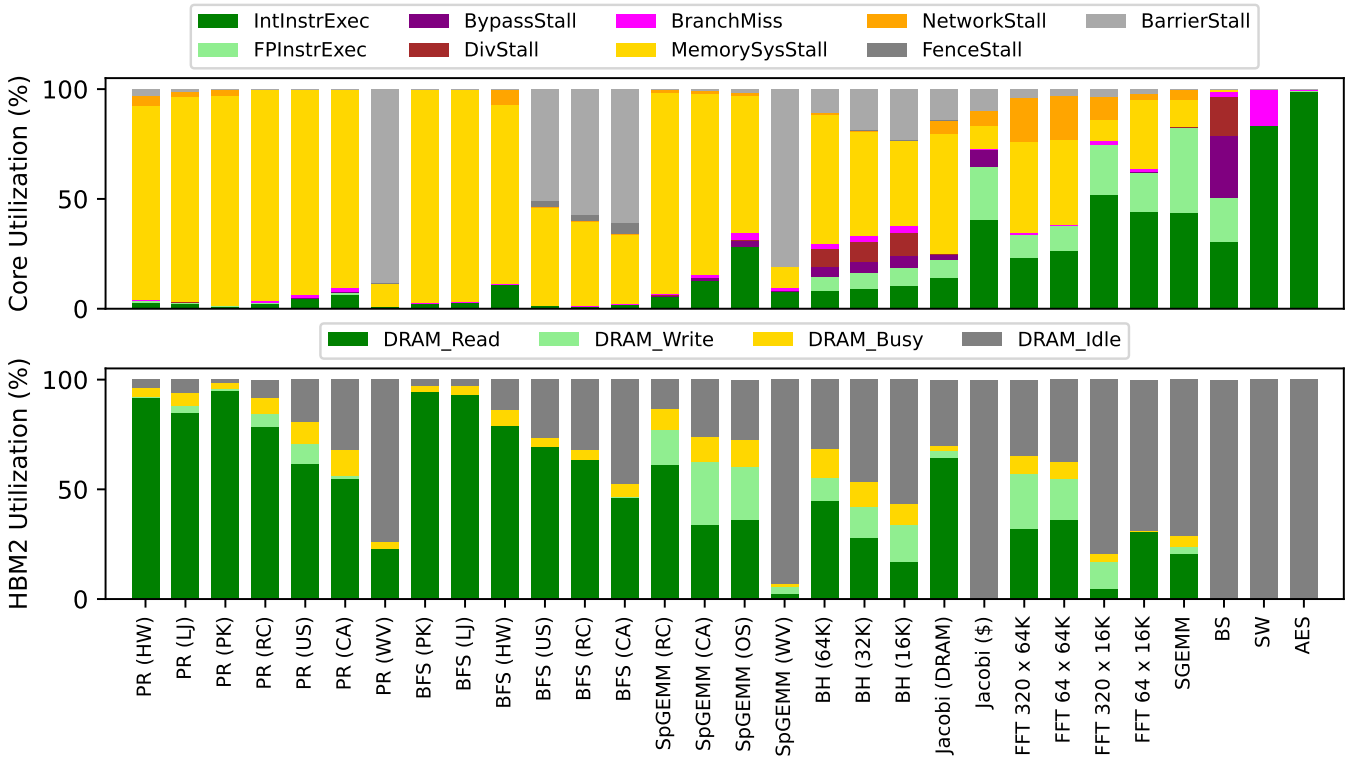

Fig. 11: **Core and HBM2 Utilization graphs give insight into where the bottlenecks are and how each kernel can be optimized further.** Kernels are ordered from memory-intensive to compute-intensive (from left to right). A broad spectrum of kernel characteristics tests bottlenecks in different parts of the architecture, which motivates creating a balanced architecture.

