Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks

Tuowen Zhao*, Samuel Williams†, Mary Hall*, Hans Johansen†
*School of Computing, University of Utah
Email: {ztuowen,mhall}@cs.utah.edu
†Computational Research Division, Lawrence Berkeley National Laboratory
Email: {swwilliams,hjohansen}@lbl.gov

***Abstract*—Achieving high performance on stencil computations poses a number of challenges on modern architectures. The optimization strategy varies significantly across architectures, types of stencils, and types of applications. The standard approach to adapting stencil computations to different architectures, used by both compilers and application programmers, is through the use of iteration space tiling, whereby the data footprint of the computation and its computation partitioning are adjusted to match the memory hierarchy and available parallelism of different platforms. In this paper, we explore an alternative performance portability strategy for stencils, a data layout library for stencils called bricks, that adapts data footprint and parallelism through fine-grained data blocking. Bricks are designed to exploit the inherent multi-dimensional spatial locality of stencils, facilitating improved code generation that can adapt to CPUs or GPUs, and reducing pressure on the memory system. We demonstrate that bricks are performance-portable across CPU and GPU architectures and afford performance advantages over various tiling strategies, particularly for modern multi-stencil and high-order stencil computations. For a range of stencil computations, we achieve high performance on both the Intel Knights Landing (Xeon Phi) and Skylake (Xeon) CPUs as well as the NVIDIA P100 (Pascal) GPU delivering up to a $5\times$ speedup against tiled code.**

***Index Terms*—stencil, performance portability, data layout, Roofline, GPU, KNL, Skylake**

I. Introduction

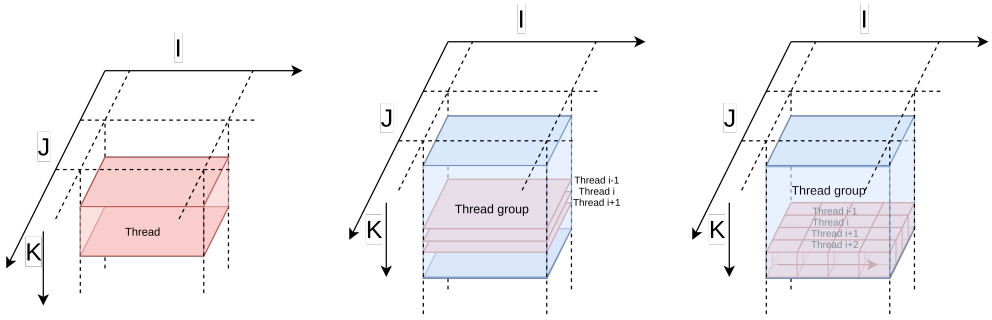
Stencil computations are widely used in scientific applications to solve partial differential equations using the finite difference or finite volume methods, where the derivative at each point in space is calculated as a weighted sum of neighboring point values (a "stencil").

The optimizations required to achieve high performance on stencil computations are greatly affected by a stencil's order of accuracy. Low-order discretizations result in smaller stencils that have limited data reuse, are typically bound by memory bandwidth, and thus underutilize the compute capability afforded by manycore, wide vector, and GPU architectures. Much of the prior work in this field has been based on lower order stencils and has thus focused on techniques to reduce data movement [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], and some of these even seek to optimize in the time domain ("2.5D") to achieve more FLOPS per byte moved from memory. Recognizing that processor architectures are becoming more compute-intensive [13], computational scientists are increasingly turning to high-order schemes that perform more computation per point (more compute intensive) but can attain equal error with larger grid spacings (smaller arrays and thus less total data movement). Although high-order stencils inevitably result in higher arithmetic intensity, they place higher pressure on the register file, cache, TLBs, and inter-process communication. As such, optimizations that eliminate redundant loads/stores and computation have been developed [14], [15], [16].

The optimization strategy also varies significantly across architectures and based on application context (e.g., multi-stencils). In practice, a high-performance stencil must incorporate architecture-specific optimizations to: (1) reduce data movement at multiple levels of the memory hierarchy (registers, caches, TLBs); (2) exploit parallelism at multiple levels (across domains, nested threading, and fine-grain SIMD parallelization); and (3) avoid redundant loads/stores and computation for stencils that exhibit high arithmetic intensity.

A desirable goal is to achieve *performance portability* of applications that incorporate complex stencils, whereby a source code can be expressed at a high level that represents the computation, and then automatically mapped to architecture-specific implementations for differing target architectures. Many previous works achieves this through the use of a domain-specific compiler that automatically generates architecture-specific code from a stylized stencil specification, where a subset of these support both CPU and GPU architectures [17], [18], [19], [20], [21], [22], [23].

Our work could be thought of as providing an embedded domain-specific language ("eDSL") implementation, but it has two distinguishing features over prior work. First, the central underlying abstraction for achieving performance portability of complex stencil computations is a data layout library called *bricks* that decomposes a stencil's grid domain into small, fixed-size multi-dimensional subdomains, an approach to fine-grain data blocking [24], [25], [26]. Although the elements within each brick are stored contiguously in memory, the bricks comprising a subdomain need not be stored in the typical i-major order. Rather, physical ordering is implementation-specific and logically neighboring bricks are represented by an adjacency list. This flexible data layout makes it possible for brick code to adapt automatically to different architectures and application contexts simply by adjusting the data footprint using autotuning. As compared to tiling approaches,



(a) 2D (b) 3D (c) 6D

Fig. 1: Different spatial tiling schemes used in the comparison on CPUs. Sizes for each dimension of the rectangular 3D region are tuned. Appendix B contains the code with OpenMP and Intel Compiler pragmas.

the use of bricks provides a number of benefits on stencils including improved memory hierarchy and TLB utilization, accelerated data copies, and improved instruction-level parallelism. Bricks are related to earlier fine-grained data blocking approaches [24], [25], [26], but this is the first work that combines support for both CPUs and GPUs.

In summary, the key contribution of this paper is the extensive measurements that demonstrate brick as an abstraction for memory hierarchy optimization, vectorization, threaded parallelism and the role of the brick data layout in achieving performance portability across CPUs and GPUs. Across a broad range of stencil computations, we achieve high performance for both the Intel Knights Landing (Xeon Phi) and Skylake (Xeon) CPUs as well as the NVIDIA P100 (Pascal) GPU sometimes significantly outperforming tiled code.

II. Architecture-Specific Adjustment of Data Footprint: Tiling vs. Bricks

From an architecture perspective, as stated previously, performance of stencil computations is driven by a number of factors including data movement through the memory system, including bandwidth requirements and TLB locality, thread-level parallelism and vectorization. Here, we examine how loop structure and code generation can synergize with architecture to minimize the impact of each of these.

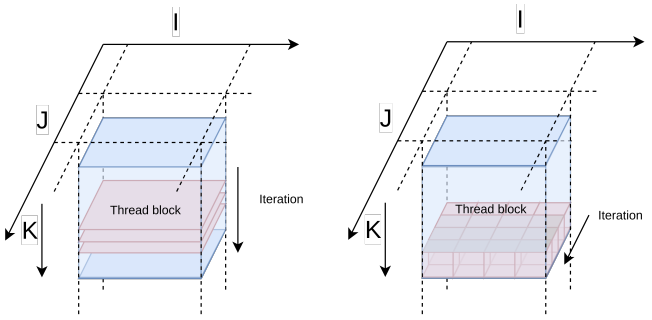
Naively, a typical 3D stencil computation walks through memory in a unit-stride fashion (inner $i-$loop is unit-stride). This code will only perform well on hardware platforms that can cache a working set proportional to the product of stencil diameter dia and the square of the problem dimension (i.e., N^2). Further, the hardware must hide memory latency when presented with a number of data streams corresponding to the square of the stencil diameter (i.e., dia^2); e.g., dia is 3, representing $+1$, 0, and -1 in the case of the 27-point stencil. The separate dia^2 streams arise from the 2D projection of the stencil onto the j-k plane (the plane normal to the streaming axis). The hardware must also maintain at least dia^2 page entries in the TLB. Failure on any one of these aspects can increase data movement or decrease effective bandwidth.

