SAND2018-6370R

EXASCALE COMPUTING PROJECT

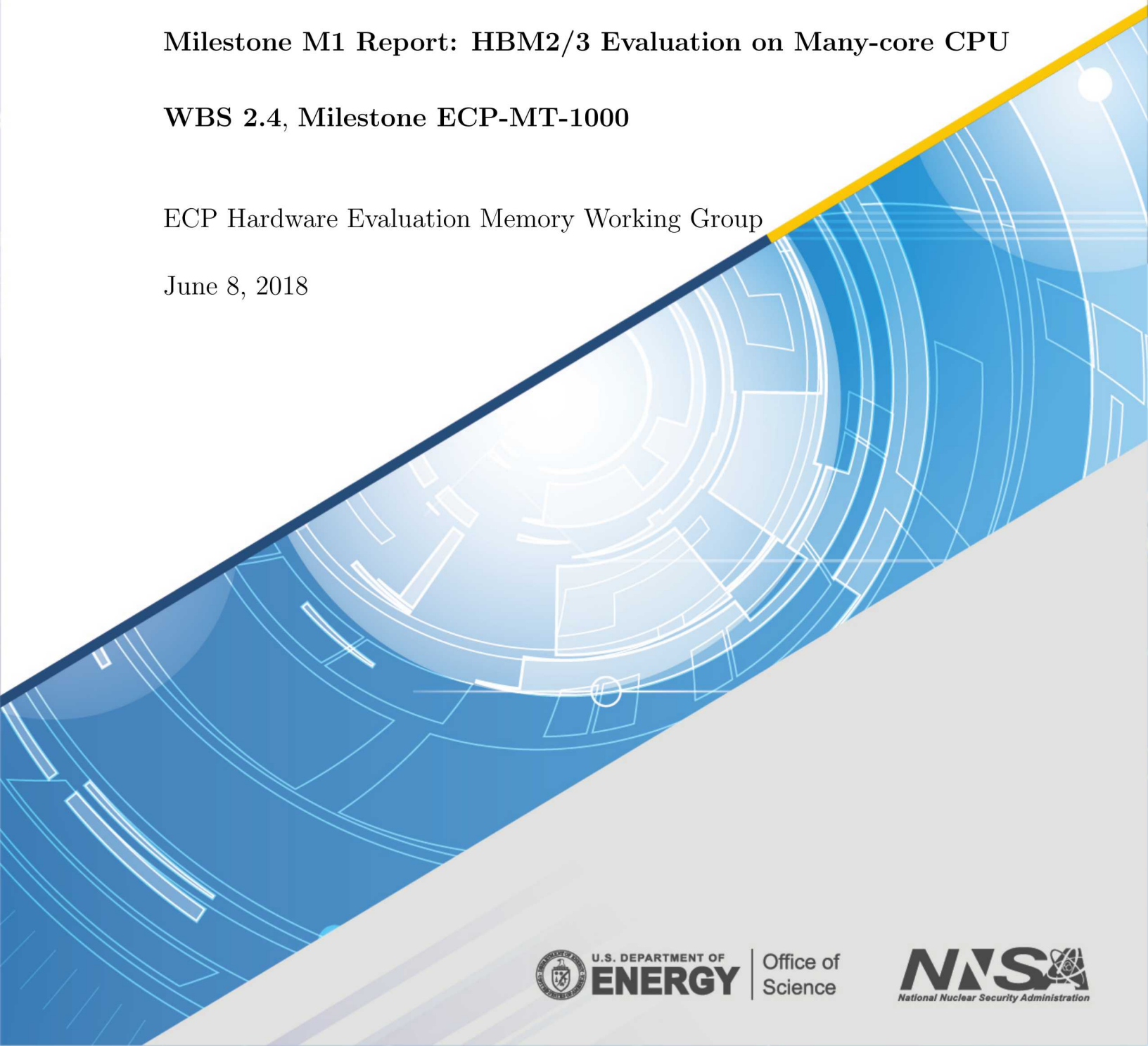
ECP-U-2018-XXX

Milestone M1 Report: HBM2/3 Evaluation on Many-core CPU

WBS 2.4, Milestone ECP-MT-1000

ECP Hardware Evaluation Memory Working Group

June 8, 2018



U.S. DEPARTMENT OF ENERGY | Office of Science

National Nuclear Security Administration

ECP-U-2018-XXX

ECP Milestone Report
Milestone M1 Report: HBM2/3 Evaluation on Many-core CPU
WBS 2.4, Milestone ECP-MT-1000

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

June 8, 2018

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

ECP Milestone Report
Milestone M1 Report: HBM2/3 Evaluation on Many-core CPU
WBS 2.4, Milestone ECP-MT-1000

APPROVALS

Submitted by:

Gwen Voskuilen, Alfredo Gimenez, Ivy Peng, Shirley Moore, Maya Gokhale ECP-MT-1000	Date

Concurrence:

Simon Hammond ECP Hardware Evaluation	Date

Approval:

Dan Hoag Federal Project Administrator	Date

1. INTRODUCTION

High Bandwidth Memory (HBM) will be a prominent feature in the exascale memory landscape, potentially having a larger role than DDRx as the primary on-node memory. HBM is a DRAM stacking technology that vertically stacks multiple DRAM dies and directly connects them with through-silicon vias (TSVs). Fundamentally, HBM and DDR use the same DRAM technology, but their architectures differ. Both support multiple (up to 8) independent channels to access the memory. However, one way that HBM differs from conventional DDR is the width of its interface; each HBM channel uses a 128-bit wide I/O interface and each connection can operate at a data rate of up to 2Gbps per wire. This extremely wide, parallel interface is the key technique used by HBM to deliver such high bandwidth rates. Total bandwidth of a single HBM2 channel can reach 32GB/s with a stack capable of delivering 256GB/s. The access latency of HBM is comparable to that of conventional DDR DRAM technology.

HBM3, expected to be available in the 2019/2020 timeframe, will double the density of the individual memory dies from 8Gb to 16Gb and will allow for more than eight dies to be stacked together in a single chip, with up to 64GB of memory capacity possible. HBM3 will feature twice the peak bandwidth: HBM2 offers 256GB/s of bandwidth per stack and HBM3 will double that to 512GB/s while remaining in a similar power envelope. When employing multiple stacks, the total amount of memory bandwidth available per socket could be several terabytes per second by 2023.

The key question that arises for ECP is to what degree applications might benefit from HBM3's increased bandwidth. In order to benefit, applications must be able to utilize the increased bandwidth. In general, maximum sustained bandwidth can be achieved through sufficient concurrent demand to saturate the memory bus, typically through a combination of prefetch and demand cache misses. Irregularity in a memory access pattern can prevent useful prefetching. Although this problem can be mitigated by increasing independent thread concurrency, the application must have enough parallelism so that the thread count can be increased enough to utilize the bandwidth. Architectural factors also affect bandwidth utilization. HBM3's increased memory bandwidth must be balanced by the bandwidths in the cache hierarchy and by the network-on-chip (NoC) bandwidth. If these bandwidths are under-provisioned, then it may be impossible for the CPU to generate enough concurrent requests to saturate the memory bus regardless of demand from application level parallel execution threads.

In HIHE01-1, "Evaluate a PathForward/Facilities memory-relevant performance study/analysis," we conducted a focused study on the performance differences between HBM2 and HBM3 as revealed through execution of representative benchmarks. We used measurements on an existing many-core system, Knight's Landing (KNL), to calibrate simulator settings, and then performed Structural Simulation Toolkit (SST) simulations of KNL-like CPUs that access future high bandwidth memories. This report documents our findings.

2. METHODOLOGY

As stated above, the goal of this study is to evaluate the performance difference between HBM2 and HBM3 for DOE applications in the context of many-core architectures. Because HBM3 is not yet available, we use simulation to evaluate its performance and calibrate the simulation using the Knight's Landing (KNL) architecture as a baseline many-core model. Starting from an existing architecture enables a more realistic simulation as we can extrapolate from known architectural parameters. The KNL architecture integrates MCDRAM, an HMC variant, rather than HBM2. The two memory technologies are similar in latency and bandwidth, enabling us to reasonably calibrate the simulation even though the hardware uses a different technology.

To simulate the KNL-like CPU, we use the Structural Simulation Toolkit (SST) [1]. SST is a parallel discrete-event simulation framework which enables users to combine simulators for cache, memory, processors, etc. to build up complex architectures such as KNL. We begin with a description of the benchmarks used in this study and then discuss the calibration and simulation methodology in Sections 2.2 and 2.3.