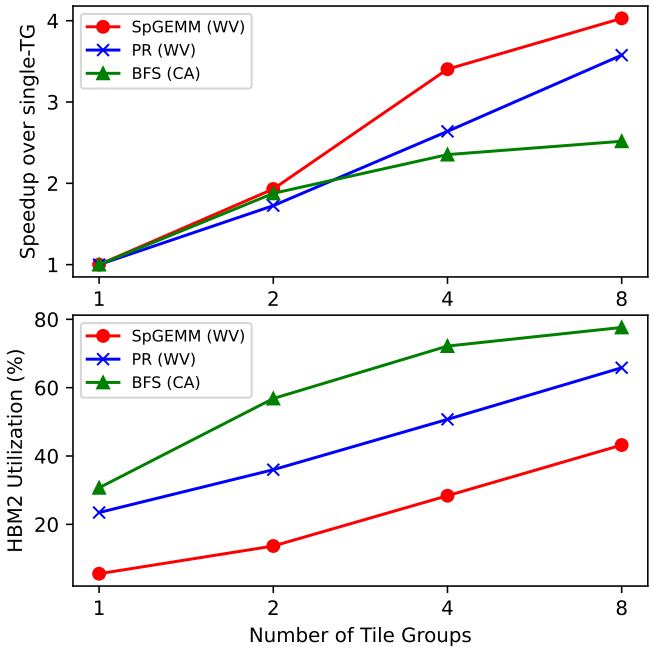

Fig. 12: **A 16×8 Cell can be regrouped into smaller tile groups to improve the throughput and DRAM utilization of irregular applications.** HB uses tile groups to manage threads into smaller groups working on different tasks while sharing the common data structure. Irregular workloads with low resource utilization (Figure 11) can be scaled by exploiting task-level parallelism that exists in large-scale applications.

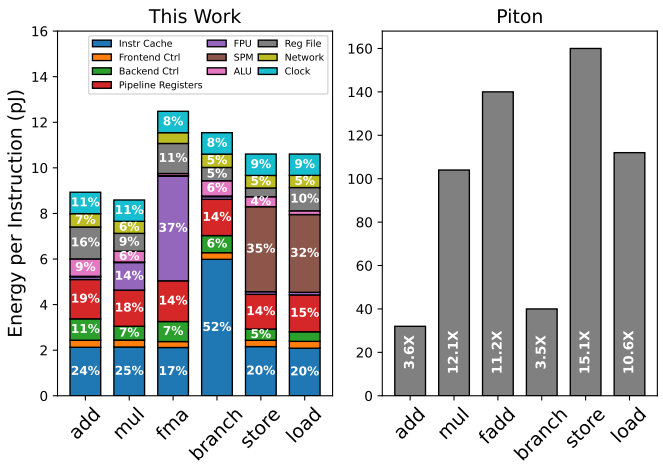

Fig. 13: **Comparison of "Energy per Instruction" (EPI) with [38].** For HB, EPI has been broken down into different parts. Energy figures from [38] have been normalized using CV^2 scaling. HB's EPI is 3.6–15.1× more efficient.

which require data transfer between clusters to highlight the impact of on-chip network on the overall performance.

Figure 16 breaks down the total run time into the execution time and the data transfer time that occur between the program phases. There are a few instances where ET's higher L2 capacity helps with the execution time, but the higher independent thread density of HB has the clearer advantage for the irregular workloads overall. It also shows that transferring a large volume of sparse data over wide 2-D mesh channels is inefficient and affects the overall program run time.

H. Energy Analysis

Figure 13 compares "Energy per Instruction" (EPI) with [38], whose figures have been normalized to the same process node using CV^2 scaling. For HB, EPI has been measured by gathering switching activities with random operands on a post-layout gate-level netlist and then running power analysis on Synopsys PrimeTime with extracted parasitics.

The result shows that HB's EPI is 3.6–15.1× more efficient. The observed efficiency can be attributed to several factors. Smaller icache (4 KB) reduces the instruction fetch energy. Using scratchpad instead of L1/L1.5 data caches reduces the energy overhead on memory operations. Instruction latencies are generally much lower for HB (3 cycles for fma, 2 cycles for mul and load/store). Furthermore, process-independent wire cap (0.2 pF/mm) perfers smaller HB tiles over Piton tiles (16.6×), since the wires for clock tree and control/data signals travel much less distance within the tile boundary.

VI. Related Work

Related manycore architectures are shown in Table IV. We summarize these and highlight the key differences.