Regarding fine-grain parallelism, both SIMD ISAs on CPUs and memory coalescing on GPUs inexorably lead both architectures to operate on chunks of points in the unit-stride. These units of work are mapped to vectors on CPUs and warps on GPUs. How these units of work stream through the global problem determines cache locality, data movement, latency hiding, and bandwidth.

Modern CPU architectures use hardware stream prefetchers to hide memory latency and maximize memory bandwidth. To make effective use of a stream prefetcher, each core must present a few (often less than 32) unit-stride address streams to memory, which result in sequences of cache misses. To avoid costly TLB misses, these streams should run for at least a TLB page (512 doubles on the Intel KNL Xeon Phi). On GPU architectures, memory latency is hidden with massive thread parallelism. Nevertheless, TLB locality remains important although typical GPU page sizes are much larger and L1 TLBs are shared among warps. On the NVIDIA Tesla P100, the L1 TLB is shared across a Texture Processing Cluster (TPC) that consists of two SMs, and the L2 is shared across the all TPCs [27].

To ensure that the working set is less than cache capacity, thereby ensuring compulsory cache misses dominate capacity cache misses, 3D loops representing stencil computations are typically tiled to create small working sets. For example, tiling by $N/2$ in the i- and j-dimensions reduces the cache working set size by $4\times$ (reuse distance). Smaller tiles generally produce smaller working sets, but there is a lower limit based on the stencil diameter. Unfortunately, tiling in the $i-$dimension is anathema to the demands of stream prefetchers on CPUs and is thus rarely employed (with the consequence of increased cache requirements). Instead, a more typical 2D tiling in the j- and k-dimensions is widely used for CPUs, with the i-dimension vectorized, as in Figure 1(a). An improvement on this scheme is the 3D tiling scheme of Figures 1(b), due to Rivera et al. [28]. This version improves reuse if N is large, and supports nested parallelism, which is desirable for larger numbers of cores. Conversely, memory coalescing and massive thread parallelism on GPUs often incentivizes tiling in the $i-$dimension as well (ignoring TLB effects); a common GPU 3D tiling strategy is shown in Figure 2(a.

Although higher stencil diameters attain high-order nu-



(a) 3D (b) 6D

Fig. 2: Tiling schemes for GPUs as used in our comparisons. Appendix C contains the code using CUDA.

merical properties as well as high arithmetic intensity, such stencils place immense demands on the number of stream prefetchers and the number of TLB entries required. For dense stencils, these structures must scale proportional to dia^2. This high pressure can lead to ineffective latency hiding or high TLB miss rates for even moderately large stencils. To mitigate this pressure, application developers often calculate the forward and backward halves of the stencil in separate loop nests. This separate calculation reduces prefetcher and TLB requirements, but it also reduces cache locality and increases data movement. Performance can be improved (but rarely optimally) by balancing these contending forces.

Rather than only employing tuning to balance these forces, in this paper, we employ a new data layout *bricks* that decomposes the stencils' input and output grids into small, fixed-size multidimensional subdomains that are stored contiguously in memory. We assume that the dimensions of the brick's subdomain is greater than the stencil radius. As such, a stencil application over a brick's subdomain will only use as many prefetch streams and TLB entries as a 27-point stencil. Bricks are then able to exploit the multidimensional reuse inherent in stencils to maximize memory bandwidth, minimize cache working set sizes, and minimize TLB pressure for a range of stencil diameters.

III. Bricks as Aggregate Unit of Parallel Work

The previous section focused on how tiling strategies or alternatively, bricks, adapt a stencil's data footprint to target architectures; this section considers how fine- and coarse-grain parallelism is also adapted to different target architectures using bricks. A stencil computation over a domain is composed of separately applying the stencil to each brick of a domain. The computation within each brick is the collection of stencil applications over the subdomain of the brick. Parallelizing the stencil application can thus be separated into exploiting fine-grain parallelism within each brick, and coarse-grain parallelism across a collection of bricks. The brick is then a fixed-size aggregate unit of parallel work whose size can be tailored to an architecture.

From a code generation perspective, fine-grain parallelism on a CPU exploits single instruction, multiple data (SIMD)

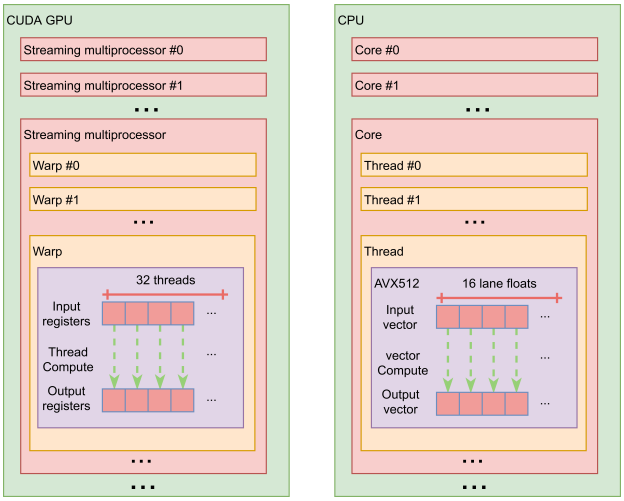


Fig. 3: SIMT and SIMD on GPUs and CPUs.

vectorization and on a GPU uses CUDA's single instruction, multiple thread model (SIMT) for both fine- and coarse-grain parallelism. Figure 3 shows the similarity between CPU SIMD and GPU SIMT. In terms of fine-grain parallelism, we noticed that a warp on a GPU is similar to a thread on CPU executing vector instructions. GPUs can issue instructions that operate on contiguous data and can exchange data across the vector lanes using the CUDA shuffle command. A warp or a thread is then the smallest parallel unit which provides vector compute capability for the stencil computation within a brick's domain. It will be assigned a brick at a time and execute the stencils using SIMT or SIMD until it finishes. In this parallelization level, a brick as an aggregate unit of data is able to optimize for data movement, TLB locality, and reuse in first-level cache. Moreover, brick as an aggregate unit of parallel work with fixed domain allows for efficient vector code generation.

Additional levels of coarse-grain parallelism are driven by the presence of shared caches and efficient synchronization. On a GPU, the second level of parallelism is a streaming multiprocessor that contains a collection of warps with a shared L1 cache. These warps are programmed using blocks that offer flexible mapping to hardware and are scheduled as a whole. On Intel Knights Landing, this level is a tile that consists of 2 cores (8 threads) and shares L2 cache. The shared cache also enables relatively more efficient OpenMP synchronization. To capitalize on the data reuse of nearby bricks, we may assign a rectangular subdomain to these units.

For computation using a single node, the third level of parallelism is the whole chip that is either a collection of streaming multiprocessors or cores. On GPUs, CUDA's grid and block decomposition allows mapping work to these hardware units. On CPUs, OpenMP's dynamic schedule offers similar flexibility.

