Maximizing Performance Through Memory Hierarchy-Driven Data Layout Transformations

Benjamin Sepanski
Computer Science Department
University of Texas at Austin
{ben_sepanski}@utexas.edu

Tuowen Zhao
Computer Science Department
Univeristy of Utah
{ztuowen}@cs.utah.edu

Hans Johansen, Samuel Williams
Computational Research Division
Lawrence Berkeley National Laboratory
{HJohansen, SWWilliams}@lbl.gov

***Abstract*—Computations on structured grids using standard multidimensional array layouts can incur substantial data movement costs through the memory hierarchy. This paper explores the benefits of using a framework (Bricks) to separate the complexity of data layout and optimized communication from the functional representation. To that end, we provide three novel contributions and evaluate them on several kernels taken from GENE, a phase-space fusion tokamak simulation code. We extend Bricks to support 6-dimensional arrays and kernels that operate on complex data types, and integrate Bricks with cuFFT. We demonstrate how to optimize Bricks for data reuse, spatial locality, and GPU hardware utilization achieving up to a 2.67× speedup on a single A100 GPU. We conclude with insights on how to rearchitect memory subsystems.**

I. Introduction

Whereas 2D and 3D arrays are commonplace in scientific computing (matrices, 3D structured grids), there are many other computational domains that require computations on high-dimensional structured grids. Although a tensor is the obvious exemplar, similarly high-dimensional arrays arise when one discretizes not only space, but also time or momentum (Lattice-QCD, phase-space computations, Fokker-Planck equations, etc...). Many of these operations see substantial reuse (the ratio of memory references to array size is large) and particularly among neighboring elements (an access to some x_i is likely to be followed by an access to a nearby (in N-dimensional space) $x_{i+\delta}$. As the trend in computing technology has highlighted DRAM bandwidth as the factor limiting performance, it is imperative hardware and software collaborate to ensure the vast majority of these memory references hit in the cache — a challenge exacerbated by the increasingly long reuse distances seen in computations on high-dimensional array-based data structures.

As of 2022, the most energy efficient path to exascale leverages massively-parallel systems of GPUs. Although GPUs deliver exceptional peak performance and bandwidth, they do so through massive parallelism. Similarly, as total GPU cache capacity has increased in recent years, it is shared by over ten thousand threads. When the corresponding per-thread cache capacity is small, software, framework, and compiler developers are required to orchestrate data access and data layout to maximize intra- and inter-thread data locality. One recent approach to obtain performance on these complex systems is the Bricks library [1]. Bricks stores data in small, multi-dimensional chunks of fixed size. These chunks are called *Bricks*, and have been shown to provide performance portability on CPUs and GPUs [1].

In this paper, we extend the Bricks [1] framework to support higher dimensions and complex data types. We show that the Bricks library can offer significant speedups over array-based layouts on high-dimensional kernels by simplifying performance tuning to just considering the Brick shape. To do this, we implemented some of the most critical stencils from GENE [2], [3]–a 6D physical+phase-space fusion code–using the Bricks library. In the process, we demonstrate that using Bricks can avoid the large engineering effort required to optimize stencil computations on a typical array-based memory layout. We show how to use Brick size as a proxy for memory resource usage, and demonstrate that, for wide ranges of Brick sizes, the performance effects of changing Brick shape parameters are predictable. We also implemented an FFT from GENE using the Bricks library and cuFFT to show that Bricks are highly performant on other non-stencil structured grid computations.

The contributions of this paper are as follows. (1) We extended the Bricks library to support GENE kernels by adding complex-type support and reducing metadata for high-dimensional Bricks. (2) We added FFT support for Bricks computations on NVIDIA GPUs using a cuFFT backend, and implemented an FFT kernel used in the GENE code. (3) We demonstrate how changing Brick shape corresponds to performance tuning, allowing us to completely separate correctness from optimization. (4) We provide detailed experiments demonstrating the Bricks performance tuning process for two GENE stencils and compare the performance advantages of Bricks over array-based layouts on NVIDIA A100 GPUs.

