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× 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., multistencils). 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 z-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,
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 prefetcers 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-
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 $d^2$. 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) 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
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.

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

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

Fig. 5: Allocating bricks and invoking the stencil.
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 hypertext routine from CNS which is a variable-coefficient Poisson stencil [30] in finite volume form that is 8th-order, and has a similar footprint as the 25-point stencil.

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

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 thatassumes 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 core[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 GFLOPs per 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 × 4 × 16 bricks for all stencils with single precision and 4 × 4 × 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 512x domain in GStencils. 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 [5]. We see almost invariably that the brick variants require less data movement than any tiled implementation.

Next, we examine how bricks significantly lower TLB...
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 non-temporal stores. Bricks generally offer much better cache locality than any 2D, 3D, or 6D tiled implementation.

<table>
<thead>
<tr>
<th>Stencil</th>
<th>Overhead</th>
<th>Blocking (Best)</th>
<th>Bricks</th>
</tr>
</thead>
<tbody>
<tr>
<td>125pt</td>
<td>uTLB</td>
<td>2.65%</td>
<td>1.27%</td>
</tr>
<tr>
<td></td>
<td>Page-walk</td>
<td>6.02%</td>
<td>4.16%</td>
</tr>
<tr>
<td>CNS</td>
<td>uTLB</td>
<td>4.54%</td>
<td>0.71%</td>
</tr>
<tr>
<td></td>
<td>Page-walk</td>
<td>16.75%</td>
<td>3.80%</td>
</tr>
</tbody>
</table>

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.

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 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
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 III. 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 $4 \times 4 \times 32$ bricks for all stencils except CNS which uses $4 \times 4 \times 8$ 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 GStencils/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.

<table>
<thead>
<tr>
<th>Stencil</th>
<th>KNL</th>
<th>SKL</th>
<th>P100</th>
</tr>
</thead>
<tbody>
<tr>
<td>7pt</td>
<td>10.96 (0.7×)</td>
<td>4.51 (0.9×)</td>
<td>24.25 (1.1×)</td>
</tr>
<tr>
<td>13pt</td>
<td>10.59 (0.8×)</td>
<td>4.39 (0.9×)</td>
<td>21.06 (1.4×)</td>
</tr>
<tr>
<td>19pt</td>
<td>9.98 (0.9×)</td>
<td>4.37 (0.9×)</td>
<td>18.84 (1.7×)</td>
</tr>
<tr>
<td>25pt</td>
<td>9.20 (1.2×)</td>
<td>4.41 (1.1×)</td>
<td>16.94 (1.8×)</td>
</tr>
<tr>
<td>125pt</td>
<td>4.08 (4.4×)</td>
<td>1.83 (4.9×)</td>
<td>11.44 (5.0×)</td>
</tr>
<tr>
<td>iso</td>
<td>6.52 (1.0×)</td>
<td>2.61 (1.1×)</td>
<td>10.56 (1.4×)</td>
</tr>
<tr>
<td>CNS</td>
<td>1.03 (2.9×)</td>
<td>0.49 (2.0×)</td>
<td>1.87 (1.9×)</td>
</tr>
</tbody>
</table>

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.

D. Performance Portability

To assess the performance portability of bricks, we adopt the performance portability metric $\Phi(a, p)$ across a set of platforms $H$ 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.

$$\Phi(a, p, H) = \frac{|H|}{\sum_{i \in H} e_i(a, p)} \quad \text{if } i \text{ is supported,}$$

$$\Phi(a, p, H) = 0 \quad \text{otherwise} \tag{1}$$

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 precision, 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 $\Phi$. Conversely, the consistently moderate efficiency for the 125-point stencil produces a moderate $\Phi$. Overall, across all platforms and stencils, we attain a performance portability $\Phi$ of 72% in double precision.  

---

**Fig. 10:** Roofline figure for benchmarks running on the P100 Pascal GPU where each dot represents our optimized brick performance for a different stencil.
vector code generation (for CPUs and GPUs), and hierarchical and the small data blocks do not have per-block ghost zones. Blocking techniques targeted large, compute-intensive stencils, and RTM on the Cell processor \[26\]. All the fine-grained blocking is explored in Bricks and Briquettes \[24\], YASK \[25\] grids, each sub-grid with its own ghost zone. Fine-grained data blocking, where the entire grid is tiled into sub-blocked data. Data along with loop tiling efforts have been 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 $2 \times 2 \times 4$ with AVX-512 instead of $4 \times 4 \times 16$. 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.