IV. Brick Library Overview

In this paper, the brick data layout is implemented as a *C++* library. When writing stencil computations with bricks, the

```
void stencil(float *ptr_next, float *ptr_prev,
     float *ptr_vel, float *coeff) {
// ... Code to assume aligned ...
// ... Code convert to 3D arrays ...
omp_set_num_threads(32);
#pragma omp parallel for collapse(3) schedule(
     dynamic, 1) proc_bind(spread)
for (long rk = GC; rk < N + GC; rk += RK)
for (long rj = GC; rj < N + GC; rj += RJ)
for (long ri = GC; ri < N + GC; ri += RI){
omp_set_num_threads(8);
#pragma omp parallel for schedule(static, 1)
     proc_bind(close)
for (long tk = rk; tk < rk + RZ; tk += 4)
for (long tj = rj; tj < rj + RY; tj += 4)
for (long ti = ri; ti < ri + RX; ti += 16)
#pragma unroll_and_jam(4)
for (long k = tk; k < tk + 4; ++k)
#pragma unroll_and_jam(4)
for (long j = tj; j < tj + 4; ++j)
#pragma vector aligned nontemporal
for (long i = ti; i < ti + 16; ++i) {
  float c = prev[k][j][i] * coeff[0] + (
    prev[k][j][i+1] + prev[k][j][i-1] +
    prev[k][j+1][i] + prev[k][j-1][i] +
    prev[k+1][j][i] + prev[k-1][j][i]) *
    coeff[1];
  next[k][j][i] = c * vel[k][j][i];
}}}
```

(a) 7-point stencil baseline code expressed using arrays.

```
void stencil(brickd &next, brickd &prev, brickd &vel,
     brick_list &blist, float *coeff) {


omp_set_num_threads(blist.lv1);
#pragma omp parallel for schedule(dynamic, 1) proc_bind(
     spread)
for (long r = 0; r < blist.rlen; ++r) {


omp_set_num_threads(blist.lv2);
#pragma omp parallel for schedule(static, 1) proc_bind(
     close)
for (long l=blist.rdat[r]; l<blist.rdat[r+1]; ++l)

for (long o=blist.bdat[l]; o<blist.bdat[l+1];++o){
long b = blist.dat[o];
for (long k = 0; k < prev.info->dim_z; ++k)

for (long j = 0; j < prev.info->dim_y; ++j)

for (long i = 0; i < prev.info->dim_x; ++i) {
  float c = prev.elem(b,k,j,i)*coeff[0]+(
    prev.elem(b,k,j,i+1)+prev.elem(b,k,j,i-1)+
    prev.elem(b,k,j+1,i)+prev.elem(b,k,j-1,i)+
    prev.elem(b,k+1,j,i)+prev.elem(b,k-1,j,i))*
    coeff[1];
  next.elem(b,k,j,i)=c*vel.elem(b,k,j,i);
}}}
```

(b) 7-point stencil code expressed using bricks.

Fig. 4: A comparison of node-level 7-point stencil code, using 3D arrays vs. bricks. The tiled code (left) and brick code (right) reflect the baseline and brick code, respectively, for KNL experiments in this paper. Autotuning is used to optimize the tile sizes RK, RJ and RI for both versions of code. GPU code has the same structure, but differs in how thread-level parallelism is expressed, its use of low-level instructions, and required more autotuning exploration to achieve high performance.

grid value reference is extended with one additional dimension that represents the index of the brick. All references to a point `(k,j,i)` are replaced by the reference to a point in a brick `b`: `(b,k,j,i)`. The brick library allows an application developer to write stencil computations using a familiar *C-style* notation with 3+1D accesses that are automatically mapped to efficient code. For example, let us assume bricks are fixed-size $4\times4\times4$ subdomains which are 3-dimensional to target typical 3D stencil codes. Figure 4(b) shows how a user would express a 7-point stencil computation using bricks and is similar to the accompanying tiled implementation using 3D arrays in Figure 4(a). Figure 5 shows how to instantiate a brick and invoke the stencil. When accessing an element in another brick (e.g. `vel.elem(b,0,0,4)` is element `(0,0,0)` in the brick following it logically in the $x-$dimension), the library internals will translate the access into accessing the corresponding element in the brick following `b` in the `i` dimension.

This adjacency list organization may also realize non-rectangular domains, and enables the code generator to explore different layouts, affinities, and scheduling schemes for non-uniform memory access (NUMA) and distributed memory systems. Bricks from multiple grids can also be interleaved in a manner that is transparent to the computation to further reduce the number of discontinuous memory accesses in a stencil computation.

We employ a source-to-source compiler to optimize the stencil computation [29] within each brick to optimize brick

```
// brick_info(Brick Dimensions)
brick_info binfo(4, 4, 4);
// genList(Domain Size in Bricks, Outer Parallel
     Subdomain, Inner Parallel Subdomain):
// for 512^3 and 4 wide ghostzone on each side
brick_list blist = binfo.genList(128+2,130,130, RK, RJ,
     RI, 4, RJ, RI);
// Allocating
brickd prev(&binfo),next(&binfo),vel(&binfo);
// ... Initialization Code ...
// ---- Run for iter * 2 time steps ----
for (long t = 0; t < iter; ++t) {
  stencil(next, prev, vel, blist, &coeff[0]);
  stencil(prev, next, vel, blist, &coeff[0]);
}
```

Fig. 5: Allocating bricks and invoking the stencil.

address translation and generate efficient vector code for different architectures. The brick library thus lends itself to autotuning of the execution schedule to adapt to different architectures, types of stencils, and application contexts. As such, the brick library, provides performance portability through a single source representation, architecture-specific code generation, and autotuning.

V. Experiments

To demonstrate that bricks provide performance portability, we evaluate a number of performance metrics for bricks vs. tiling across both CPU- and GPU-based systems. Table I describes a collection of 3D stencils explored in the experiment. Six synthetic stencils for the Laplacian operator of

varying order are used to capture the memory-compute ratio of different kinds of stencil shapes and diameters. Two real-world stencils have been taken from application code. The six synthetic stencils are named according to the number of points. The 7-point, 13-point, 19-point, and 25-point belong to stencils for the Laplacian operator, that only operate on elements along each of the axes (star-shaped) in each dimension. For these stencils, the stencil diameter is equivalent to the order as there are no off-axis points in the stencil (e.g., the 7-point stencil is second order and also has a diameter of 3). The 27-point and 125-point are stencils for the "compact" Laplacian that touch all points in a cube (dense), with Manhattan distance equal to the radius. The stencil diameter (twice the radius plus 1) is again the same as the order; in some applications these stencils can produce more accurate solutions. The `iso` stencil resembles the 25-point stencil, with diameter 9. We use the `hypterm` routine from CNS which is a variable-coefficient Poisson stencil [30] in finite volume form that is 8^{th}-order, and has a similar footprint as the 25-point stencil.

Stencil	Description	Data Movement per point (Read:Write)	FLOPs per point
7pt	2^{nd} order	1:1 words	13
13pt	4^{th} order	1:1 words	25
19pt	6^{th} order	1:1 words	37
25pt	8^{th} order	1:1 words	49
27pt	2^{nd} order	1:1 words	53
125pt	4^{th} order	1:1 words	249
iso	8^{th} order, isotropic finite difference [31]	3:1 words	61
CNS	8^{th} order, compressible Navier Stokes [30]	8:5 words	466

TABLE I: Stencils used in our experiments. Data movement is a lower bound assuming only compulsory cache misses and cache bypass for write (nontemporal stores for KNL) while FLOPs counts only those in the source code.

Compulsory data movement in Table I is a lower bound that assumes ideal cache behavior and blocking. All the synthetic stencils must move two elements per stencil application (read the full input array; write the full output array). FLOP/Point refers to how many floating-point operations each application of the stencil requires. All synthetic stencils have a unique weight for each of point in the stencil (but spatially constant). Thus, a 27-point stencil has 27 weights, 27 multiplies, and 26 adds. One can calculate the theoretical arithmetic intensity for these kernels by dividing the number of FLOPs per point with the total number of bytes of data moved per point and observe all but the 125-point will be ultimately bound by memory bandwidth on KNL.