II. Background

Recovering data locality on multi-dimensional array layouts often requires loop optimizations like tiling [4], polyhedral analysis [5], or a scheduling language [6]. Some tools, such as OpenMP loop transformations [7], try to automate these optimizations. In contrast, Bricks obtains spatial locality by storing data in small, fixed-size multidimensional arrays called "Bricks" [1]. To store a d-dimensional array in a Bricked layout, one must specify a Brick shape at compile-time. Figure 1 shows an example of a 16×16 array stored in 4×4 Bricks.

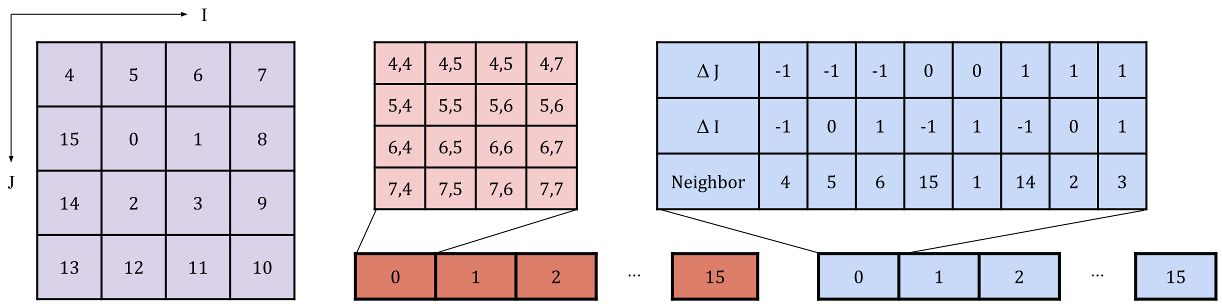

Fig. 1. Example layout of a 16×16 array stored in 4×4 Bricks. (Left) A layer of indirection maps each logical Brick location to its physical position in memory. In the above example, the Brick at logical index $(1, 1)$ is stored at index 0. (Center) Bricks are stored contiguously in memory. Each Brick is a 4×4 array. The above figure shows which array elements are stored in the Brick at index 0. (Right) In order to quickly access adjacent Bricks, each Brick has an adjacency list storing the physical indices of neighboring Bricks. The adjacency lists are stored contiguously in memory in the same order as the Bricks. The above figure shows the adjacency list for the Brick at index 0.

The Bricks library stores arrays using three data structures. The first is an array of “Bricks.” Each Brick is a small, fixed-size array stored in a traditional multi-dimensional array layout. These Bricks store the array’s data. The second data structure is an indirection map. This map takes the logical location of a Brick to its physical location in memory. For example, in Figure 1 the Brick at physical index 0 corresponds to the Brick at logical position $(i, j) = (1, 1)$. The third data structure is an array of adjacency lists, providing each Brick with the indices of its logical neighbors on a single data stream. Note that in this example, interior and surface Bricks are stored contiguously in memory, allowing MPI communication with neighbors to use contiguous memory and completely avoid “packing” of MPI message buffers.

Other multi-dimensional array libraries such as Kokkos views [8] or mdspan layouts [9] try to abstract away the underlying data layout of arrays for performance portability. However, high performance on these layouts still requires loop optimization. In contrast, Bricks embeds a d-dimensional array into a $2d$-dimensional space by adding a layer of indirection. By relying on code generation for each Brick computation, these d new parameters (the Brick shape) are free to be used for tuning in place of complex polyhedral methods. While this layout could be implemented in Kokkos or mdspan, the key benefit of Bricks comes from programmatically optimizing the data structure instead of the code.