2.1 BENCHMARKS

To stress the memory system capabilities, we selected three well-known memory-intensive benchmarks with different memory access patterns: STREAM, HPCG, and XSBench.

STREAM [2] is a standard benchmark that measures the peak memory bandwidth available on an architecture for four simple multithreaded kernels. STREAM uses OpenMP to perform thread-parallel iterations over large arrays in each kernel, effectively forcing the hardware to access data from memory rather than cache. The four kernels have perform slightly different computations:

1. *Copy* streams through an array, copying it to another.
2. *Scale* has the same memory access pattern as Copy, but multiplies the read value by a constant before writing it.
3. *Add* sums two values from two arrays and writes to a third.
4. *Triad* is identical to Add but scales the first read value before performing the sum and write.

STREAM may be configured to run the set of kernels multiple times to obtain more samples, and after collecting them, STREAM reports the amount of data read and written divided by the fastest run time for each kernel, effectively providing an upper-bound on the available memory bandwidth for each. Because STREAM's kernel access patterns are extremely simple and regular, they represent a best-case scenario for memory-bandwidth bound codes.

HPCG [3] is an open-source benchmark developed as an alternative to High Performance LINPACK (HPL) for ranking HPC systems. It mimics a multi-grid conjugate gradient (CG) solver for sparse systems. CG solvers are widely used in scientific applications for solving systems of linear equations for which the matrix is symmetric and positive-definite. An iterative implementation of a CG solver attempts to find an approximate solution of x for the given equation $Ax = b$, where A and b are known. The iterative procedure starts from an initial guess of x, denoted x_0, gradually moving toward the solution by minimizing the residual, which measures the distance from the solution. Preconditioners are often used to accelerate this procedure by transforming the system into another system with more favorable convergence properties.

HPCG does a fixed number (default 50) of CG solver iterations. Each iteration runs a symmetric Gauss-Seidel preconditioner kernel, *ComputeMG*, followed by multiple invocations of three main functions, *ComputeDotProduct*, *ComputeWAXPBY*, and *ComputeSPMV*. Both ComputeMG and ComputeSPMV demonstrate irregular data access patterns over sparse data structures. Other functions exhibit regular data access patterns. With the exception of ComputeMG, the functions are parallelized with OpenMP in the reference implementation. Thus, ComputeMG dominates execution time due to its serial implementation.

XSBench [4] is a proxy for the Monte Carlo particle transport simulation code OpenMC. XSBench emulates the most time-consuming portion of OpenMC, which involves a high volume of thread-parallel random lookups to a large table with little intervening compute. Thus, it exhibits low locality and a high last-level cache miss rate.

Table 1 shows the default parameters for each benchmark. All benchmarks use 256 threads. To simulate these applications in a reasonable timeframe (i.e., less than 24-48 hours), we simulate just a representative portion of the benchmarks, reducing the problem size as needed, and simulate only the main computation, as described in the *Simulation* column. We validate the scaled down problem sizes by comparing to performance on hardware. Because simulation is a controlled environment with precise statistic tracking, it is less susceptible to performance variability due to outside effects such as operating system noise than the hardware platform.

The *Simulation–fast* column lists parameters that are considerably faster to simulate but yield similar simulated results to the *Simulation* parameters. In the following sections, we use *Simulation* parameters for calibration and *Simulation–fast* parameters for the HBM analysis. For XSBench, the Simulation and Simulation–fast parameters are the same.

2.2 KNIGHT'S LANDING EMPIRICAL MEASUREMENTS

To calibrate the simulated architecture, we ran identical problems on the simulator and native KNL hardware and compared equivalent performance metrics. We targeted the RZ Oz testbed at LLNL, which features two

Table 1: Default and simulated benchmark parameters

Benchmark	Parameter	Default	Simulation	Simulation–fast
STREAM	Array size	100M	100M	10M
	Memory footprint	2.2 GB	2.2 GB	229 MB
	Iterations	10	2	2
HPCG	Input	272x272x136	88x88x88, 272x272x136	88x88x88
	Memory footprint	8 GB	0.5 GB, 8 GB	0.5 GB
	Other modification	Simulate first iteration of CG loop only, as iterations behave similarly		
XSBench	Dataset	large	large	
	Lookups	15M	1M	
	Memory footprint	5.3 GB	5.3 GB	
	Other modification	Simulate cross-section (XS) lookups only		

KNL nodes with MCDRAM. The MCDRAM may either be used in "flat" mode where allocations explicitly target either DRAM or MCDRAM or in "cached" mode where the MCDRAM is essentially treated like a last-level cache shared between all cores. To evaluate solely the performance of the stacked memory subsystem, all our experiments target flat mode and allocate memory exclusively in MCDRAM.

In the calibration study, we measured *core bandwidth* and *memory bandwidth*. Core bandwidth is the effective total bandwidth seen by the executing cores, regardless of whether memory accesses were satisfied by caches or memory. Memory bandwidth, on the other hand, is a measure of the MCDRAM memory controller's traffic. By collecting both metrics, we are able to distinguish between memory access time spent in caches and in MCDRAM, and thus infer what fraction of the application benchmark's memory accesses actually access MCDRAM.

Core Bandwidth, specifically, is the sum of the sizes of all memory micro-operations (reads or writes) issued by all cores. We collect core bandwidth by tracing all instructions that involve memory micro-operations using Intel's PIN tool [5], extracting the read and write sizes of each operation, and summing them up at runtime. We invoke this measurement for specific regions of code, time the same region without instrumentation, and divide the read and write sizes by the uninstrumented time. This way, we obtain accurate counts of bytes read and written at the tradeoff of excessive instrumentation overhead and also collect accurate timing without overhead.

While this metric gives the precise bandwidth used by the issued core instructions, it does not precisely capture data movement within the cache and memory system. Accessed data is often smaller than the cacheline size, in which case sparse accesses incur high memory bandwidth overhead to bring in a full cacheline while denser, data-local accesses may require just a single cacheline transfer. Furthermore, request-for-access (RFO) and other software prefetch operations are counted as memory micro-operations, redundantly counting accesses to prefetched data (i.e., the access is counted twice - once for the prefetch and once for the actual access).

Memory Bandwidth captures memory data movement as the number of bytes read from and written to MCDRAM per second. We collect memory bandwidth using uncore hardware counters for the Embedded DRAM Controller (EDC), the MCDRAM's memory controller [1]. These counters enable separate measurement of cacheline reads and writes to MCDRAM. We sum the reads and writes, multiply by the cacheline size of 64 bytes, and divide by the time for different code regions. While measuring counters incurs relatively low overhead, we again measure time on the unmodified application to mitigate any possible perturbation.

For highly memory-bound applications such as STREAM and applications bound by data contention beyond the last-level cache, memory bandwidth accurately reflects application performance. However, the memory bandwidth of cache-bound and compute-bound applications does not necessarily reflect their performance as they are unable to effectively stress the memory system. For these applications, core bandwidth more accurately reflects performance.

[1] We configure and collect the memory counters using the MemOry Bandwidth (MOB) tool developed at LLNL, which communicates with a privileged daemon to obtain uncore counters securely.

2.3 SST SIMULATION

We used the Structural Simulation Toolkit (SST) to model the KNL architecture with HBM2 and HBM3 in place of MCDRAM. SST integrates numerous cycle-level simulation models for processors, on-chip networks, caches, memories, and other architectural features. Figure 1 shows the simulated KNL architecture. The overall layout is a 6x6 mesh with a directory slice and processor tile at each mesh stop. Along the top and bottom are eight HBM stacks (in contrast to the MCDRAM of actual KNLs), and six DDR4 channels appear along the left and right. For this study, we focus on HBM only and ignore the DDR4 channels.

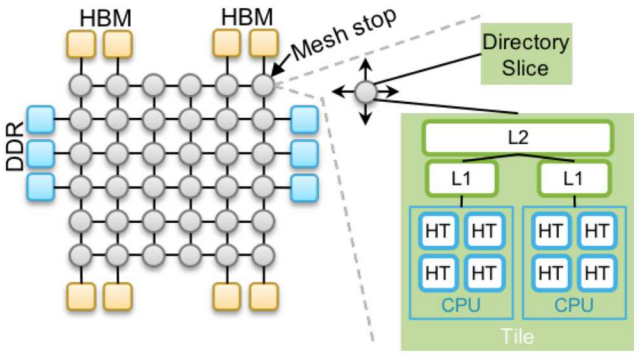


Figure 1: Knight's Landing Architecture. The overall layout is shown on the left; the right shows a zoomed in view of each mesh stop. *(HT: hardware thread)*

To simulate the above architecture, we use the following SST models.