A. Intel Xeon Phi Knights Landing

We generate AVX-512 code for the Intel Xeon Phi 7250 Knights Landing (KNL) CPU. The processor has 68 physical cores organized into a 2D on-chip mesh of 34 tiles each with two CPU cores[1] and a shared 1MB L2 cache. Each core has a private 32KB L1 data cache, implements 4-way multithreading, and has two AVX-512 vector processing units (VPUs). Although each core has a nominal frequency of 1.4GHz, under AVX-heavy computations, the cores will downclock to 1.2GHz. With 64 cores, the theoretical peak performance is 4915 GFLOP/s for single-precision fused multiply-and-add (half that for double, half that again for add or multiply instructions). Each chip has both standard DDR4 DRAM memory and high-bandwidth MCDRAM memory. MCDRAM can be configured as either a last level cache, or as a separate addressable memory (exposed to programmers as a second NUMA node). We preserve our target machine's (NERSC's Cori) nominal configuration of the MCDRAM as a last level cache (*quadcache*) to reflect typical user models. This yields a STREAM bandwidth of about 332 GB/s.

For the brick implementation, we used $4 \times 4 \times 16$ bricks for all stencils with single precision and $4 \times 4 \times 8$ bricks for double precision.

We compare the stencil computation using bricks against three spatial tiling schemes — 2D tiling, 3D tiling from Rivera and Tseng [28] and 6D tiling that resembles bricks without the layout transformation. The concepts are shown in Figure 1. Note that the 6D version resembles the brick code without a data layout transformation. For all tiling implementations and brick variants, we use exhaustive search on the CPUs to find the best tiling factor. The performance is based on the average throughput of the stencil kernel when running consecutively for 2 seconds on a 512^3 domain in *GStencil/s*. All the stencils are compiled with the pragma for nontemporal stores. All tiling code are successfully vectorized by the Intel compiler with AVX-512.

Across different tiling implementations, the results are as expected. Generally, 3D tiling often produces the best performance except on the 27-point and 125-point stencils in double precision and CNS in both precision due to better cache reuse compared with the 2D version and reduced TLB pressure compared with the 6D tiling scheme. Except for CNS, the best tuned 3D tiling variant did not tile in the unit-stride dimension (*TILEI=512*) and as such, is equivalent to 2D tiling with nested parallelism). The 6D version produces the best performance on CNS due to significantly improved reuse with larger stencil diameter and cache pressure from the larger number of simultaneous grid accesses.

We first analyze performance by noting the significant reduction in data movement using bricks as compared against different tiling implementations. For this purpose, we use the Intel VTune Amplifier to collect the Read and Write MCDRAM data movement using the frame API and the count of hardware cache events. Assuming the write data movement is fixed for each of the computations, the empirical read:write ratio shows how many more reads from MCDRAM are incurred from non-ideal caching (superfluous capacity misses). The result is shown in Figure 6. We see almost invariably that the brick variants require less data movement than any tiled implementation.

Next, we examine how bricks significantly lower TLB

[1]We use 32 tiles for a total of 64 cores in all our experiments to isolate system overhead.

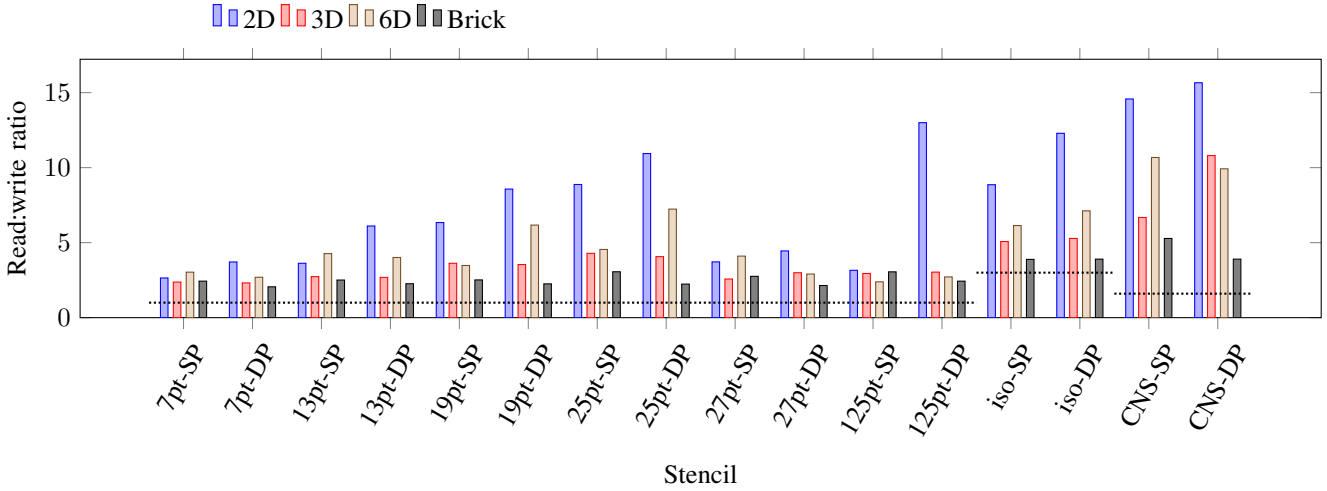


Fig. 6: Intel KNL MCDRAM Read:write ratio with different tiling schemes. Capacity misses induce superfluous reads and inflate the read:write ratio (lower is better). Dotted lines represent the ratio for the compulsory (ideal) case with nontemporal stores. Bricks generally offer much better cache locality than any 2D, 3D, or 6D tiled implementation.

Stencil	Overhead	Blocking (Best)	Bricks
125pt	uTLB	2.65%	1.27%
	Page-walk	6.02%	4.16%
CNS	uTLB	4.54%	0.71%
	Page-walk	16.75%	3.80%

TABLE II: TLB pressure (measured in percentage of clock cycles) for single-precision stencils on KNL. Observe that bricks dramatically reduces the time incurred from uTLB misses and page walks — an affect that improves bandwidth without changing arithmetic intensity.

pressure. Once again, using the Intel VTune Amplifier, we calculated the fraction of the clock cycles spent handling TLB misses in Table II for two of the higher-order stencils. The uTLB overhead captures the first-level data TLB misses and the page-walk overhead captures the second-level data TLB misses. We observe that bricks result in a huge reduction in both uTLB and Page Walk overhead.

Finally, we show that bricks can perform at a high fraction of machine performance. Figure 7 presents an empirical Roofline figure for our attained brick performance. As we use uniquely weighted synthetic stencils there is no redundant computation. The FLOP/s rates are then calculated using theoretical FLOPs per point from Table I and stencil throughput. FLOP/s may be overestimated for iso and CNS. Arithmetic intensity is the ratio of floating point performance to the achieved bandwidth collected using Intel VTune Amplifier. For many computations our brick library attains performance near the machine's capabilities. However, attaining peak performance for the most compute-intensive stencils is increasingly challenging as non-floating-point vector instructions e.g. `valign` consume vector unit (VPU) cycles and displace useful floating-point computations.