<table>
<thead>
<tr>
<th>Stencil</th>
<th>Single-Precision</th>
<th>Double-Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KNL</td>
<td>SKL</td>
</tr>
<tr>
<td>7pt</td>
<td>96%</td>
<td>83%</td>
</tr>
<tr>
<td>13pt</td>
<td>91%</td>
<td>83%</td>
</tr>
<tr>
<td>19pt</td>
<td>90%</td>
<td>83%</td>
</tr>
<tr>
<td>25pt</td>
<td>94%</td>
<td>83%</td>
</tr>
<tr>
<td>27pt</td>
<td>73%</td>
<td>76%</td>
</tr>
<tr>
<td>125pt</td>
<td>44%</td>
<td>45%</td>
</tr>
<tr>
<td>iso</td>
<td>78%</td>
<td>98%</td>
</tr>
<tr>
<td>cns</td>
<td>74%</td>
<td>60%</td>
</tr>
</tbody>
</table>

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 $\Phi_i$. 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 $2 \times 2 \times 4$ with AVX-512 instead of $4 \times 4 \times 16$. 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 \[50\], \[51\], \[52\], \[53\], \[54\].
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 allows 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 mathematicians reorganize computation to avoid the memory wall. Moreover, we demonstrate and quantify performance portability 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

APPENDIX

TILING IMPLEMENTATIONS

A. Kernel example

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

```c
#pragma omp parallel for collapse(2) schedule(dynamic, 1) proc_bind(spread)
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 + kk)
for (long j = tj; j < tj + TILEJ + jj)

#pragma omp proc_bind(closed)
for (long k = tk; k < tk + TILEK + kk)
for (long j = tj; j < tj + TILEJ + jj)

#pragma omp proc_bind(1) proc_bind(spread)
for (long tk = SH; tk < N + SH; tk += REG)
for (long tj = SH; tj < N + SH; tj += REG)
for (long ri = SH; ri < N + SH; ri += REGI)

#pragma omp parallel for collapse(2) schedule(static, 1) proc_bind(closed)
for (long tk = rk; tk < rk + REG; tk += TILE)
for (long tj = rk; tj < tj + REG; tj += TILE)
for (long ti = rk; ti < ti + REGI; ti += REGI)
```

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

```c
#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 + kk)
for (long j = tj; j < tj + TILEJ; ++j)
```

3D

```c
#pragma omp parallel for schedule(dynamic, 1) proc_bind(spread)
for (long tk = SH; tk < N + SH; tk += REG)
for (long tj = SH; tj < N + SH; tj += REG)
for (long ri = SH; ri < N + SH; ri += REGI)
```

C. GPU implementations

3D

```c
__global__ void kernel(...)
for (long tk = blockIdx.x * REG + SH; tk < N + SH)
for (long tj = blockIdx.y * TILEJ + SH + threadIdx.y; tj < N + SH + THREAD_WIDTH)
for (long k = blockIdx.x * TILEI + SH + threadIdx.x; k < N + SH + THREAD_WIDTH)
```

6D

```c
__global__ void launch_bounds...(32+TWARP, NBLOCK)
kernel(...)
for (long n = threadIdx.x * TILEI; n < N; n += NBLOCK)
for (long r = blockIdx.x / REG; r < N / REG)
for (long t = blockIdx.y / SH; t < N / SH)
for (long k = threadIdx.z; k < k + TILEI; ++k)
```

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

```c
jstride = (N + 2*SH); kstride = (N + 2*SH) * (N + 2*SH);
out[(k + kstride + j) + jstride + i] = coeff0 * in[(k + kstride + j) + jstride + i] +
coeff0 * in[(k + kstride + (i + 1)) + j + jstride + i] +
coeff0 * in[(k + kstride + (j + 1)) + j + jstride + i] +
coeff0 * in[(k + 1) + k + kstride + j + jstride + i] +
```

```c
void call_kernel(...)
for (long n = threadIdx.x * TILEI; n < N; n += NBLOCK)
for (long r = blockIdx.x / REG; r < N / REG)
for (long t = blockIdx.y / SH; t < N / SH)
for (long k = threadIdx.z; k < k + TILEI; ++k)
```