1. **Cores** – *Ariel*. CPU model which uses Intel's PIN [5] binary instrumentation library to dynamically collect a natively running executable's instruction stream.
2. **Caches** – *MemHierarchy* and *Cassini*. MemHierarchy provides coherent cache models. Cassini adds a stride prefetcher.
3. **Directories** – *MemHierarchy*. MESI directory.
4. **Network-on-Chip** – *Kingsley*. Mesh NoC model with input-buffered routers and an XY-routing algorithm.
5. **HBM** – *CramSim*. CramSim provides detailed models of various DRAM-based memory technologies including HBM2. CramSim was developed and validated by IBM as part of Fast Forward 2.

Table 2 lists the major simulation parameters used. To simulate HBM2, we use CramSim's default HBM2 configuration. This configuration leverages the HBM2 pseudo-channel mode. Because the HBM3 JEDEC specification is not finalized, we model HBM3 by extrapolating from HBM2. In particular, HBM3 is expected to double bandwidth while maintaining similar latency. Thus, to model HBM3, we use the CramSim HBM2 model with double the channels (16 instead of 8), half the rows (to keep overall capacity constant), and all other parameters identical. Although HBM3 is also projected to increase capacity, the benefit of capacity is much clearer than the benefit of bandwidth (i.e., determining that an application is memory capacity bound is much more straightforward than determining to what degree it is memory-bandwidth bound). Therefore, we focus on HBM3's bandwidth increase for this study.

While the parameters in Table 2 reflect current KNLs, we observe that because KNL's bandwidth is network-on-chip (NoC) limited, this configuration is unable to fully make use of HBM2 bandwidth. Therefore, we use this configuration for calibration only and define two additional configurations for the HBM comparison: *Baseline* and *Optimized*. We refer to the original configuration as the *Calibration* configuration.

As shown in Table 3, Baseline eliminates KNL's NoC bandwidth limitation by raising the NoC bandwidth to twice (HBM2) and three times (HBM3) that of the Calibration configuration. To determine these

Table 2: Simulation parameters for calibrating the KNL model

CPU	Frequency	1.4GHz
	Cores	72, 4 hardware threads/core
	Issue	2 instructions/cycle
	Outstanding requests	12
	Page size	4KB
Cache subsystem	Coherence	MESI
	Cache line	64B
L1 Cache	Frequency	1.4GHz
	Configuration	Private, 32KB, 8-way set associative, LRU replacement
	Latency	5
	MSHR	12 entries
	Bandwidth	2 requests per cycle
L2 Cache	Frequency	1.4GHz
	Configuration	Semi-private (shared by two cores), 1MB, 16-way set associative, LRU replacement, inclusive
	Latency	
	MSHR	48 entries
	Bandwidth	64B response/cycle
Network-on-Chip	Topology	6x6 mesh. Four parallel networks: 3 control (request, ack, inv/fetch), 1 data
	Flit	36B for data network, 8B for control networks
	Frequency	1.7GHz
	Link bandwidth (control)	12.7 GB/s
	Link bandwidth (data)	57 GB/s
	Input buffers	4 packets (Control: 32B, Data: 288B)
Directory	Configuration	Distributed at 34/36 mesh stops. 2 per HBM, 3 per DDR channel
	Frequency	1.7GHz
	Latency	8 cycles
	MSHR	128 entries
HBM Memory Controller	Frequency	475MHz
HBM (per stack)	Frequency	1GHz
	Capacity	2GB
	Channels	8 channels, 2 pseudo-channels(pCh)/channel
	Bandwidth	2 GB/s
	Banks	4 bank groups of 4 banks each
	Rows	16384
	Access	32B
	Timing parameters	nRAS=33, nCL=20, nRP=14, nRTP=4, nRFC=350, nBL=2, nWR=16, nRTW=18
	Bank policy	Open
	External address mapping	Addresses interleaved across HBM stacks at cacheline granularity
	Internal address mapping	row : col(n-1) : bank : bank group : pChannel : col(0) : burst
	Outstanding requests	1024
DDR4	Not used in this study	

Table 3: Simulation parameter changes for the Baseline and Optimized configurations. The Calibration column shows reference values from Table 2.

	Parameter	Calibration	Baseline–HBM2	Baseline–HBM3	Optimized
CPU	Issue (instr./cycle)	2	2	2	16
	Outstanding requests	12	12	12	24
L1 Cache	MSHR entries	12	12	12	24
	Bandwidth (request/cycle)	2	2	2	Unlimited
L2 Cache	MSHR	48	48	48	64
	Bandwidth (request/cycle)	1	1	1	Unlimited
Network-on-Chip	Frequency (GHz)	1.7	3.4	5.1	6.8
	Control Link bandwidth (GB/s)	12.7	25.3	38	50.7
	Data Link bandwidth (GB/s)	57	114	171	228
Directory	MSHR entries	128	128	128	512
HBM	Channels	8	8	16	8/16
	PseudoChannels / Channel	2	2	2	2
	Bandwidth (GB/s)	256	256	512	256/512
	Rows	16384	16384	8192	16384/8192

parameters we ran STREAM and swept integer multipliers of KNL's NoC bandwidth until STREAM's performance saturated. As such, the Baseline NoC settings are upper limits, not necessarily tight bounds. The third configuration, Optimized, improves the bandwidth and concurrency available from the cores and caches, enabling better use of HBM3. As with the HBM2 to HBM3 modifications, Optimized avoids changing latency parameters. To evaluate the effect of architectural changes and HBM3 independently, we run this configuration with both HBM2 and HBM3.

2.4 CALIBRATING THE KNL SIMULATION MODEL

We established the initial simulation parameters by gathering publicly available information on the KNL and making best guesses where necessary. We then compared performance metrics gathered from the simulated model with HBM2 to KNL hardware equipped with MCDRAM, and iterated on the parameters, benchmarks, and performance counter collection until we were confident that our simulated architecture was reasonably consistent with hardware. Here, we present the final comparison after completing the calibration.

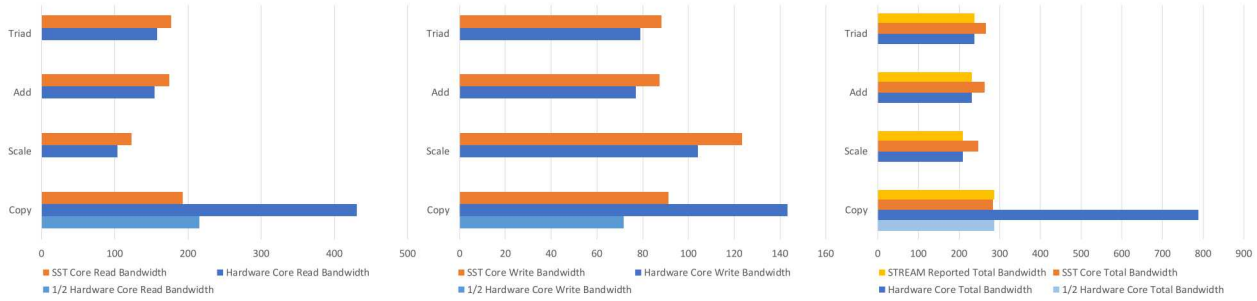
Overall, SST is able to model KNL closely, but not exactly. Some KNL hardware details are not publicly available, for example, hardware prefetching internals, and other architecture-specific details are not configurable through the SST models used, e.g., datatype-specific cache load latency differences. Further, the SST CPU model is "optimistic" where concurrency is concerned—the CPU ignores instruction dependencies, and the caches are perfectly banked so accesses will not conflict unless they access the same address.

The comparison in this section helps us fine tune configuration parameters of the SST models and quantify the differences between the simulated model with HBM2 and existing KNL hardware with MCDRAM.