Table III shows that on KNL the brick code is almost always faster and achieves up to $4.4\times$ the performance compared

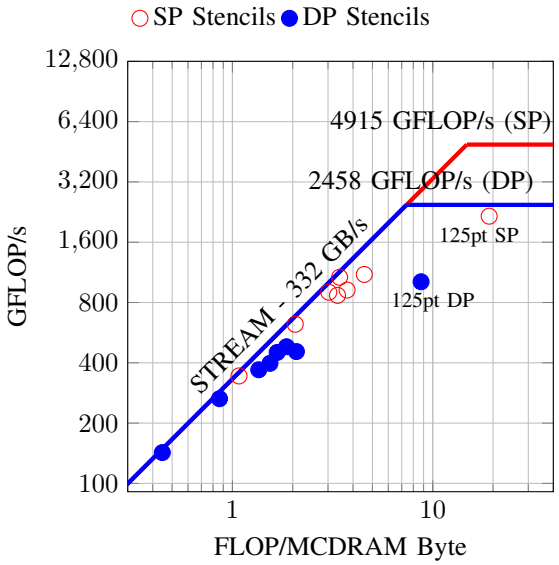


Fig. 7: Roofline figure for the KNL 7250 where each dot represents our optimized stencil performance. Observe that most stencils are near the Roofline.

with the tiling version. Bricks are only slower for lower order memory-bound stencils that don't offer sufficient cache reuse to make bricks profitable (7pt-SP, 7pt-DP, and 13pt-DP).

B. Intel Xeon Skylake Gold

With the AVX-512 code generation capability, we can also run on traditional CPU architectures. To that end, we evaluated bricks on a 2.1GHz Intel Xeon Gold 6130 which is based on the Skylake core architecture. This Skylake has 16 cores with 32 threads and each core is equipped with two AVX-512 units providing a theoretical (assuming no AVX downclocking) single-precision peak performance of

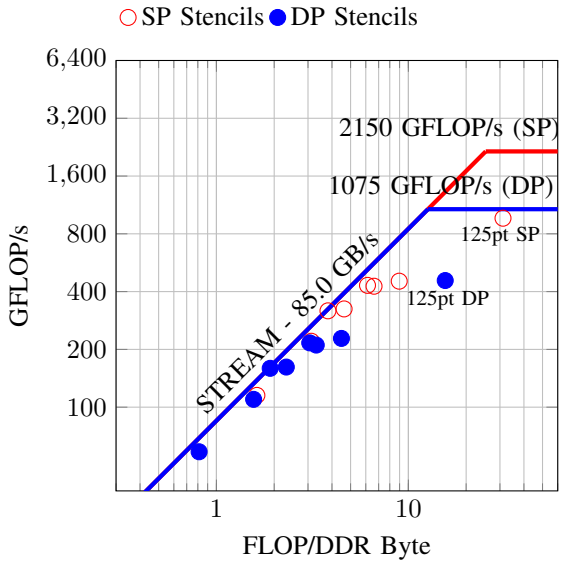


Fig. 8: Roofline figure for the Skylake Gold CPU where each dot represents our optimized brick performance. Note, the FLOP ceiling is theoretical and overly optimistic in that it does not account for an lower AVX clock rate.

2150 GFLOP/s and a double-precision peak performance of 1075 GFLOP/s. Concurrently, 6 memory controllers provide a STREAM bandwidth of 85GB/s. We used $4 \times 4 \times 16$ bricks for all stencils with single precision and $4 \times 4 \times 8$ bricks for double precision.

The target machine prevents Intel VTune Amplifier from accessing the performance counters required to accurately measure DRAM data movement. Nevertheless, we may estimate DRAM data movement (and thus arithmetic intensity) using compulsory cache misses of Table I. Figure 8 shows our brick library can deliver performance close to the Roofline for both single- and double-precision implementations. Although compulsory data movement is an underestimate for total data movement (upperbound on arithmetic intensity), the resultant Roofline plot proves it is very close to total data movement (points can never be left of the bandwidth ceiling).

As with KNL, we expect non-floating-point vector instructions to have limited the available performance and thus limited the performance of the most compute-intensive stencil (125-point). Moreover, in this paper, we have assumed SKL runs AVX-512 code at the nominal 2.1GHz. However, if like KNL it underclocks when running AVX-512 heavy code, then we have overestimated SKL peak performance and our brick performance is much closer to the true peak.

As with KNL, we also compared our brick-based stencil performance against the three spatial tiling schemes. Table III shows that the brick code can deliver up to $5.0\times$ the performance compared with the tiled version and is only (slightly) slower on the simplest stencils.

C. NVIDIA P100 Pascal GPU

To deliver performance portability across a wide range of HPC platforms, the brick library can generate CUDA code for the NVIDIA Tesla P100 GPU. The P100 GPU has 56 streaming multiprocessors. Each streaming multiprocessor has 64 single-precision and 32 double-precision CUDA cores and has a warp size of 32. The P100 has a theoretical peak single-precision performance of 9.3 TFLOP/s, a peak double-precision performance of 4.7 TFLOP/s, and a GPU-STREAM [32] bandwidth of 586 GB/s.

For GPUs, we compared the stencil computation using bricks against two tiling schemes: 3D and 6D. 6D tiling is representative of the parallel schedule used for the brick code. We tune the tiling implementations exhaustively and report the tiling factor and the number of warps in 6D. For the brick code, we also experimented with different execution order within the thread group domain. However, the schedule shown for 6D often yielded the best performance. With the expanded tuning space, we tune the brick code using random forest and search for at most 6 hours for each stencil or stops early when the global best value didn't change for 1000 iterations. We use $4\times4\times32$ bricks for all stencils except `CNS` which uses $4\times4\times8$ bricks for single-precision and $4 \times 4 \times 4$ bricks for double-precision to reduce cache pressure. The stencil performance is based on the average stencil throughput over 100 timesteps on a 512^3 domain (384^3 for `CNS`) in *GStencil/s*.

Comparing between the two tiling schemes, we find that the 6D version actually outperforms the 3D version on all stencils except the 13-point and 19-point stencil with double precision. Nevertheless, as shown in Table III, our brick library outperforms the highly-optimized 6D tiling GPU baseline by up to $5\times$ in double precision.

On the GPU, bricks offer comparable L2 cache reuse and the brick code generator yields significantly better register reuse. Figure 9(left) shows the read:write ratio computed using NVProf. The brick version does not attain the same L2 locality as the tiled version on the 27-point and 125-point stencils. This is inconsequential as the tiling implementation of these stencils is bound by the L1 cache on the GPU. When it comes to L1 pressure, we show in Figure 9(right) that the brick code generator reduces the L1 pressure by up to $11\times$ by attaining much better locality in the register file.

Our brick implementation attains a high fraction of the GPU performance. Figure 10 shows the Roofline model plotted from empirical FLOPs and bandwidth collected using NVProf. Our brick code attains a high fraction of the machine bandwidth and operates close to the HBM Roofline while relatively low occupancy (at 12.5% for iso-SP and 125pt-DP). Although higher occupancy is generally preferable for latency-bound kernels, for well-optimized codes, higher occupancy may stress certain functional units unnecessarily and increase the latency [33].

Table III shows that our brick code CUDA generator achieves a speedup between $1.1\times$ and $5.0\times$ compared with the higly-optimized tiled version.