A. Flat Manycore

Raw [58] is a 16-core, general-purpose, 32-bit manycore architecture with RISC ISA. Although Raw supports a global address space, it does not support load and store instructions that could directly access other core's memories; instead, explicit dynamic messages have to be sent in software, and the receiving core services this request either by triggering an interrupt or by explicitly receiving the memory request. HB attains much higher compute density by removing the expensive scalar operand network [8], and provides simplified mechanisms for accessing remote memory.

TILE64 [12] is a 3-wide VLIW 64-core Linux-capable many-core. A DMA engine in each tile can facilitate block-oriented data copy between caches and memory interfaces in the background. Message-based communication between tiles is prone to head-of-line blocking. In order to support out-of-order processing of messages (e.g. different from the order that messages arrived), messages can be tagged by the sender to be sorted into one of many receiving queues in each tile.

Epiphany-V [42] is a 1024-core 64-bit dual-issue RISC processor. It has three 136-bit wide 2-D mesh networks for read requests, on-chip write, and off-chip write traffic. The off-chip traffic is separated from the on-chip traffic to make the on-chip latency more deterministic. Epiphany-V does not have any L1 or L2 caches. Instead, each processor has a multi-ported scratchpad that can service instruction fetch, local load/store, and remote load/store simultaneously. A dual-issue core is good at keeping multiple hardware units busy (e.g. scratchpad and FPU every cycle), but the heavily-ported register file and complex bypass paths can become area-consuming.

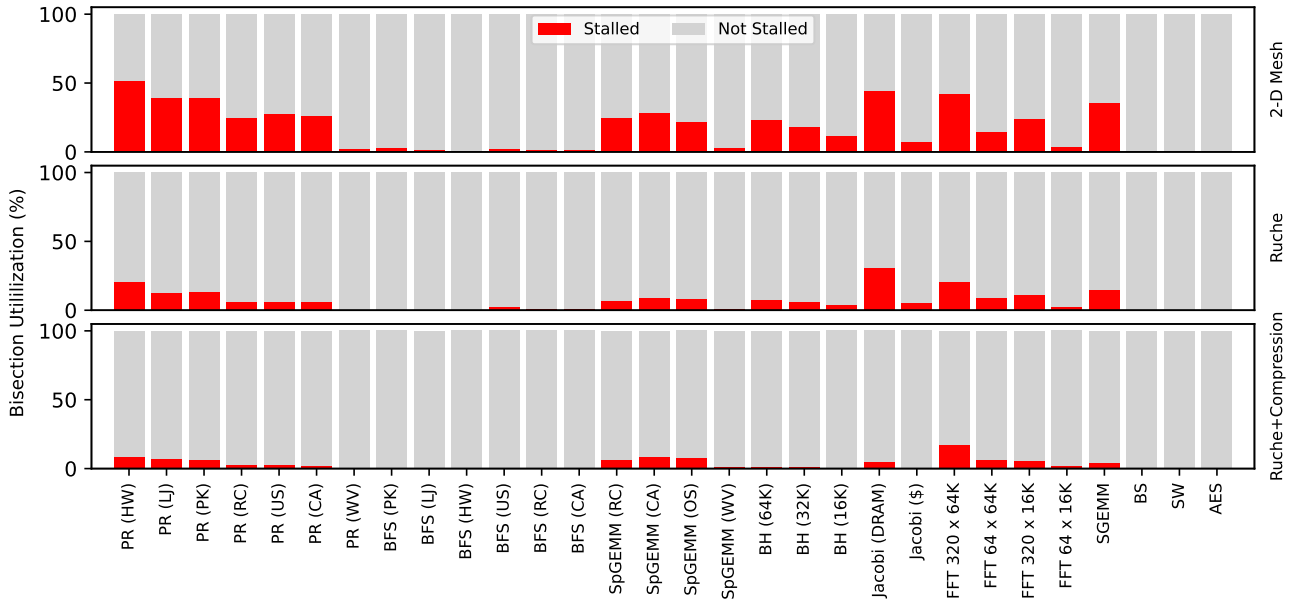

Fig. 14: **Even for a modestly sized 16×8 Cell, the horizontal bisection links on the 2-D mesh can be stalled up to 50% of the time.** Ruche Networks boost the bisection bandwidth by creating more channels. Load Packet Compression is particularly useful for the kernels with sequential accesses. These allow building larger Cells, which have the benefit of larger cache capacity and larger thread pool.