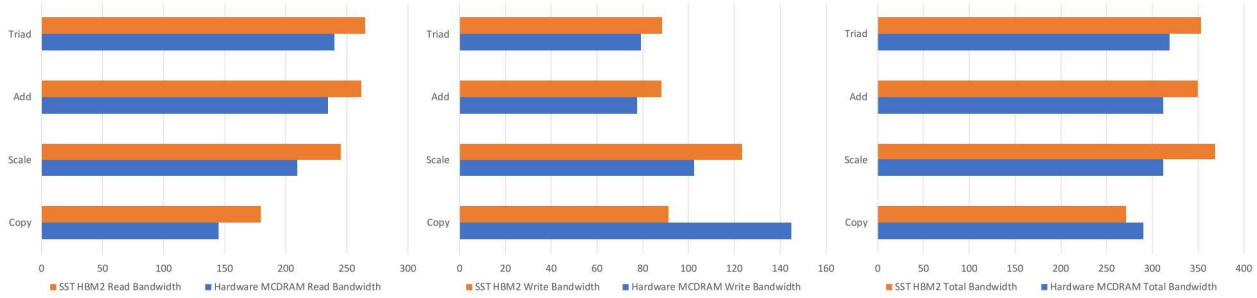
KNL hardware experiments were primarily performed on an LLNL testbed RZ Oz that uses early beta KNL hardware, and our SST configuration simulates a production KNL system with better performance. For reference, we observed up to a 40% improvement in performance metrics when running on later hardware (the Bowman testbed at SNL). As our measurement infrastructure requires root permission (not available on Bowman) we used RZ Oz for the measurements shown below, and simply note that the bandwidth and elapsed time are frequently lower than current KNLs.

STREAM calculates bandwidth by dividing the size of the arrays it streams through by the time taken to do so. Thus, the bandwidth reported is essentially the effective data access bandwidth and closely resembles our core bandwidth metric. For large arrays, which spill out of last-level cache, the STREAM bandwidth also reflects the memory bandwidth metric relatively well.

Using the Default and Simulation benchmark parameters listed in Table 1 for hardware and simulation, respectively, we compare hardware and simulation on STREAM's reported bandwidth, core bandwidth, and memory bandwidth. As shown in Figure 2, nearly all measurements between hardware and SST differed by a factor of approximately 10%, and relative differences were retained between simulation and hardware.



(a) Core bandwidth: Read (left), Write (middle), Combined bandwidth compared to STREAM-reported results (right)



(b) MCDRAM and HBM2 memory bandwidths: Read (left), Write (middle), Combined (right)

Figure 2: STREAM core and memory bandwidths compared between hardware, simulation, and STREAM-reported results

The difference is due to SST's optimistic treatment of concurrency and the hardware's less-than-optimistic performance. The exception is Copy, where the compiler recognizes the computation and invokes a processor-specific implementation of *memcpy*. On KNL, this invocation uses software prefetches (RFO instructions), which cause the Core Bandwidth to include both accesses and prefetches (Figure 2a). We therefore also show the core bandwidth scaled by half, which effectively eliminates the prefetch instructions. The simulator is running on a different processor (Skylake) and invokes a different memcpy optimization in this case. As such, while the kernel is nominally the same, the exact instructions executed differ, leading to different results between hardware and simulation.

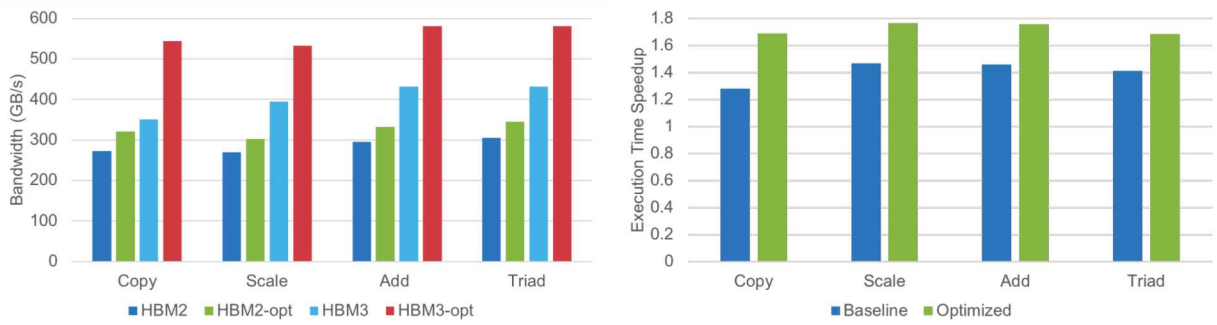
Overall, the hardware's total core bandwidth matched STREAM's reported bandwidth almost identically, and SST's results matched closely as well (Figure 2a, right). For both hardware and simulation, the read and write memory bandwidth measurements (Figure 2b) did not quite match STREAM's reported bandwidth, indicating differences due to cache bandwidth and cacheline size as opposed to memory micro-operation access sizes.

We performed similar calibrations using HPCG and XSBench. In these cases, the differences between SST simulation and KNL hardware were more significant. In HPCG, the highest differences were in subroutines that took very little time and had few memory references. Performance trends for memory-intensive subroutines and serial code remained similar. In XSBench, the differences were not consistent with problem size. The detailed comparison breakdown is shown in Appendix A.

Overall, the calibration process helped tune the KNL CPU parameters. We felt confident that increasing bandwidths and other simulation parameters from the initial configuration will reflect real performance gains (or lack thereof) reasonably well.

3. RESULTS

In this section we analyze the performance impact of HBM3 on each benchmark. As described in the previous section, we evaluate the performance of HBM2 and HBM3 on two configurations: *Baseline* and



(a) Bandwidth as reported by STREAM

(b) Execution time speedup for HBM3 over HBM2

Figure 3: STREAM bandwidth and speedup for the *Baseline* (HBM2, HBM3) and *Optimized* (HBM2-opt, HBM3-opt) configurations

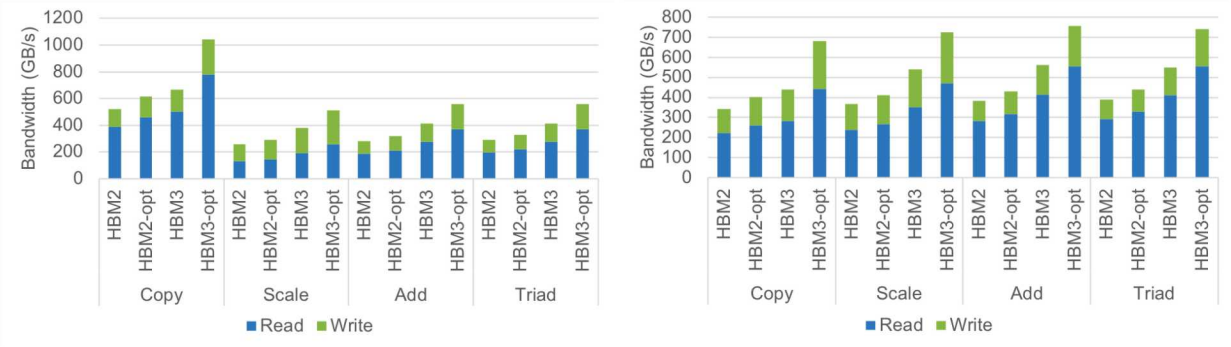
Optimized. Recall that compared to the Calibration configuration, *Baseline* increases the NoC bandwidth to 2X (for HBM2) and 3X (for HBM3) of the KNL's. With this increase, the NoC is not the limiting factor in performance. *Optimized* additionally adds significant concurrency and bandwidth improvements to the cores and caches (Table 3), including higher outstanding request and cache miss limits, wider buses, and more cache accesses per cycle. Section 3.4 more deeply explores NoC performance and the impact of the Optimized configuration's parameters.

3.1 STREAM

In contrast to Section 2.4, where we configured STREAM with 100M element arrays, this section uses 10M element arrays. The smaller size enables faster simulation with similar results.