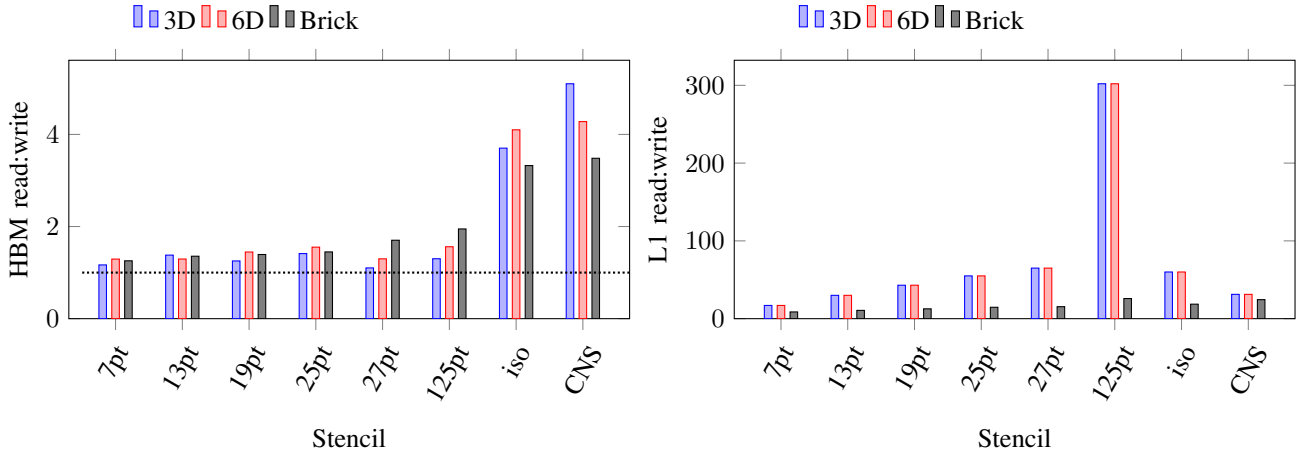


Fig. 9: Read:write ratio on P100 with single precision. Double precision yields similar comparison. HBM data movement ratio captures reuse in the L2 cache, while L1 data movement ratio captures reuse in registers. All of the tiled stencils are bound by their L1 usage. Note that the code generator for bricks offers much better register reuse.

	Double-Precision			Single-Precision		
Stencil	**KNL**	**SKL**	**P100**	**KNL**	**SKL**	**P100**
7pt	10.96 (0.7×)	4.51 (0.9×)	24.25 (1.1×)	26.46 (0.9×)	8.86 (0.9×)	41.04 (1.4×)
13pt	10.59 (0.8×)	4.39 (0.9×)	21.06 (1.4×)	24.93 (1.0×)	8.82 (0.9×)	35.79 (1.8×)
19pt	9.98 (0.9×)	4.37 (0.9×)	18.84 (1.7×)	24.35 (1.2×)	8.78 (0.9×)	32.18 (2.1×)
25pt	9.20 (1.2×)	4.41 (1.1×)	16.94 (1.8×)	21.83 (1.3×)	8.80 (1.1×)	29.08 (2.3×)
27pt	8.59 (0.9×)	3.97 (1.2×)	19.29 (2.1×)	20.84 (1.0×)	8.05 (1.2×)	26.50 (2.0×)
125pt	4.08 (4.4×)	1.83 (4.9×)	11.44 (5.0×)	8.67 (4.1×)	3.87 (5.0×)	16.78 (4.1×)
iso	6.52 (1.0×)	2.61 (1.1×)	10.56 (1.4×)	14.24 (1.1×)	5.21 (1.0×)	19.93 (2.0×)
CNS	1.03 (2.9×)	0.49 (2.0×)	1.87 (1.9×)	1.98 (2.3×)	0.97 (2.1×)	3.18 (2.3×)

TABLE III: Attained autotuned performance with bricks in GStencil/s and speedup (in parenthesis) over the best tuned tiled implementation. Observe, as stencil complexity increases, so too does the benefit of bricks.

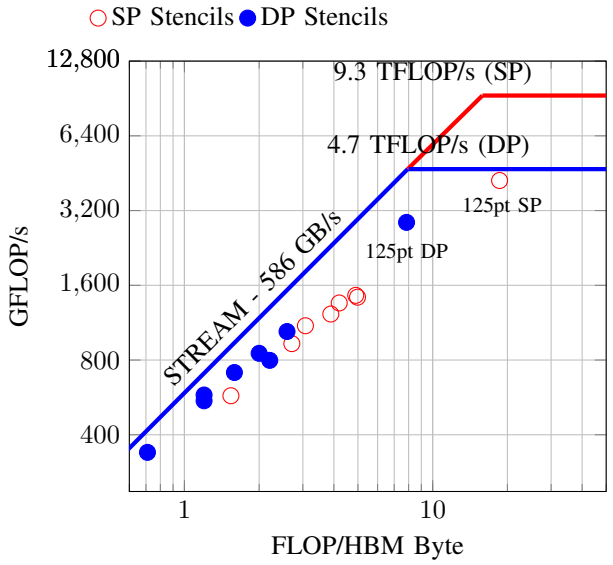


Fig. 10: Roofline figure for benchmarks running on the P100 Pascal GPU where each dot represents our optimized brick performance for a different stencil.

D. Performance Portability

To assess the performance portability of bricks, we adopt the performance portability metric $\mathbf{\Phi}$ across a set of platforms $\boldsymbol{H}$ as defined by Pennycook et al. [34], [35] as reproduced in Equation 1. In that formalism, we define the metric's efficiency $e_i(a,p)$ for application a and problem p on platform i as the *fraction-of-roofline*.

$$\mathbf{\Phi}(a,p,\boldsymbol{H}) = \begin{cases} \dfrac{|\boldsymbol{H}|}{\sum_{i\in\boldsymbol{H}} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported, } \forall i \in \boldsymbol{H} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Table IV presents the resultant efficiencies (performance relative to Roofline) for each platform, stencil, and precision. Additionally, we show performance portability using Equation 1 for a given stencil and precsision, and again averaged over all stencils for a given precision. We observe that the smaller stencils are generally memory-bound and attain high fractions of the Roofline for most platforms. Consistently high efficiencies produce high performance portability $\mathbf{\Phi}$. Conversely, the consistently moderate efficiency for the 125-point stencil produces a moderate $\mathbf{\Phi}$. Overall, across all platforms and stencils, we attain a performance portability $\mathbf{\Phi}$ of 72% in double precision.

	Single-Precision				Double-Precision			
Stencil	**KNL**	**SKL**	**P100**	**$\mathbf{\Phi}$**	**KNL**	**SKL**	**P100**	**$\mathbf{\Phi}$**
7pt	96%	83%	64%	**79%**	96%	85%	82%	**87%**
13pt	91%	83%	59%	**75%**	92%	83%	78%	**84%**
19pt	90%	83%	54%	**72%**	82%	82%	77%	**80%**
25pt	94%	83%	51%	**71%**	81%	83%	73%	**79%**
27pt	73%	76%	49%	**63%**	66%	75%	69%	**70%**
125pt	44%	45%	45%	**45%**	41%	43%	62%	**47%**
iso	78%	98%	61%	**76%**	78%	98%	82%	**85%**
cns	74%	60%	55%	**62%**	78%	60%	62%	**66%**
				66%				**72%**

TABLE IV: Application Efficiency (e_i) and Performance Portability of various stencils when using bricks. Numbers below the table represent averages across all stencils.

Currently, we calculate efficiency e_i based solely on the number of useful floating-point operations. However, brick-based stencils can require vector shuffle and alignment operations that consume cycles that would otherwise be used for floating-point operations. As a result, although our efficiency calculations are accurate in that they include all floating-point operations, they may not be sufficient as they do not incorporate all vector operations. To that end, in the future, we will expand our efficiency metric to incorporate all vector operations in order to better account for contended resources as suggested by Yang et al. [36]. This will more accurately calculate e_i and thus increase our brick library's $\mathbf{\Phi}$.

VI. Related Work

A large body of prior work on optimizing stencils has focused on stencils applied on large grids which are usually bound by capacity or compulsory cache misses, leading to a variety of studies on spatial and temporal tiling [37], [28], [38], [39], [40], [41], [5], [10], [8], [4], [1], [2], [3], [42], [9], [7], [12], [6]. A few specifically focused on generating GPU code [43], [44]. These tiling techniques have focused on loop or iteration space tiling and sometimes seek to optimize in the time domain ("2.5D") to increase data reuse. The applicability of 2.5D tiling is highly dependent on the structure and complexity of the underlying numerical method and presence of distributed memory communication. It has been shown to be effective for the simplest operators running on a single node, and when profitable is straightforward to express with a 2.5D loop structure built on top of our generated brick iteration.