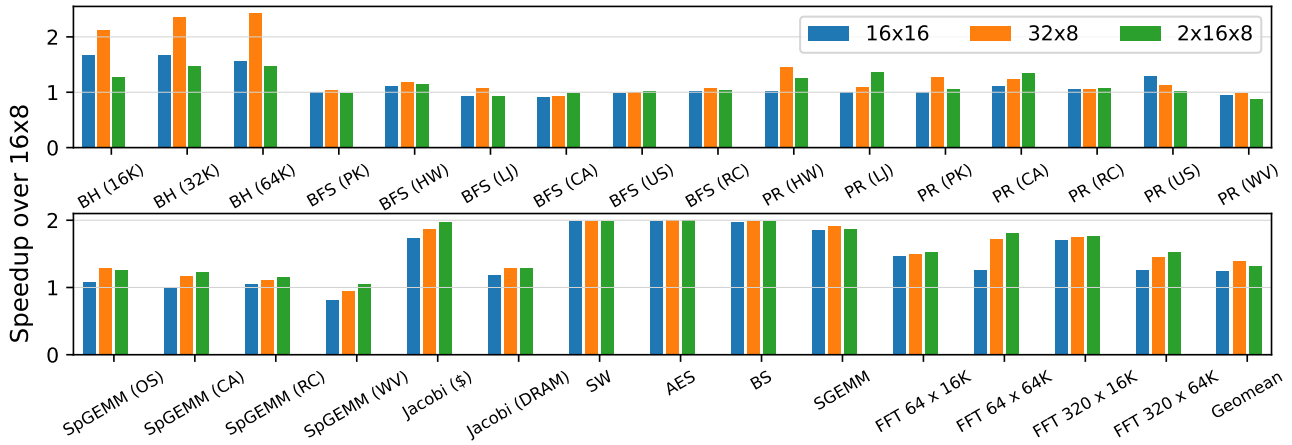

Fig. 15: **Three different strategies of doubling hardware resources (16×16, 32×8, 2×16×8) achieve geomean speedup of 1.25×, 1.39×, 1.34× respectively over Baseline HB.** Compute-intensive kernels are relatively easy to accelerate with more cores. Doubling the core without doubling the cache capacity is not as effective. The benefit of having larger Cells horizontally (32×8) is more evident than having more Cells (2×16×8).

Celerity [19] is a 496-core RV32I manycore processor. As in HB, all scratchpads in Celerity are globally addressable over the 2-D mesh network with a PGAS system. Celerity lacks remote load, cache banks, instruction cache, and FPU, which makes it much less programmable.

OpenPiton [9] is a cache-coherent manycore architecture with three 64-bit wide 2-D mesh networks to implement distributed, directory-based MESI protocol. Like TILE64, OpenPiton focuses on Linux capability, so the compute density is low. Each tile contains a private L1 and L1.5 cache, and a shared L2 cache with inclusive policy at every level, which proliferates duplicate data across the hierarchy. By default, cache lines are distributed among all L2 caches in the system. Although Coherence Domain Restriction [23] can be used to map a page to a specific subset of tiles to enable nearest-neighbor or consumer-producer communication, this adds complexity by requiring additional storage and hardware to maintain a software-defined mapping from virtual page to physical L2.

B. Hierarchical Manycore

ET-SoC-1 [21] has 1088 RV64IMAF *minion* cores, with configurable L1 data cache/scratchpad and FP vector units. Eight minion cores are grouped into *neighborhoods*, which