We begin by examining the bandwidths reported by STREAM. Figure 3a shows the bandwidth in GB/s along the Y-axis for the different architectures and STREAM kernels (X-axis). Comparing the baseline and optimized KNL architectures with HBM2 (HBM2 and HBM2-opt), it is evident from the small 14% difference that the baseline KNL architecture is already well optimized for bandwidth-limited applications such as STREAM. This is expected as KNL was designed for MCDRAM which supports a similar bandwidth to HBM2. Replacing the HBM2 with HBM3 (third bar) leads to moderate average performance improvements of 41% and 24% compared to HBM2 and HBM2-opt, respectively. This implies that for bandwidth-limited applications, HBM3 has the potential to provide benefit. Still, STREAM does not come close to making use of the full 2X bandwidth improvement provided by HBM3. The HBM3-opt bar shows the impact of removing many of the core and cache bandwidth limits. This configuration achieves a 96% improvement on average, compared to the baseline HBM2. We conclude therefore that while HBM3 alone, with significant NoC bandwidth improvements, does benefit STREAM, HBM3's full potential cannot be realized unless substantial improvements are made throughout the architecture. Figure 3b confirms this conclusion. The figure shows the execution time speedup (Y-axis) for HBM3 compared to HBM2 on each kernel and architecture (X-axis). We calculate speedup as the ratio between the execution times of HBM2 and HBM3 for each of baseline and optimized KNL configurations, i.e., $T^{f_i}_{hbm2}/T^{f_i}_{hbm3}$. The left set of bars, Baseline, indicate that without architectural changes, replacing HBM3 with HBM2 yields a moderate 28-47% speedup. By contrast, HBM3 improves execution time on the optimized architecture by a much larger 69-77%.

Figures 4a and 4b show the simulated CPU and memory (HBM) bandwidths, respectively, for the same architectures and kernels shown in the previous figure. Note that the CPU bandwidth differs from that reported by STREAM because the CPU bandwidth includes non-array accesses and, for *Copy*, software prefetches. As expected, the negligible reuse of the STREAM arrays implies similar bandwidth between HBM and CPU, while for Copy, the prefetch requests incur nearly double the number of accesses, roughly doubling CPU bandwidth. Still, neither HBM2 nor HBM3 comes close to the total available bandwidth. Even the HBM2-opt configuration running Triad uses just 439GB/s of the available 2TB/s. The HBM3-opt configuration fares worse, consuming only 758GB/s of its 4TB/s across the eight HBM channels. This raises a question as to whether further optimizations can improve bandwidth utilization, even for the HBM2 case.



(a) CPU bandwidth

(b) HBM bandwidth

Figure 4: STREAM: Read and write bandwidths for HBM2 and HBM3 in the baseline and optimized KNL configurations.

Overall, we conclude that HBM3 performs moderately well with the baseline configuration, and very well with the optimized architecture. However, we are concerned that attaining the kind of processor enhancements needed to support HBM3 will be challenging. In Section 3.4, we tighten the bounds on the HBM3-opt parameters to provide deeper insight into the scale of improvements that will be necessary.

3.2 HPCG

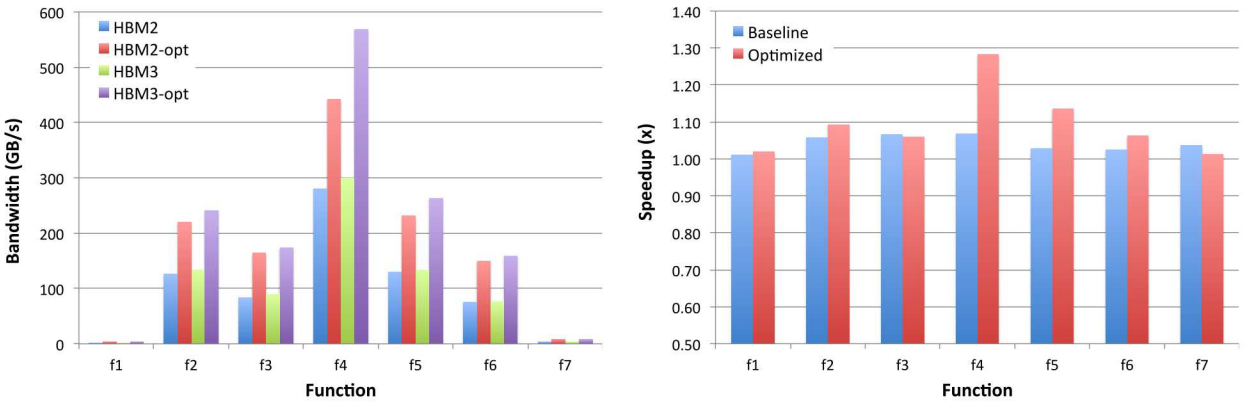
In this section, we report SST simulation results for the problem size $nx = 88$, $ny = 88$, and $nz = 88$, which has a 540 MB memory footprint. We first report the total simulated memory bandwidth for the four configurations and the projected speedup. Then, we break down the CPU and HBM bandwidths into their read and write components.

Recall that HPCG has a number of different functions, each of which has a distinct memory access patterns. Because the preconditoner function, ComputeMG dominates execution time, we present results in this section broken down by function. Table 4 shows the function invocations for a single HPCG iteration and provides shorter labels f1-f7 used in the following figures.

Table 4: List of the main equations, their respective function calls, and their dominant data access patterns in one CG iteration of HPCG.

S/N	Equation	Function	Access Pattern
f1	Gauss-Seidel preconditioner	ComputeMG	Irregular
f2	$p = beta * p + z$	ComputeWAXPBY	Regular
f3	$rtz = r * z$	ComputeDotProduct	Regular
f4	$Ap = A * p$	ComputeSPMV	Irregular
f5	$alpha = p * Ap$	ComputeDotProduct	Regular
f6	$x = x + alpha * p$ and $r = r - alpha * Ap$	ComputeWAXPBY(x2)	Regular
f7	$normr = r * r$	ComputeDotProduct	Regular

Figure 5a presents the measured memory bandwidth for each configuration. The highest total memory bandwidth reaches 568 GB/s in the optimized configuration with HBM3. In contrast to STREAM, we note that optimized HBM2 performs much better than baseline HBM3. In fact, HBM2 in the optimized setup shows the second highest performance in all functions, indicating that the baseline architecture does not enable HPCG to fully exploit HBM2 bandwidth. Under the optimized processor configuration, HBM3 achieved from 1% to 28% higher bandwidth than HBM2 in each function. However, in the baseline configuration, we observe minimal speedup from HBM3 compared to HBM2, improving bandwidth over HBM2 by just 1% to 6%. From the performance gap between HBM3 and HBM2 in the two processor architectures, we conclude that HPCG is very sensitive to processor architecture when using HBM3.



(a) Total memory bandwidth

(b) Execution time speedup for HBM3 over HBM2

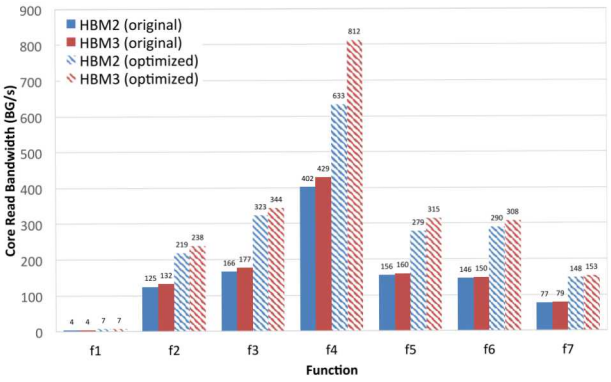
Figure 5: Performance comparison for HPCG's main functions using HBM2 and HBM3 in the baseline and optimized KNL configuration.

Figure 5b presents the execution time speedup when using HBM3 compared to HBM2 for the two processor setups. The most significant execution time reduction is 28%, achieved on the Optimized configuration for $f4$. In contrast, $f1$, which has a similar data access pattern, shows only 1% and 2% improvements in Baseline and Optimized, respectively. The low impact of HBM3 on $f1$ is likely due to its single-threaded implementation. There is not enough concurrency in $f1$ to saturate the memory bandwidth from the core to the memory system. The contrast between $f1$ and $f4$ highlights the importance of concurrency for exploiting HBM3 performance.