The Bricks library [1] is a C++ header library which supports computations on Bricks by using templates to automatically translate (Brick index, index in Brick) tuples to the appropriate offset into memory, utilizing the adjacency list to handle accesses into neighboring Bricks. Bricks supports efficient, vectorized code generation for stencil computations on CPUs and GPUs [10] and efficient MPI communication with minimal message counts and no packing costs via data layout transformation [11]. Bricks reduces user effort by providing a DSL that separates stencil coding from optimized code generation [10], with Brick shape as a tuning option.

Higher-dimensional fusion application frameworks, such as GENE and COGENT [12], often have a mix of algorithms in different problem dimensions. Previous work using Bricks showed excellent results for real-valued stencil computations in 2D and 3D [10]. This work extends that and examines large, high-dimensional, complex-valued fields, stencils of much lower dimension than the problem dimension (e.g. a 2D stencil over the k-ℓ axis for each index $(\text{idx}_i, \text{idx}_j, \text{idx}_m, \text{idx}_n)$), and fields which depend on only a subset of the axes of a Brick. Further, we investigate the performance impact of the Bricks layout on other-than-stencil structured grid computations (e.g. FFTs).

III. Bricks Performance Considerations in High-Dimensional Spaces

A. GENE Proxy Benchmarks

To understand the issues facing Bricks in high-dimensional settings, we focus on three microbenchmarks as a proxy for the computations found in the full GENE code. These benchmarks are taken from GTensor [13], a software productivity library developed for the GENE project designed to generate optimized CUDA kernels and allow kernel fusion via lazy expression evaluation. These consist of a 1D stencil along the i axis, a 2D stencil with variable coefficients along the k-ℓ axes, and an FFT along the j axis, typically with size that’s a power of 2. The n-axis corresponds to a GENE species, so it is typically very small (e.g. between 1 and 3). Thus, if not otherwise specified, we use a problem size taken from examples in the GTensor repository of $I \times J \times K \times L \times M \times N = 72 \times 32 \times 24 \times 24 \times 32 \times 2$. This includes a ghost-zone of depth 2 along the i axis for the 1D stencil and ghost-zones of depth 2 along the k and ℓ axes and corners for the 2D stencil.

1) 1D Stencil: This benchmark is a fused kernel described in [13], and computes a 5-point stencil for a derivative along the i axis fused with a derivative along the j axis and scales the result. Since the j axis is represented in Fourier space, the j-derivative is computed using multiplication. Specifically, given complex-valued input field g and complex-valued fields p_1 and p_2, which do not depend on j, it computes

$$\begin{aligned} &p_1[i, k, \ell, m, n] \cdot D_i\, g[i, j, k, \ell, m, n] \\ &\quad + p_2[i, k, \ell, m, n] \cdot \imath k_y \cdot g[i, j, k, \ell, m, n], \end{aligned} \tag{1}$$

where $\imath$ in Equation 1 refers to the imaginary unit, k_y to the Fourier mode, and D_i is the 1-dimensional 5-point stencil with coefficients $\left[\frac{1}{12}, \frac{-2}{3}, 0, \frac{2}{3}, \frac{-1}{12}\right]$.

Using double precision arithmetic this stencil has a theoretical peak arithmetic intensity (FLOPs per byte) of roughly 1.12. As such, we expect this stencil to be completely memory bandwidth bound on an A100 GPU, assuming one can generate code that maximizes cache locality and memory bandwidth. (n.b, all arithmetic intensity calculations in this section are included in Appendix A).

2) 2D Stencil: This GENE kernel is also described in [13]. This 2D stencil is a 13-point, diamond-shaped, variable-coefficient stencil used in an Arakawa method [14] by GENE. While the field on which the stencil is computed is complex-valued, the coefficients are real-valued. Pseudo-code for this stencil is shown in Listing 1. This stencil has a theoretical peak

```
for i, j, k, l, m, n in index_space:
  result = 0
  for s in range(0, 13):
    c = coeffs[s](i, /* (no j) */ k, l, m, n)
    dk, dl = offsets[s]
    result += c * inp(i, j, k+dk, l+dl, m, n)
  out(i, j, k, l, m, n) = result
```

Listing 1: Pseudo-code for the 2D stencil.

arithmetic intensity of roughly 1.28, so we expect this stencil to also be heavily memory bandwidth bound on an A100 GPU despite the increase in complexity.