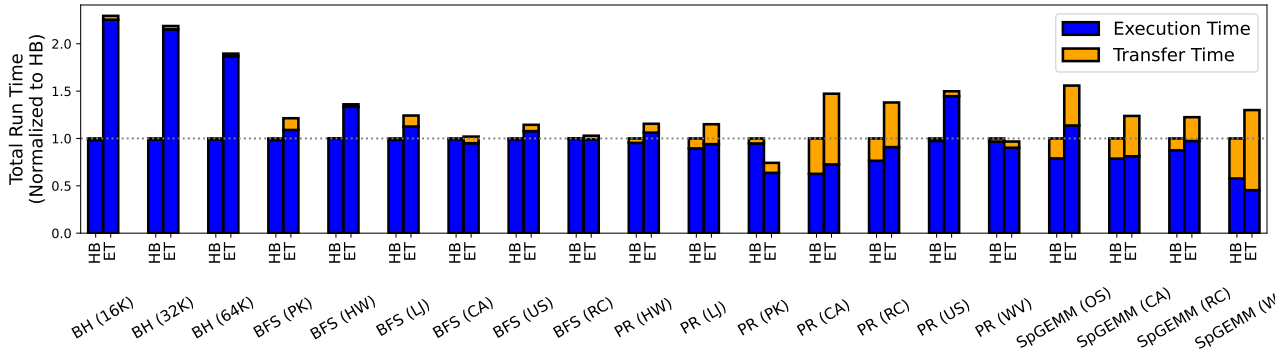

Fig. 16: **Performance comparison of irregular workloads between HB (32×8) and the manycore model (ET) based on [21].** There are a few instances where ET benefits from having larger L2 capacity. However, in most cases, higher independent thread density gives more advantage to HB. For ET, data transfer time is exacerbated by transferring a large amount of sparse data over a wide concentrated mesh channel with poor utilization.

Related Work	Category	Networks	Processor	Cores	FPUs	Scaled Area	Cores/mm2	Our ×	FPUs/mm2	Our ×
HammerBlade	Cellular	2 × 2-D Ruche	Single-issue	2048	2048	77.5 mm^2	26.4	1.0 ×	26.4	1.0 ×
TILE64 [12]	Flat	5 × 2-D Mesh	VLIW	64	0	19.4 mm^2	3.3	8.0 ×	0.0	–
RAW [58]	Flat	4 × 2-D Mesh	Single-issue	16	16	2.6 mm^2	6.2	4.3 ×	6.2	4.3 ×
Celerity [19], [48]	Flat	2 × 2-D Mesh	Single-issue	496	0	15.3 mm^2	32.4	0.8 ×	0.0	–
Epiphany V [42]	Flat	3 × 2-D Mesh	Dual-issue	1024	2048	117 mm^2	8.8	3.0 ×	17.5	1.5 ×
OpenPiton [9]	Flat	3 × 2-D Mesh	Single-issue	25	25	11.1 mm^2	2.3	11.7 ×	2.3	11.7 ×
ET-SoC-1 [21]	Hierarchical	Crossbar, 2 × 2-D CMesh	Vector	1088	8704	1710 mm^2	0.6	41.4 ×	5.1	5.2 ×
MemPool [15]	Hierarchical	Crossbar, Radix-4 Butterfly	Single-issue	256	0	8.6 mm^2	29.9	0.9 ×	0.0	–

TABLE IV: **Comparison of manycore designs on network topology, processor type, and compute density.** An area of each manycore chip has been scaled to 14/16 nm node for comparison. HB's area-optimized RISC-V cores and simplified NoCs significantly improve the network scalability and compute density. HB provides as much as 41.4× greater core density and 5.2× greater 32-bit FPU density over the previous manycore designs.

share a single large instruction cache. Four neighborhoods are grouped by a crossbar into *shires*. The shires communicate over a mesh network with 1024-bit links. Outside the shire, communication is block-structured, which reduces the ability to make fine-grained random accesses. All cores in HB have global word-level access to every scratchpad and cache banks on the chip on a globally uniform network.

MemPool [15] is a proposed architecture with 256 single-cycle *Snitch* [64] cores implementing RV32IMA. Eight cores are aggregated into a *Tile* that shares a 2 KB L1 instruction cache, and a 16 KB data scratchpad via two fully-connected crossbars. 16 Tiles are aggregated into *Groups* via two 16×16 crossbars, and four Groups are interconnected by a radix-4 butterfly network. In HB, the network routers are co-placed with the processor logic in a tile using NoC Symbiosis [46], which allows seamless integration of thousands of tiles. In MemPool, however, the routers are placed between Tiles, which results in inefficient area utilization and routing congestion.

VII. Conclusion

HammerBlade Manycore proposes a scalable approach to integrate a massive array of *scalar processors*. These scalar processors (i.e. MIMD) are generally easier to program for both regular and irregular data-level parallelism (DLP) than vector or SIMT processors. HB makes up for its inefficiency with regular DLP (e.g. a single instruction acting on multiple functional units or operands) with its extreme compute density.