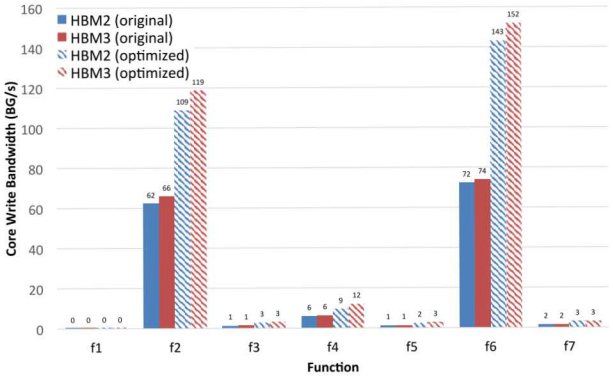
In the baseline configuration (blue bars in Figure 5b), functions $f1 - f6$ achieved speedups of 1%-7% when using HBM3. With the optimized configuration, the speedups ranged from 5% to 28%. $f7$ shows the least improvement mainly due to its good data locality (its last level cache miss rate is 2.7% from SST simulations) that makes it nearly memory insensitive. Functions $f2$, $f3$, $f5$, and $f6$ exhibit regular data access patterns enabling the cache prefetchers to effectively bridge the bandwidth gap from the different HBM generations. One important insight is that optimizing the core side configuration is critical to enabling a memory-sensitive function like $f4$ to benefit from HBM3. Without optimization on the core side, $f4$ achieved only 7% speedup with HBM3. After optimizing the NoC bandwidth and number of MSHRs, $f4$ achieved a 28% speedup. Future sensitivity analysis is needed to provide more accurate insights on the specific core modifications that applications similar to $f4$ will benefit from (with or without HBM3).

To evaluate the gap from the peak performance, we also break down bandwidth usage by read and write, and report core bandwidth in Figure 6. Function $f4$ exhibits the highest read bandwidth for both core and memory, reaching 812 (Figure 6a) and 546 GB/s (Figure 6c) on the optimized HBM3 configuration. Comparing to STREAM's bandwidth under the same configuration, HPCG reaches a high utilization of the sustainable memory bandwidth in the system. Other functions exhibit relatively lower read bandwidth than $f4$, ranging from 153 GB/s to 344 GB/s in core bandwidth (single-threaded $f1$ exhibits 7 GB/s). $f4$ (*ComputeSPMV*) has mostly irregular data accesses in both the Gauss-Seidel kernel and sparse matrix-vector multiplication. The performance improvement of $f4$ indicates that memory-sensitive applications with similar characteristics can substantially benefit from the high bandwidth of HBM3 when there is enough concurrency on a suitable processor architecture.

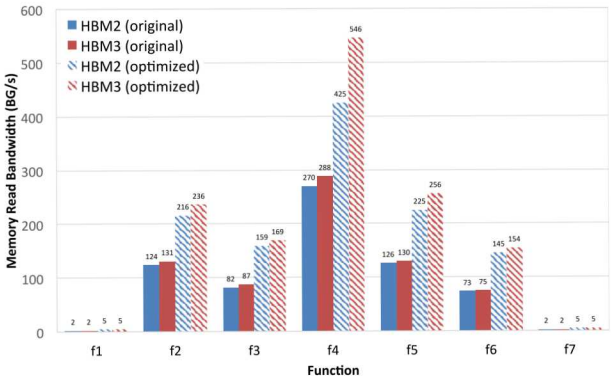
The write bandwidth from core and memory, however, differs in distribution as shown in Figure 6b and Figure 6d. In particular, functions $f2$ and $f6$ exhibit high core write bandwidths, reaching 119 and 152 GB/s, respectively, in the optimized HBM3 configuration. Both $f2$ and $f6$ call function *ComputeWAXPBY* which has more write accesses relative to other functions. The highest write bandwidth from memory comes from $f4$ (Figure 6d), indicating that more dirty cache lines are written back in this function than in other functions. Note that $f4$ (*ComputeSPMV*) actually has a lower write ratio than $f2$ and $f6$. The difference in core and memory write bandwidths indicates that functions like *ComputeSPMV* stress not only read bandwidth but also write bandwidth because they need to evict more dirty cache lines to bring in new data.



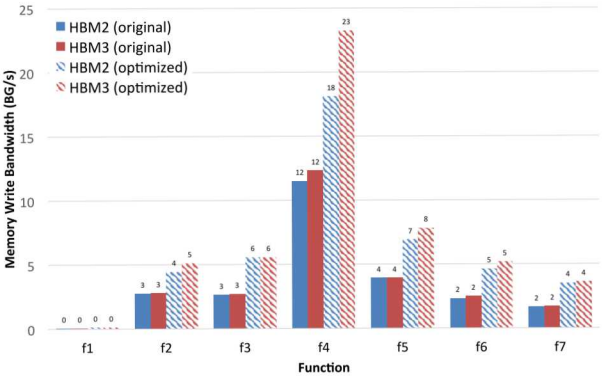
(a) CPU read bandwidth (GB/s)



(b) CPU write bandwidth (GB/s)

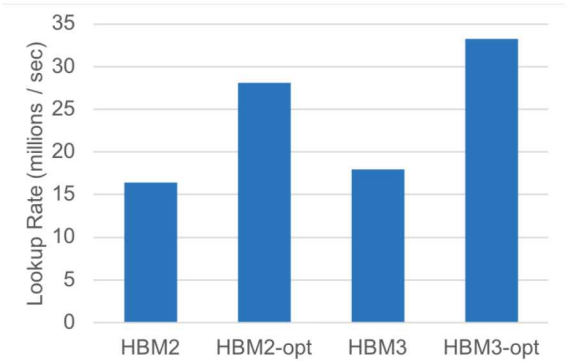


(c) HBM read bandwidth (GB/s)

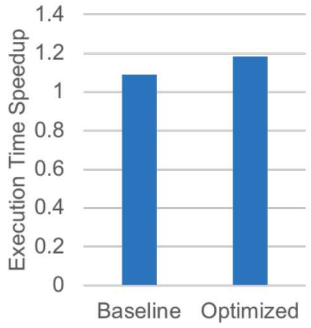


(d) HBM write bandwidth (GB/s)

Figure 6: CPU and memory bandwidth in HPCG main functions using HBM2 and HBM3 for the baseline and optimized KNL configurations.



(a) Performance of XSBench measured as the lookup rate in millions/second



(b) Execution time speedup of HBM3 compared to HBM2 for each architecture

Figure 7: XSBench lookup rate and speedup for the *Baseline* (HBM2, HBM3) and *Optimized* (HBM2-opt, HBM3-opt) configurations

In summary, HPCG demonstrates the advantage of HBM3 over HBM2 for memory-sensitive functions like *ComputeSPMV* and also moderate improvements in other memory-intensive functions. To fully unleash the benefits of HBM3, the application needs to provide enough concurrency. Finally, core side modifications, such as higher NoC bandwidth and larger MSHR banks, are critical for fully exploiting the performance benefits of HBM3.

3.3 XSBENCH

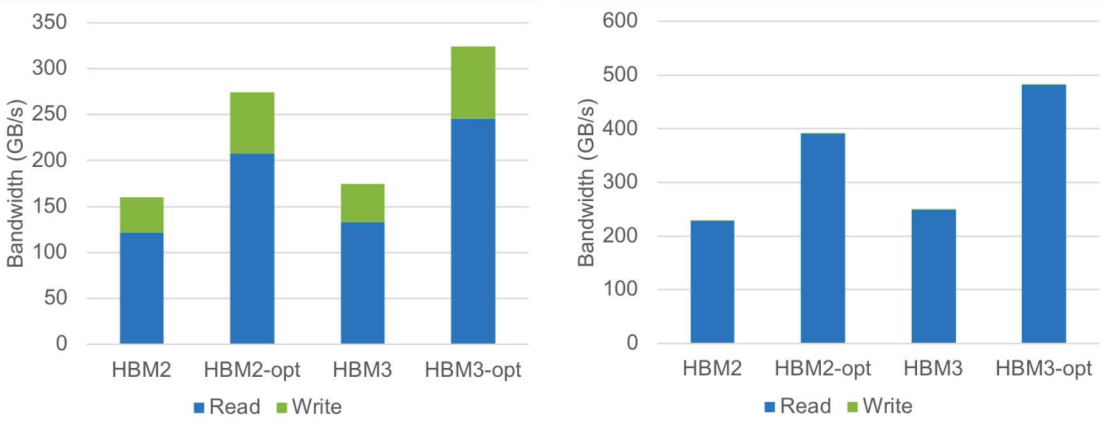
We now evaluate the performance of XSBench's cross-section lookup computation with next-generation HBM. Figure 7a shows the lookup rate, measured as millions of lookups per second, for XSBench executing on each simulated architecture. Comparing HBM2 and HBM3, it is evident that HBM3 alone benefits XSBench very little, improving lookup rates by just 9%. In contrast, the optimized architecture significantly improves performance over the baselines for both HBM2 and HBM3, by 71% and 86%, respectively.