3) 1D FFT: Our final benchmark is a one-dimensional FFT computed along the j axis of the 6D domain. Bricks transformations have only been evaluated for stencil operations, so this benchmark evaluates the approach for applications with a mix of stencil- and non-stencil kernels. For our grid, this corresponds to an arithmetic intensity of 0.78. As such, we expect the FFT will also be heavily memory bandwidth bound.

B. Expanding the Bricks library to support GENE kernels

Extending the Bricks library to support these high-dimensional kernels required supporting computations with both real and complex-typed Bricks, specializing the metadata to facilitate low-dimensional computations on high-dimensional arrays, and building a hook-in into cuFFT.

The Bricks library stores an adjacency list for each Brick so that neighboring indices can be read in on a single prefetcher stream. This adjacency list grows exponentially with respect to the dimension. For example, in 6D each Brick has $3^d - 1 = 728$ neighbors. On a $I \times J \times K \times L \times M \times N$ grid of shape $72{\times}32{\times}24{\times}24{\times}32{\times}2$ with Bricks of size $2{\times}16{\times}2{\times}2{\times}1{\times}1$, such an adjacency list would require 1.93GB. For comparison, each complex field takes up only 1.36GB. This is clearly far too much overhead for a memory-bound computation.

The underlying issue is that most dimensions are inactive, and only some of the dimensions interact with each other. For the 1D derivative, only $3^1 - 1 = 2$ neighboring Bricks are needed to compute the stencil on each Brick. Similarly, for the 2D derivative, only $3^2 - 1 = 8$ neighboring Bricks need to be computed. To minimize this metadata, we have extended the Bricks library to support customizable adjacency lists, only storing neighbors along the active dimensions. Storing just these adjacency lists uses 5.31MB (1D) and 21.23MB (2D) and, in place of the 1.93GB 6D one, ensures the adjacency metadata movement ($\sim$1%) does not significantly reduce arithmetic intensity.

To support the FFT computation performed in GENE, we extended the Bricks library with a hook-in to cuFFT. The cuFFT callback feature allows an external function to return the memory references read from/stored to by cuFFT during its computation. Using callbacks, cuFFT can compute FFTs on non-standard array layouts (in particular, the Bricks layout).

C. Low dimensional stencils in high dimensional Bricks

In stencil computations, managing data movement is typically the key to performance [15], [16]. This process usually involves parameter tuning, which in this case includes the choice of Brick shape. We demonstrate important interpretations of the Brick shape parameters in terms of data reuse.

As recommended in [1], we compute each output Brick independently for coarse-grained parallelism and rely on the code generation technique from [10] to exploit fine-grained parallelism. With the assumption that all resources/temporaries needed for each Brick computation fit into L1 cache/registers, we can assume that only cold misses to L1 occur (we ensure this implicitly by growing the Brick size until performance starts to decrease). Then, analyzing *intra*-Brick reuse becomes tractable. The following analysis is derived in Appendix B.