Looking at Table IV, we see that HammerBlade attains significant thread and FP compute density, exceeded only by less scalable or less programmable alternatives such as Celerity (our prior work).

HB programming interface exposes data placement and thread management for better physical locality and resource utilization. Simplified network and cache hierarchy provides unprecedented on-chip bandwidth and highly parallel irregular memory accesses. Our detailed analysis provides insights into how this system can be scaled and how the bottlenecks can be fixed for further improvement.

A 2048-core version of HB Manycore ASIC has been validated in silicon with the 14/16 nm FinFET process, running at 1.35 GHz at the nominal voltage. The synthesizable RTL source code is free and open-sourced with extensive performance profiling tools. It is capable of *2.8 Tera RISC-V instructions/s* at its peak, and can be programmed for a wide spectrum of parallel workloads using a familiar C++ / SPMD interface.

Acknowledgments

We thank the many contributors to HammerBlade. HammerBlade was funded by DARPA SDH Award #FA8650-18-2-7863. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and not of AFRL and DARPA or the U.S. Government.

REFERENCES

[1] "RISC-V Bit-manipulation," 2021. [Online]. Available: https://github.com/riscv/riscv-bitmanip

[2] "RISC-V GNU Compiler Toolchain," 2023. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain

[3] "Solderpad Hardware License," 2023. [Online]. Available: https://solderpad.org/licenses/

[4] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving predictable performance through better memory controller placement in many-core cmps," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2009, p. 451–461.

[5] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Rao, A. Rovinski, N. Sun, C. Torng, L. Vega, B. Veluri, S. Xie, C. Zhao, R. Zhao, C. Batten, R. Dreslinski, R. Gupta, M. Taylor, and Z. Zhang, "Experiences using the RISC-V ecosystem to design an accelerator-centric SoC in TSMC 16nm," in *1st Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*, 2017.

[6] A. M. Aji, M. Daga, and W. C. Feng, "Bounding the Effect of Partition Camping in GPU Kernels," ser. CF '11. New York, NY, USA: Association for Computing Machinery, 2011.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," 2006.

[8] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar operand networks: on-chip interconnect for ILP in partitioned architectures," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 341–353.

[9] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "OpenPiton: An Open Source Manycore Research Framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2016, p. 217–232.

[10] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing Breadth-First Search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[11] S. Beamer, K. Asanović, and D. A. Patterson, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[12] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008, pp. 88–598.

[13] A. Brahmakshatriya, E. Furst, V. A. Ying, C. Hsu, C. Hong, M. Ruttenberg, Y. Zhang, D. C. Jung, D. Richmond, M. B. Taylor, J. Shun, M. Oskin, D. Sanchez, and S. Amarasinghe, "Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 429–442.

[14] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151.

[15] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 701–706.

[16] L. Cheng, P. Pan, Z. Zhao, K. Ranjan, J. Weber, B. Veluri, S. B. Ehsani, M. Ruttenberg, D. C. Jung, P. Ivanov, D. Richmond, M. B. Taylor, Z. Zhang, and C. Batten, "A Tensor Processing Framework for CPU-Manycore Heterogeneous Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1620–1635, 2022.

[17] L. Cheng, M. Ruttenberg, D. C. Jung, D. Richmond, M. Taylor, M. Oskin, and C. Batten, "Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures with Software-Managed Scratchpad Memories," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 46–58.

[18] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights," *Proceedings of the IEEE*, vol. 109, no. 10, pp. 1706–1752, 2021.

[19] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, 2018.

[20] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011.

[21] D. R. Ditzel and the Esperanto team, "Accelerating ML Recommendation With Over 1,000 RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip," *IEEE Micro*, vol. 42, no. 3, pp. 31–38, 2022.

[22] A. C. Elster and T. A. Haugdahl, "Nvidia Hopper GPU and Grace CPU Highlights," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95–100, 2022.

[23] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence Domain Restriction on Large Scale Systems," ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 686–698.

[24] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[25] S. Hooker, "The Hardware Lottery," *Commun. ACM*, vol. 64, no. 12, p. 58–65, nov 2021.