This occurs because the baseline architecture does not support enough concurrency to hide the latency induced by XSBench's intensive random memory access pattern, even for HBM2. In contrast to STREAM, XSBench has a very high cache miss rate (i.e., 78% L2 miss rate versus 53% for STREAM), putting significantly higher pressure on non-memory resources like MSHRs. Thus adding HBM3 to the optimized architecture enables a larger benefit (18%) than adding it to the baseline architecture (9%).

Overall, the optimized architecture with HBM3 increases the lookup rate by 109% compared to the HBM2 baseline. From this graph, it is clear that for HBM3 to benefit XSBench-like applications, significant core, cache, and network enhancements need to be made.

Further, the execution time speedups shown in Figure 7b confirm that HBM3 gives little benefit to XSBench. With the optimized architecture, HBM3 improves execution time by a very modest 18%, while for the baseline architecture the benefit is even smaller, at just 9%.

Figures 8a and 8b show the CPU and HBM bandwidth, respectively, achieved by the various architectures. As expected from the previous result, HBM3 uses very little bandwidth beyond HBM2, while HBM2-opt and HBM3-opt significantly outperform their respective baselines. HBM read bandwidths are higher than CPU bandwidths due to the random access pattern's lack of spatial locality–on a miss, the HBM provides a 64B cacheline but the core does not access the entire line. Further, the disparity in write bandwidth between CPU and HBM occurs because the majority of XSBench's writes are to the same few memory addresses. To prevent the compiler from optimizing out XSBench's primarily read-only lookup function, the benchmark copies the function's return values to a small array, overwriting the array for each function call. Hence, while a number of writes occur from the CPU's perspective, the HBM incurs very few writebacks of dirty cache blocks.



(a) CPU bandwidth

(b) HBM Bandwidth

Figure 8: XSBench: Read and write bandwidths for HBM2 and HBM3 in the baseline and optimized KNL configurations.

3.4 SENSITIVITY TO NETWORK, CORE, AND CACHE OPTIMIZATIONS

From the preceding results, it is evident that HBM3 has moderate to no benefit without significant improvements to the core, cache, and network architectures. In this section, we seek to better quantify some of these improvements. For brevity, we evaluate only for STREAM and look at the best performing architecture–HBM3-opt. First, we evaluate sensitivity to NoC bandwidth and then perform some experiments to determine tighter upper bounds on the parameters that were changed for the optimized architecture.

3.4.1 Network-on-Chip Bandwidth Sensitivity

Figure 9 shows the reported bandwidths for each STREAM kernel (Y-axis) as network-on-chip (NoC) frequency is increased along the X-axis. We report NoC frequency and not bandwidth because KNL's four networks have the same clock frequency but different bandwidths, depending on their flit size. A flit is the unit of data transferred each cycle; generally, transferring a packet requires one or more flits. Bandwidth is simply the product of flit size and NoC frequency. While one could increase the flit size to achieve higher bandwidth, increasing beyond the size of a packet implies either wasted bandwidth or extra complexity to transfer multiple packets in parallel. Hence we study the impact of increasing NoC frequency, keeping flit size constant. As reported in Table 2, the data network uses a 36B flit size, equal to half a data packet, whereas the three control networks use 8B flits, corresponding to a full control packet. Figure 9 indicates that the NoC frequency has a considerable effect on STREAM's performance. Between the baseline KNL frequency of 1.7GHz and about 5GHz, every 500MHz increase in clock rate yields a 50GB/s improvement in the STREAM numbers. This is substantial, considering that a 500MHz clock improvement theoretically yields a 51GB/s bandwidth improvement, but only 34GB/s of that bandwidth goes towards transferring data payloads. While achieving network clock rates of 5GHz will be extremely challenging, other NoC improvements may help bridge the gap. For example, improvements around better network management, congestion control, and routing algorithms might contribute to higher bandwidth. This study highlights the need for comprehensive NoC research to faciliate high-bandwidth memory systems.

3.4.2 Tightening the Optimized KNL Parameters

The optimized KNL architecture makes considerable bandwidth improvements to the core and cache subsystem. The parameters were chosen either to double the amount of throughput available in the baseline KNL, mirroring the doubling of HBM bandwidth, or to be unlimited so as to form an upper bound on performance. To identify more realistic upper limits, in this section we look at the effect of tightening the parameters. To that end, we ran STREAM on the HBM3-opt architecture, reducing each optimized parameter independently. These parameters can be divided into two groups - those that affect the rate of data movement between

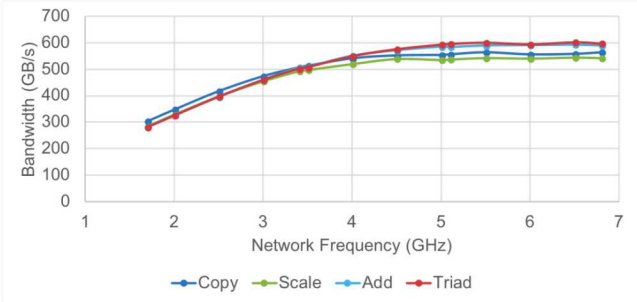
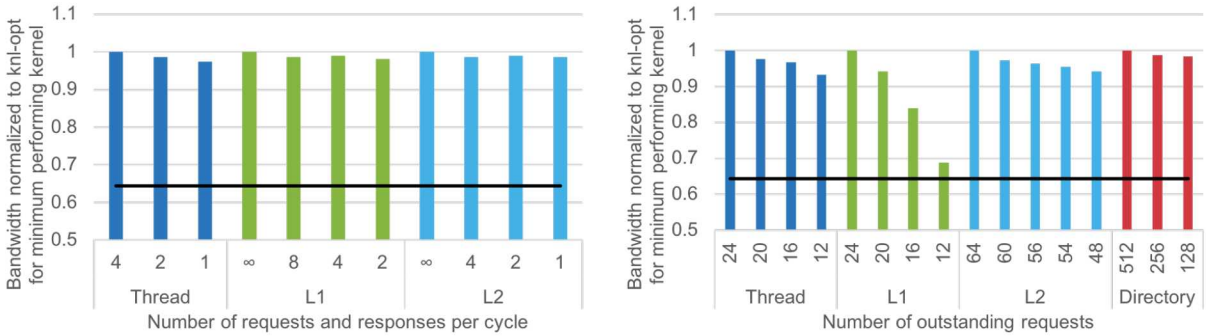


Figure 9: STREAM bandwidth as a function of network frequency (bandwidth)



(a) Rate parameters

(b) Volume parameters

Figure 10: Sensitivity to optimized parameters measured as bandwidth normalized to the HBM3-opt configuration. For each configuration, only the minimum normalized bandwidth across all four kernels is reported. The black line shows the performance of the baseline HBM3-base configuration for the worst performing kernel (Copy).

memory levels, and those that affect the volume of requests in flight. Figure 10a shows the parameters affecting the data movement rate as they vary between the optimized and baseline values. It is clear that varying these parameters has little effect on the overall performance, and in fact, issue rates equal or similar to the baseline case are sufficient. In contrast, Figure 10b shows the parameters supporting a higher volume of requests in flight, namely the number of outstanding requests allowed per thread and the number of MSHRs per cache and directory. Note that our simulation approximates hardware threads by modeling them as independent cores which share an L1. Therefore, 20 outstanding requests per thread is roughly equivalent to an 80 entry load-store queue in a conventional architecture where the resource is shared across hardware threads. As shown in Figure 10b, STREAM's performance is much more sensitive to the volume of requests in flight. Overall, Figure 11 shows the effect of combining the baseline data movement parameters with the optimized data volume parameters. We call this case, *Tight* and compare to the *Baseline* and *Optimized* configurations. As the figure shows, Tight is comparable to Optimized. Still, because STREAM is the least sensitive to the optimized architecture among the three benchmarks studied, Tight likely forms a lower upper bound with the actual scope of optimizations needed to support HBM3 falling somewhere between the Tight and Optimized configurations. We leave further exploration of the architecture for future work, but note that even the tightened parameters may be hard to achieve. Larger MSHR and load-store queues incur higher latency, energy, and area costs which may not be feasible.