1) Input field reuse: First, we show that intra-Brick data reuse of the input field is described by the Brick shape along the stencil dimensions. By intra-Brick data reuse, we mean the average reuse of each element used to compute the output Brick, i.e. the average number of times an element read from memory is used per Brick. Consider a stencil computation Δ which accesses $size$ neighbors, each at a distance of at most rad_d along axis d. For a Brick of shape $b_0 \times \cdots \times b_{D-1}$, we estimate the intra-Brick reuse R:

$$size \cdot \prod_{\substack{d=0\\ rad_d \neq 0}}^{D-1} \frac{1}{1 + 2\,rad_d / b_d} \leq \min\left(size, \prod_{\substack{d=0\\ rad_d \neq 0}}^{D-1} b_d \right) \qquad (2)$$

To understand these bounds, consider the 1D and 2D stencils from Section III-A. Letting $e^i = (1, 0, 0, 0, 0, 0)$ represent an offset of 1 in the i axis, and similarly for e^k and e^ℓ, the stencil offsets are

$$\begin{aligned}\Delta_{1D} &= \{-2e^i, -e^i, 0, e^i, 2e^i\}\\ \Delta_{2D} &= \{-2e^\ell, -e^k - e^\ell, -e^\ell, e^k - e^\ell, -2e^k, -e^k, 0,\\ &\qquad e^k, 2e^k, -e^k + e^\ell, e^\ell, e^k + e^\ell, 2e^\ell\}.\end{aligned}$$

Applying Equation 2, the intra-Brick input field reuses R_{1D} and R_{2D} are in the range

$$\frac{5}{1+4/b_i} \leq R_{1D} \leq \min(5, b_i) \tag{3}$$

$$\frac{13}{(1+4/b_k)(1+4/b_\ell)} \leq R_{2D} \leq \min(13, b_k b_\ell). \tag{4}$$

In particular, the 1D stencil's access pattern encourages increasing b_i, while the 2D stencil incentivizes increasing b_k and b_ℓ. Thus, for a fixed Brick size, we see that relying on intra-Brick field reuse for some stencils in an application may force us to rely on inter-Brick field reuse for others.

2) Auxiliary field reuse: Intra-Brick reuse for auxiliary fields is described by the Brick dimensions that the auxiliary fields do *not* depend on. For example, both stencils contain auxiliary fields which do *not* depend on the j-axis. Any two points which differ only in their j coordinate depend on the same auxiliary field values, exposing an opportunity for reuse.

For Bricks, the amount of reuse is explicit. If there are N input field elements stored in Bricks of size B, and N_{aux} auxiliary field elements stored in Bricks of size B_{aux}, then the number of auxiliary field elements loaded is

$$\frac{N \cdot B_{aux}}{B}, \tag{5}$$

with reuse factor

$$R_{aux} := \frac{B}{B_{aux}}. \tag{6}$$

For example, in both the 1D and 2D stencil, the reuse of each scalar field is just the j Brick shape parameter, b_j.

Note that in Equation 5, for a fixed Brick shape the number of auxiliary coefficients loaded depends on the number of *input* field elements. This is because each output Brick is computed independently. Importantly, since many fields do not depend on the j axis across both stencils, increasing b_j can dramatically improve the performance of both the 1D and 2D stencils, even though the auxiliary fields are much smaller than the input fields.

3) Using Reuse Estimates: Of course, in this entire discussion, the cost of loads is dependent on where the data reside inside the memory hierarchy (i.e. *inter-Brick* reuse). For example, because (in our implementation) the Bricks are laid out contiguously in i, input fields may achieve a much higher reuse than is indicated by Equation 3.

However, holding Brick iteration order constant, Equations 2 and 6 provide easy translations from the structure of the computation into estimates of data reuse as a function of Brick shape, and in practice can explain many of the performance effects observed in reasonably-sized Bricks. Since many stencils are bandwidth bound, optimizing reuse is necessary for high performance. Using these equations to optimize reuse changes the search for a Brick shape from a general tuning problem to a guided search.