[26] JEDEC, Jan 2020. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

[27] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore ipu architecture via microbenchmarking," *arXiv preprint arXiv:1912.03413*, 2019.

[28] N. P. Jouppi, "Cache Write Policies and Performance," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, p. 191–201, may 1993.

[29] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[30] D. C. Jung, S. Davidson, C. Zhao, D. Richmond, and M. B. Taylor, "Ruche Networks: Wire-Maximal, No-Fuss NoCs," in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2020, pp. 1–8.

[31] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[32] Y. LeCun, "Deep Learning Hardware: Past, Present, and Future," in *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2019, pp. 12–19.

[33] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: enabling power efficient GPUs through register compression," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 502–514.

[34] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. New York, NY, USA: Association for Computing Machinery, 1998, p. 46–57.

[35] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 129–140.

[36] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[37] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[38] M. McKeown, A. Lavrov, M. Shahrad, P. J. Jackson, Y. Fu, J. Balkind, T. M. Nguyen, K. Lim, Y. Zhou, and D. Wentzlaff, "Power and Energy Characterization of an Open Source 25-Core Manycore Processor," in *HPCA*, 2018, pp. 762–775.

[39] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 27–39.

[40] NVIDIA, "NVIDIA Tesla V100 GPU Architecture," 2017.

[41] NVIDIA, "CUDA C++ Progamming Guide," 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[42] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC System-on-Chip," *arXiv preprint arXiv:1610.01832*, 2016.

[43] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, 2014, pp. 1719–1726.

[44] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 130–139.

[45] Y. Ou, S. Agwa, and C. Batten, "Implementing Low-Diameter On-Chip Networks for Manycore Processors Using a Tiled Physical Design Methodology," in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2020, pp. 1–8.

[46] D. Petrisko, C. Zhao, S. Davidson, P. Gao, D. Richmond, and M. B. Taylor, "NoC Symbiosis," in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2020, pp. 1–8.

[47] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 74–83.

[48] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski, "A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS," in *2019 Symposium on VLSI Circuits*, 2019, pp. C30–C31.

[49] D. Schor, "A Look At The ET-SoC-1, Esperanto's Massively Multi-Core RISC-V Approach To AI," 2021. [Online]. Available: https://fuse.wikichip.org/news/4911/a-look-at-the-et-soc-1-esperantos-massively-multi-core-risc-v-approach-to-ai/

[50] D. Seo, A. Ali, W.-T. Lim, and N. Rafique, "Near-optimal worst-case throughput routing for two-dimensional mesh networks," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 432–443.

[51] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph Processing on GPUs: A Survey," *ACM Comput. Surv.*, vol. 50, no. 6, jan 2018.

[52] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity," *Journal of Parallel and Distributed Computing*, vol. 27, no. 2, pp. 118–141, 1995.

[53] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, p. 27, 2012.

[54] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing Load Imbalance of Graph Partitioning," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: Association for Computing Machinery, 2017.

[55] E. Talpes, D. D. Sarma, D. Williams, S. Arora, T. Kunjan, B. Floering, A. Jalote, C. Hsiong, C. Poorna, V. Samant, J. Sicilia, A. K. Nivarti, R. Ramachandran, T. Fischer, B. Herzberg, B. McGee, G. Venkataramanan, and P. Banon, "The Microarchitecture of DOJO, Tesla's Exa-Scale Computer," *IEEE Micro*, vol. 43, no. 3, pp. 31–39, 2023.

[56] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[57] M. B. Taylor, "Basejump STL: SystemVerilog Needs a Standard Template Library for Hardware Design," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018.

[58] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04, USA, 2004, p. 2.

[59] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0," Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[60] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[61] WikiChip, "Lithography Scaling Database," 2022. [Online]. Available: https://en.wikichip.org/wiki/14_nm_lithography_process

[62] S. Yan, C. Li, Y. Zhang, and H. Zhou, "YaSpMV: Yet Another SpMV Framework on GPUs," *SIGPLAN Not.*, vol. 49, no. 8, p. 107–118, feb 2014.

[63] J.-S. Yang and C.-T. King, "Designing tree-based barrier synchronization on 2D mesh networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 6, pp. 526–534, 1998.

[64] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1845–1860, 2021.