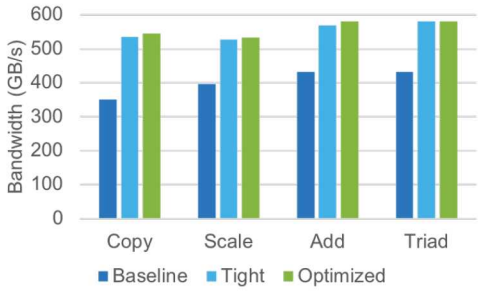


Figure 11: STREAM bandwidth for the baseline, optimized, and tightened parameter sets

4. CONCLUSION AND RECOMMENDATIONS

The purpose of this study was to evaluate the performance benefit of next-generation HBM3 compared to HBM2 for many-core CPU architectures. HBM3 is expected to arrive on the market in the 2019/2020 timeframe, and while the JEDEC specification has not been finalized, it is currently expected to provide double the bandwidth (to 512GB/s) at similar latency to HBM2. It is also expected to provide higher capacity. In this study, we focused on the performance benefit from the increased bandwidth, not capacity. We expect that increased capacity will benefit applications that already show benefit from MCDRAM on KNL. The performance impact of higher bandwidth is less clear.

To evaluate the effect of doubling HBM2 bandwidth, we calibrated and simulated a KNL architecture running three benchmarks: STREAM, HPCG, and XSBench. These benchmarks were selected because they put high pressure on the memory system with irregular memory access patterns and high ratios of memory accesses to compute. We chose the KNL architecture as a baseline because it is representative of the many-core approach to CPUs, and we are able to calibrate our model against existing KNL systems. For throughput-oriented architectures such as GPUs, which support extremely high bandwidth for specific classes of kernels, the results may not hold. Through the calibration we found that for bandwidth-bound codes, the SST model was accurate, but because the CPU and cache models are optimistic with respect to concurrency, the simulation is less accurate for latency-bound code where instruction dependencies limit available concurrency. Note that this inaccuracy means the simulation is more likely to overestimate than underestimate performance gains with HBM3.

Overall, the major observations we made in this study are:

1. The current KNL architecture, simulated HBM2, and simulated HBM3 are all limited by network-on-chip (NoC) bandwidth.
2. With the NoC bandwidth limitation removed, HBM3 provides modest benefit in the best, bandwidth-bound, case (STREAM), and very little benefit for the other cases where latency and instruction dependencies limit concurrency.

Based on these observations, we make the following recommendations:

1. HBM3 should only be employed if other improvements are made to increase the NoC bandwidth and number of outstanding memory requests supported by the hardware.
2. Increasing NoC bandwidth and number of supported outstanding memory requests should take precedence over increasing the bandwidth of the memory architecture.
3. Increasing HBM capacity will likely improve performance for memory-bound applications more than increasing HBM bandwidth.

Thus, we conclude that HBM3 alone, in the absence of substantial improvements to the node architecture, will yield little performance benefit. In that case, the increased capacity should be seen as the primary

advantage of HBM3. If capacity does not improve between HBM2 and HBM3, and resources are limited, we further recommend that improvements to the node architecture take precedence over pursuing HBM3.

ACKNOWLEDGEMENTS

We thank Kento Sato (sato5@llnl.gov) for his development effort on creating the PIN utility for tracing memory micro-operations and Kathleen Shoga (shoga1@llnl.gov) for her effort extending the MemOry Bandwidth (MOB) utility to collect uncore memory bandwidth counters from MCDRAM on KNL systems. SST has many contributors, but we would like to extend special thanks to Scott Hemmert (kshemme@sandia.gov) who developed and refined the Kingsley network-on-chip model for this study and Clay Hughes (chughes@sandia.gov) who performed some of the early SST/KNL validation studies.

REFERENCES

[1] A. F. Rodrigues, K. S. Hemmert, B. W. Barret, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, 2011.

[2] STREAM: Sustainable memory bandwidth in high performance computers. `https://www.cs.virginia.edu/stream`. Accessed: 2018-04-16.

[3] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. HPCG benchmark: a new metric for ranking high performance computing systems. *Knoxville, Tennessee*, 2015.

[4] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[5] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

A. HARDWARE VS. SIMULATION PERFORMANCE RESULTS

A.1 HPCG

HPCG hardware and simulation execution time matched closely for function ComputeMG, which represented 98+% of the run. Memory bandwidths were within 7% of each other, and SST core bandwidth was about 20% higher than hardware, which is not unreasonable considering the difference in instructions invoked between KNL hardware and SST on a different architecture. The remaining functions differed significantly but ran for little time (75-2000 us) and produced too little data to provide conclusive results.

Table 5: List of the main equations, their respective function calls, and their dominant data access patterns in one CG iteration of HPCG.

S/N	Function	Hardware Time (us)	Simulated Time (us)
f1	ComputeMG	636385.85	644933.77
f2	ComputeWAXPBY	1252.56	98.37
f3	ComputeDotProduct	371.35	75.06
f4	ComputeSPMV	1981.6	1005.46
f5	ComputeDotProduct	860.18	83.14
f6	ComputeWAXPBY(x2)	524.11	169.05
f7	ComputeDotProduct	864.91	86.28

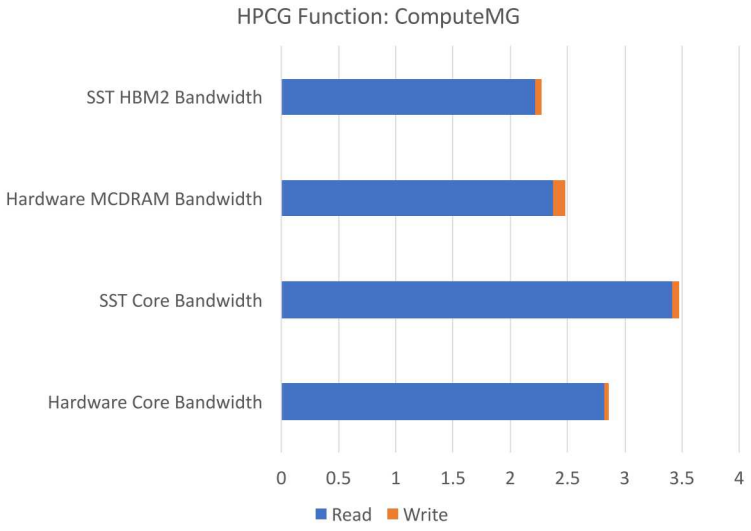


Figure 12: Core and memory bandwidth metrics for HPCG function $f1$, *ComputeMG*

A.2 XSBENCH

In XSBench, the ratio of core to memory bandwidths matched closely between hardware and simulation, as did memory traffic quantities except core writes. However, execution time was overestimated by simulation by a factor of 2.25x for problem sizes with stable performance results. The self-reported performance metric, lookups/second, reached a stable value of around 15K for problem sizes of 100K and above in simulation. In hardware, lookups/second stabilized at 1M+ lookups at around 6.6K. XSBench involves a significantly higher degree of concurrency and instruction dependencies than STREAM or the ComputeMG function in HPCG.

Table 6: XSBench runtimes and reported lookups/sec in hardware and simulation

Problem Size	Metric	Hardware	Simulated
100K lookups	Runtime	79 ms	6.69 ms
1M lookups	Runtime	152 ms	67.53 ms
100K lookups	Lookups/Sec	1,261,777	15,056,256
1M lookups	Lookups/Sec	6,585,566	14,818,913

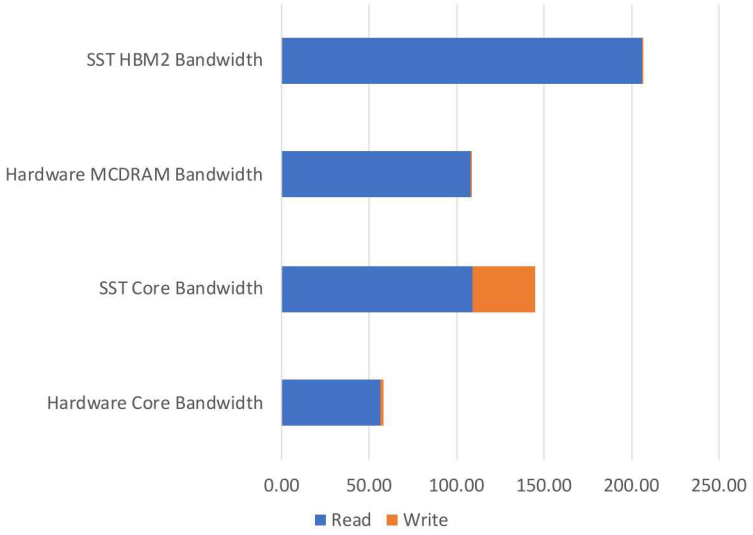


Figure 13: Core and memory bandwidth metrics for XSBench