IV. Results

A. NVIDIA A100 GPU

In this paper, we use a GPU-accelerated system where each node consists of four NVIDIA Ampere A100 GPUs and one AMD EPYC 7763 (Milan) CPU. Each GPU consists of 108 streaming multiprocessors (SMs) and a total of 3,456 FP64 cores clocked at 1.41GHz, providing a theoretical (non-Tensor core) peak double-precision performance of 9.7 TFLOP/s. Each A100 has 40 GiB HBM2 memory at the maximum bandwidth of 1,552.2 GB/s. This produces a machine balance of about 6.5 FLOPs per byte. Kernels with an arithmetic intensity less than 6.5 will ultimately be memory-bound and unable to use the full compute capability. [1]

B. Performance Methodology

To evaluate the benefits of Bricks in these high-dimensional settings, we wish to demonstrate that Bricks separates performance concerns from correctness, achieves highly performant stencil code, and is performant for non-stencil computations.

We use the Roofline model [17] to evaluate the performance potential of these kernels in terms of not only their achieved floating-point performance, but also their cache performance (in that any unnecessary bytes moved affect arithmetic intensity). Since all of the kernels considered are bandwidth-bound, we analyze performance by measuring and comparing the volume of data moved to/from HBM, measured using the NSight Compute Profiler. As the number of floating-point operations is fixed within each kernel, performance is dependent on data volume and bandwidth, and the arithmetic intensities presented earlier denote goalposts. As applications written in Bricks create, load, operate, and store data in a Bricked layout for the entirety of the application, there is nominally zero overhead for data layout conversions. If one required interoperability with an external library, the overhead would be twice the number of elements (read from one layout; write to the other).

We intend to demonstrate that, using the insights in Equations 2 and 6, the Bricks library provides predictable performance behavior. Then, we show how to use this predictability to achieve near-roofline performance.

Finally, we address the concerns that Bricks may negatively affect non-stencil computations. We implement the FFT over the j axis as described in Section III-A3 using cuFFT, and investigate the performance characteristics of performing an FFT along a line spanning multiple Bricks. As GTensor uses cuFFT to compute FFTs on the GPU, this is a reasonable comparison of optimized FFT implementations accessing data either in the traditional array layout or a Brick layout.

C. Stencils on NVIDIA A100

1) Performance tuning using Brick shape: Since we can rely on the Bricks code generation to write the kernel, we investigate the first question by measuring data movement for

[1]Experimental setup and code can be found at https://github.com/benSepanski/bricklib/tree/mchpc_0.0/gene

various Brick shapes. In Figures 2 and 3, we show the total amount of data movement for the 1D and 2D stencils over a large set of Brick shape parameters. We investigate whether Equations 2 and 6 can be used to tune Bricks for performance.

One notable anomaly in Figure 2 is the 1D stencil with Brick size 1024 with $b_i = 4$. This was traced to a compiler optimization decision, where several loops are not unrolled, preventing key memory optimizations. In future work, outer dimensions could be collapsed using code generation.

Equations 2 and 6 attempt to predict intra-Brick data reuse. To see how useful these predictions are, we write the estimated global data volume $DataVol$ in terms of the size of the output field $size_{out}$, the predicted input field reuse R_{inp} (taken from the lower bound in Equation 2), the auxiliary field reuse R_{aux} (defined as in Equation 6), and N, N_{aux} (as defined as in Equation 5). We then find a least squares fit of Equation 7 to the data in Figures 2 and 3.

$$DataVol = size_{out} + \frac{size(\Delta) \cdot c_{inp}}{R_{inp}} + \frac{N \cdot c_{aux}}{N_{aux} R_{aux}}. \quad (7)$$

Intuitively, c_{inp} (resp. c_{aux}) represents the total size in GB of the input field (resp. auxiliary scalar fields). Note that the $size(\Delta)$ (resp. N/N_{aux}) factor is just a normalization factor: the number of times each element is used.

First observe that the model in Equation 7 is an especially good fit for the 2D stencil, and fits both stencils well. A standard t-test fails to reject the fit with $p < 10^{-4}$.