In addition to loop tiling, researchers have also tiled or blocked data. Data along with loop tiling efforts have been addressed by [24], [45], [26], [25]. TiDA [45] uses coarse-grained data blocking, where the entire grid is tiled into sub-grids, each sub-grid with its own ghost zone. Fine-grained data blocking is explored in Bricks and Briquettes [24], YASK [25] and RTM on the Cell processor [26]. All the fine-grained blocking techniques targeted large, compute-intensive stencils, and the small data blocks do not have per-block ghost zones.

This paper introduces a new approach to data blocking called bricks, which is encapsulated in a library and supports vector code generation (for CPUs and GPUs), and hierarchical node-level parallelism. The bricks used in our research are similar to briquettes in [24], but there is significant difference in our approaches. Briquettes were designed to perform 3D stencils split into 1D stencils, thus requiring multiple sweeps to compute the output. Furthermore, a data transpose was required between each 1D stencil sweep to ensure good SIMDization. Their code generation required data staging tailored for 1D stencils. In contrast to Briquettes, we optimize 3D stencils without dimensional splitting in addition to fine-grained data blocking to improve computation by reducing reads cache or DRAM and improving SIMDization.

Considering other fine-grained data blocking, YASK is a C++ template-based approach to generating code for large stencils with fine-grained data blocks. YASK autotuned their data block size, and, used vector-length data blocks which are smaller than our method, such as $2\times2\times4$ with AVX-512 instead of $4\times4\times16$. YASK targets x86 based architectures and thus lacks portability. Our approach addresses this challenge by generating code for both CPUs and GPUs. RTM was optimized on the Cell processor in [26] using fine-grained data blocking. The code was manually optimized, and focused on a single stencil.

A common approach to deriving high-performance stencil computations is to use a domain-specific compiler that automatically generates architecture-specific code from a stylized stencil specification [21], [17], [18], [19], [20], [22], [23]. Among these, only MODESTO, PATUS, ExaStencil, and Halide can generate both CPU and GPU code. MODESTO [20] is focused on scheduling the computation and data movement of multiple stencil computations. PATUS [21] uses both a stencil description and a machine mapping description to generate architecture-specific code. The ExaStencil project [22], [46], [47] uses layered DSLs to map from one high-level stencil description to different target architectures. Halide separates computation specification from architecture-specific schedule, which can be automatically-generated or written by a programmer, and applies a limited set of optimizations [23]. While bricks could be the target of a DSL and use the brick code generator, our work is distinguished from all of these systems in its use of the brick data layout to tackle the memory system and parallelism for both CPU and GPU architectures.

Perhaps the closest work is the QCD Grid library [48], [49] wherein the 4-lattice used in QCD is folded into small arrays of virtual nodes stored contiguously in memory. Although Grid provides a QCD-specific code generator, it lacks the compiler infrastructure required for broad applicability and performance portability.

Although there are many stencil benchmarks, very few include 3D stencils, and in general they are limited to simple examples and lack comparison to Roofline models [50], [51], [52], [53], [54].

In summary, the stencil optimization approach based on the brick data layout in this paper offers a new abstraction for memory hierarchy optimization, vectorization, thread parallelism and is the centerpiece for achieving performance

portability of stencil computations across CPUs and GPUs.

VII. Conclusions and Future Work

In order to attain performance portability across different modern architectures including SIMD CPUs and SIMT GPUs, this paper introduced a brick data layout. Bricks provide an abstraction for code generation and optimization that allow an implementation to be adapted to different stencil computations and different target architectures.

Ultimately, we show we can consistently attain high performance on compute-intensive stencils and high-bandwidth on memory-intensive stencils close to the Roofline bound of the respective architecture. Additionally, we demonstrate that our brick library delivers much better performance as stencil complexity grows — a key imperative as applied mathemeticans reorganize computation to avoid the memory wall. Moreover, we demonstrate and quantify performance portaiblity across both the AVX-512 based Knights Landing and Skylake CPUs as well as the P100 GPU.

Although other stencil optimizations such as temporal blocking and wavefront parallelism are beyond the scope of this paper, they are complementary and we will explore combining them with bricks at the source code level where applicable.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration. This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Sciences Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231.

References

[1] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[2] D. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *Proc. Interational Conference on Parallel and Distributed Computing Systems*, 2000.

[3] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," Department of Computer Science, Rutgers University, Tech. Rep. DCS-TR-379, 1999.

[4] M. Frigo and V. Strumpen, "Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations," in *Proc. ACM International Conference on Supercomputing (ICS)*, 2005.

[5] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the Cell processor for scientific computing," in *Proc. Conference on Computing Frontiers*, 2006.

[6] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2007.

[7] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager, "Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method," *Progress in Computational Fluid Dynamics*, vol. 8, 2008.

[8] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009.

[9] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *International Computer Software and Applications Conference*, 2009.

[10] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[11] P. Ghysels, P. Kosiewicz, and W. Vanroose, "Improving the arithmetic intensity of multigrid with the help of polynomial smoothers," *Numerical Linear Algebra with Applications*, vol. 19, no. 2, pp. 253–267, 2012.

[12] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2012.

[13] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[14] P. Basu, M. Hall, S. Williams, B. Van Straalen, L. Oliker, and P. Colella, "Compiler-directed transformation for higher-order stencils," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 313–323.

[15] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Eliminating redundancies in sum-of-product array computations," in *Proceedings of the 15th international conference on Supercomputing*. ACM, 2001, pp. 65–77.

[16] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 65–76.

[17] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *ACM symposium on Parallelism in algorithms and architectures*, 2011.

[18] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *International Symposium on Code Generation and Optimization (CGO)*, 2012.

[19] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox, "Snowflake: A lightweight portable stencil dsl," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 795–804.

[20] T. Gysi, T. Grosser, and T. Hoefler, "Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751223

[21] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 676–687. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.70

[22] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter *et al.*, "Exastencils: advanced stencil-code engineering," in *European Conference On Parallel Processing*. Springer, 2014, pp. 553–564.

[23] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 519–530. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462176

[24] J. Jayaraj, "A strategy for high performance in computational fluid dynamics," Ph.D. dissertation, University of Minnesota, 2013.

[25] C. Yount, J. Tobin, A. Breuer, and A. Duran, "Yask-yet another stencil kernel: A framework for hpc stencil code-generation and tuning," in *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC*, ser. WOLFHPC '16, 2016.

[26] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, "3d seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors," *Sci. Program.*, vol. 17, no. 1-2, pp. 185–198, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/382638

[27] T. Karnagel, T. Ben-Nun, M. Werner, D. Habich, and W. Lehner, "Big data causing big (tlb) problems: Taming random memory accesses on the gpu," in *Proceedings of the 13th International Workshop on Data Management on New Hardware*, ser. DAMON '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/3076113.3076115

[28] G. Rivera and C. Tseng, "Tiling optimizations for 3D scientific computations," in *Supercomputing (SC)*, 2000.

[29] T. Zhao, M. Hall, P. Basu, S. Williams, and H. Johansen, "Simd code generation for stencils on brick decompositions," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 423–424. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178537

[30] M. Emmett, W. Zhang, and J. B. Bell, "High-order algorithms for compressible reacting flow with complex chemistry," *Combustion Theory and Modelling*, vol. 18, no. 3, pp. 361–387, 2014.

[31] C. Andreolli, P. Thierry, L. Borges, G. Skinner, and C. Yount, "Characterization and optimization methodology applied to stencil computations," in *High Performance Parallelism Pearls*, 1st ed., J. Jeffers and J. Reinders, Eds. Morgan Kaufmann, 2015, ch. 23, pp. 377–396.