Our fit for the 1D stencil estimates $c_{inp} \pm \sigma_{inp} = 1.53 \pm 0.16$ GB. This is slightly larger than the size of the input array (1.36 GB), indicating that the actual reuse is slightly lower than predicted. In Figure 2, we see no marked decrease in data movement as the Brick-dimension increases along the i axis. As mentioned in Section III-C, this is likely because the reuse is occurring between Bricks even for small b_i, due to the simpler 1D stencil access pattern.

In contrast, Figure 3 shows a clear dependence of $DataVol$ on the 2D Brick stencil axes (namely, b_k and b_ℓ). Our fit estimates $c_{inp} \pm \sigma_{inp} = 0.74 \pm 0.05$ GB, well under the size of the input array (1.36 GB). This suggests that the actual reuse

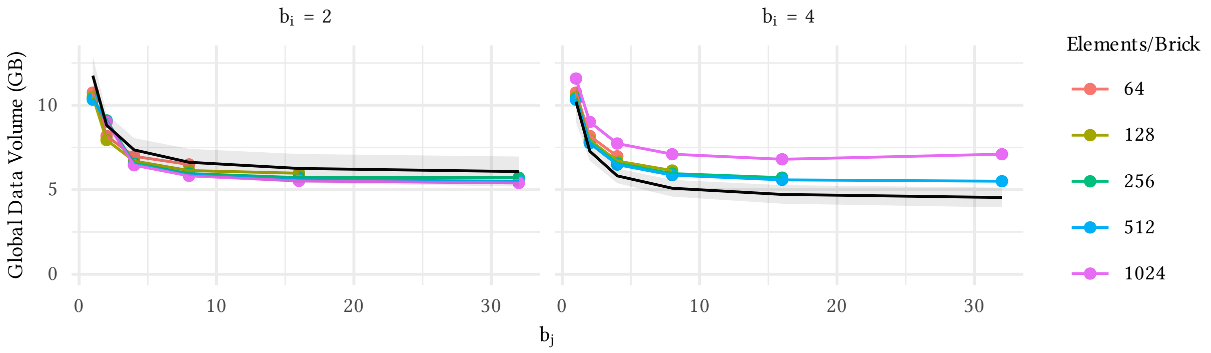

Fig. 2. Total volume of data loaded to/stored from L1 in the CUDA global namespace (i.e. user data) during computation of the 1D Stencil as Brick dimension j changes. The solid black line corresponds to a least squares fit of our reuse predictions to the data volume with a 95% confidence interval shown in gray. This kernel is bandwidth bound, so lower data movement is better. Equation 6 can be seen in the inverse relationship between global data volume and b_j–the predicted auxiliary field reuse factor. Note that, although the auxiliary fields are 32× smaller than the input field, they can affect total data volume by a factor of 2×. The 1024-sized Brick with $b_i = 4$ exhibits anomalous behavior due to a change in loop unrolling decisions made by NVCC which prevents key memory optimizations.