[32] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Gpu-stream v2. 0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *International Conference on High Performance Computing*. Springer, 2016, pp. 489–507.

[33] V. Volkov, "Understanding latency hiding on gpus," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[34] S. J. Pennycook, J. D. Sewall, and V. Lee, "A metric for performance portability," *arXiv:1611.07409*, 2016.

[35] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, 2017.

[36] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. Deslippe, and S. Williams, "An empirical roofline methodology for quantitatively assessing performance portability," in *International Workshop on Performance, Portability and Productivity in HPC P3HPC*, 2018.

[37] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, 2004.

[38] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Supercomputing (SC)*, 2008.

[39] S. Williams, L. Oliker, J. Carter, and J. Shalf, "Extracting ultra-scale lattice Boltzmann performance via hierarchical and distributed autotuning," in *Supercomputing (SC)*, 2011.

[40] M. Kowarschik and C. Wei, "Dimepack - a cache-optimized multigrid library," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), volume I*, 2001.

[41] C. C. Douglas, J. Hu, M. Kowarschik, U. Rde, and C. Weiss, "Cache optimization for structured and unstructured grid multigrid," *Elect. Trans. Numer. Anal*, vol. 10, pp. 21–40, 2000.

[42] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2, 2009.

[43] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304619

[44] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 66:66–66:75. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544160

[45] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, *TiDA: High-Level Programming Abstractions for Data Locality Management*. Cham: Springer International Publishing, 2016, pp. 116–135.

[46] S. Kronawitter and C. Lengauer, "Optimizations applied by the exastencils code generator," 2015.

[47] S. Kuckuk, G. Haase, D. A. Vasco, and H. Köstler, "Towards generating efficient flow solvers with the exastencils approach," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 17, 2017.

[48] P. Boyle, "Data parallel c++ mathematical object library." [Online]. Available: https://github.com/paboyle/Grid

[49] R. Li, C. Detar, D. W. Doerfler, S. Gottlieb, A. Jha, D. Kalamkar, and D. Toussaint, "Milc staggered conjugate gradient performance on intel knl," *Proceedings of Science (POS)*, 2016.

[50] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," *SIGPLAN Not.*, vol. 51, no. 8, pp. 5:1–5:12, Feb. 2016. [Online]. Available: http://doi.acm.org/10.1145/3016078.2851157

[51] A. Schafer and D. Fey, "High performance stencil code algorithms for gpgpus," *Procedia Computer Science*, vol. 4, pp. 2027 – 2036, 2011, proceedings of the International Conference on Computational Science, ICCS 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050911002791

[52] A. D. Pereira, L. Ramos, and L. F. W. Goes, "Pskel: A stencil programming framework for cpugpu systems," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4938–4953.

[53] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, "Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 259–268.

[54] N. Maruyama and T. Aoki, "Optimizing stencil computations for nvidia kepler GPUs," 2014. [Online]. Available: http://www.exastencils.org/histencils/2014/histencils2014.pdf

APPENDIX
TILING IMPLEMENTATIONS

A. Kernel example

A 7-point stencil kernel of tiled code is implemented as follows.

```
jstride = (N + 2*SH);  kstride = (N + 2*SH)*(N + 2*SH);
out[k*kstride + j*jstride + i] = coeff[0]*in[k*kstride +
     j*jstride + i] +
  coeff[0]*in[k*kstride + j*jstride + (i+1)]+coeff[0]*in
     [k*kstride + j*jstride + (i-1)] +
  coeff[0]*in[k*kstride + (j+1)*jstride + i]+coeff[0]*in
     [k*kstride + (j-1)*jstride + i] +
  coeff[0]*in[(k+1)*kstride + j*jstride + i]+coeff[0]*in
     [(k-1)*kstride + j*jstride + i];
```

Following are tiling implementations for different architectures. SH is the width of the ghost zone and is set to be the length of a cacheline. REG* and TILE* are tuning parameters.

B. KNL & Xeon implementations

2D

```
#pragma omp parallel for collapse(2) schedule(dynamic,
     1)
for (long tk = SH; tk < N + SH; tk += TILEK)
for (long tj = SH; tj < N + SH; tj += TILEJ)
  for (long k = tk; k < tk + TILEK; ++k)
    for (long j = tj; j < tj + TILEJ; ++j)
      #pragma vector nontemporal
      #pragma omp simd
      for (long i = SH; i < N + SH; ++i)
        // Kernel
```

3D

```
#pragma omp parallel for schedule(dynamic, 1) collapse
     (3) proc_bind(spread)
for (long tk = SH; tk < N + SH; tk += TILEK)
for (long tj = SH; tj < N + SH; tj += TILEJ)
for (long ti = SH; ti < N + SH; ti += TILEI) {
  omp_set_num_threads(8);
  #pragma omp parallel for schedule(static, 1) proc_bind
     (close)
  for (long k = tk; k < tk + TILEK; ++k)
    for (long j = tj; j < tj + TILEJ; ++j)
    #pragma vector nontemporal
    #pragma omp simd
    for (long i = ti; i < ti + TILEI; ++i)
      // Kernel
}
```

6D

```
#pragma omp parallel for collapse(3) schedule(dynamic,
     1) proc_bind(spread)
for (long rk = SH; rk < N + SH; rk += REG)
for (long rj = SH; rj < N + SH; rj += REG)
for (long ri = SH; ri < N + SH; ri += REGI) {
  omp_set_num_threads(8);
  #pragma omp parallel for collapse(2) schedule(static,
     1) proc_bind(close)
  for (long tk = rk; tk < rk + REG; tk += TILE)
  for (long tj = rj; tj < rj + REG; tj += TILE)
    for (long ti = ri; ti < ri + REGI; ti += TILEI)
    for (long k = tk; k < tk + TILE; ++k)
    for (long j = tj; j < tj + TILE; ++j)
      #pragma vector nontemporal
      #pragma omp simd
      for (long i = ti; i < ti + TILEI; ++i)
        // Kernel
}
```

C. GPU implementations

3D

```
__global__ void
kernel(...) {
  long tk = blockIdx.z * REG + SH;
  long j = blockIdx.y * TILEJ + SH + threadIdx.y;
  long i = blockIdx.x * TILEI + SH + threadIdx.x;
  for (long k = tk; k < tk + TILEK; ++k)
    // Kernel
}
void call_kernel(...) {
  dim3 block(N/TILEI, N/TILEJ, N/TILEK);
  dim3 thread(TILEI,TILEJ);
  kernel<<<block,thread>>>(...);
}
```

6D

```
__global__ void
__launch_bounds__(32*TWARP, NBLOCK)
kernel(...) {
  long rk = blockIdx.z * REG + SH;
  long rj = blockIdx.y * REG + SH;
  long ri = blockIdx.x * REGI + SH;
  long line = REG / TILE;
  long iline = REGI;
  long tot = line * line * xline;
  if (threadIdx.y < NWARP)
  for (long n=threadIdx.y*TILEI;
       n<tot; n+=NWARP*TILEI) {
    long ti = ri + n % REGI;
    long r = n / REGI;
    long tj = rj + r % line * TILE;
    long tk = rk + r / line * TILE;
    for (long i = ti + threadIdx.x;
         i < i + TILEI; i += 32)
    for (long k = tk; k < tk + TILE; ++k)
    for (long j = tj; j < tj + TILE; ++j)
      // Kernel
  }
}
void call_kernel(...) {
  dim3 block(N/REGI,N/REG, N/REG);
  dim3 thread(32, TWARP);
  kernel<<<block,thread>>>(...);
}
```