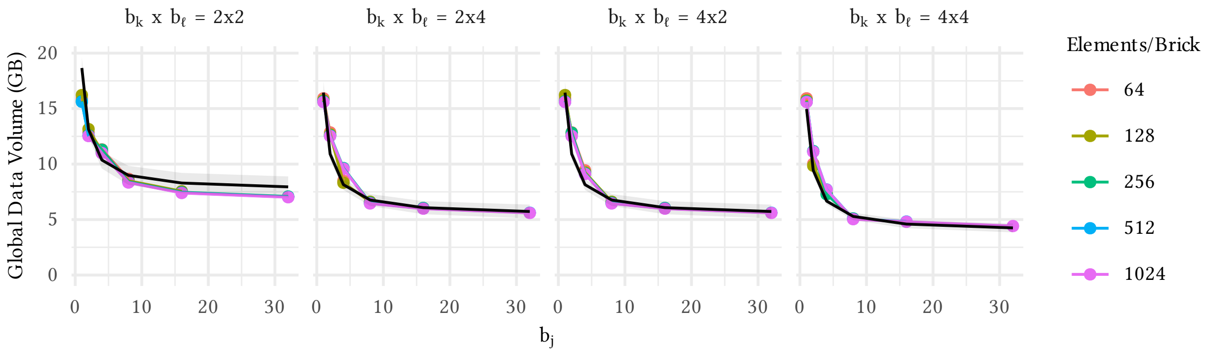

Fig. 3. Total volume of data loaded to/stored from L1 in the CUDA global namespace (i.e. user data) during computation of the 2D Stencil as Brick dimension j changes. The solid black line corresponds to a least squares fit of our reuse predictions to the data volume with a 95% confidence interval shown in gray. This kernel is bandwidth-bound, so lower data movement is better. As described by Equation 2, data volume decreases as b_k and b_ℓ increase due to reuse of the input field. Even though the coefficients are 40% smaller than the input field, we see they make up a large component of the data volume. This component is inversely proportional to b_j, as expected from Equation 6. Increasing coefficient reuse can improve data volume by over a factor of 2

is higher than predicted, likely due to inter-Brick reuse.

In both figures, the pattern of coefficient reuse is readily apparent. The $DataVol \propto 1/b_j$ pattern evinces itself clearly. Our fit predicts $c_{aux} \pm \sigma_{aux} = 0.18 \pm 0.02$ for the 1D stencil, about $2.3\times$ larger than the 0.080 GB of auxiliary fields. Our fit estimates $c_{aux} \pm \sigma_{aux} = 0.34 \pm 0.02$ GB for the 2D stencil, about $1.8\times$ larger than the 0.19 GB auxiliary field. In both cases, the actual array size is much smaller than c_{aux}, suggesting that even inside of a Brick, the auxiliary coefficients do not attain their full potential reuse.

Despite the large amounts of inter-Brick reuse, the model in Equation 7 captures the primary data movement characteristics of both stencils quite well. Our above observations suggest that Equations 2 and 6 describe the dominant trends in global data movement (in L1, for reasonably sized Bricks). Thus, Equations 2 and 6 provide useful models to evaluate expected data reuse of a Brick shape *relative to other Brick shapes*. Further, although auxiliary fields can be more than ten times smaller than the input fields, the Bricks library and CUDA scheduler are apt to ignore them in order to take full advantage of stencil access patterns.

2) Understanding memory resource limits through Brick size: The reasoning in Section III-C relies on the assumption that the working set of the computation fits in cache, for a Brick size "small enough" that the assumption held. Figure 4 investigates the implications of this assumption.

Since both kernels are bandwidth-bound, lower data volume is better. In particular, all 1D configurations are DRAM-bound. Some 2D configurations with small $b_j \leq 8$ and Brick size ≤ 128 are L2-bound. The remainder are DRAM-bound. Thus, in Figure 4, data movement through L2 and DRAM is the performance bottleneck.

First, we observe that decisions made by NVCC on what to put in registers or local storage can have massive, anomalous effects on memory spilling out to DRAM/L2. For example, the large peak for $b_j \in \{16, 32\}$ and Brick size of 512 in Figure 4 is due to NVCC storing a large buffer in registers for the duration of the kernel, forcing many temporaries into local memory. The curve $b_j = 2$ in Figure 4 is much lower than the others for Brick sizes greater than 512 elements due to similar decisions. Again, these could potentially be addressed through code generation optimizations.

Next, we notice that while the global data movement in L1 (user data) is relatively constant with respect to Brick size, the local data movement is not. In fact, the data movement through L2 and DRAM is shaped almost exactly like the local data movement lines in L1. This suggests that, for GPUs, cache and capacity misses to L2 and DRAM are largely determined by local data movement. Further, local data movement is roughly monotonic in Brick size until Bricks become very large (more than 512 elements per Brick). In particular, increasing Brick dimensions to take advantage of intra-Brick reuse as described in Equations 2 and 6 will eventually decrease performance because of local data movement. This leads to a natural, two-

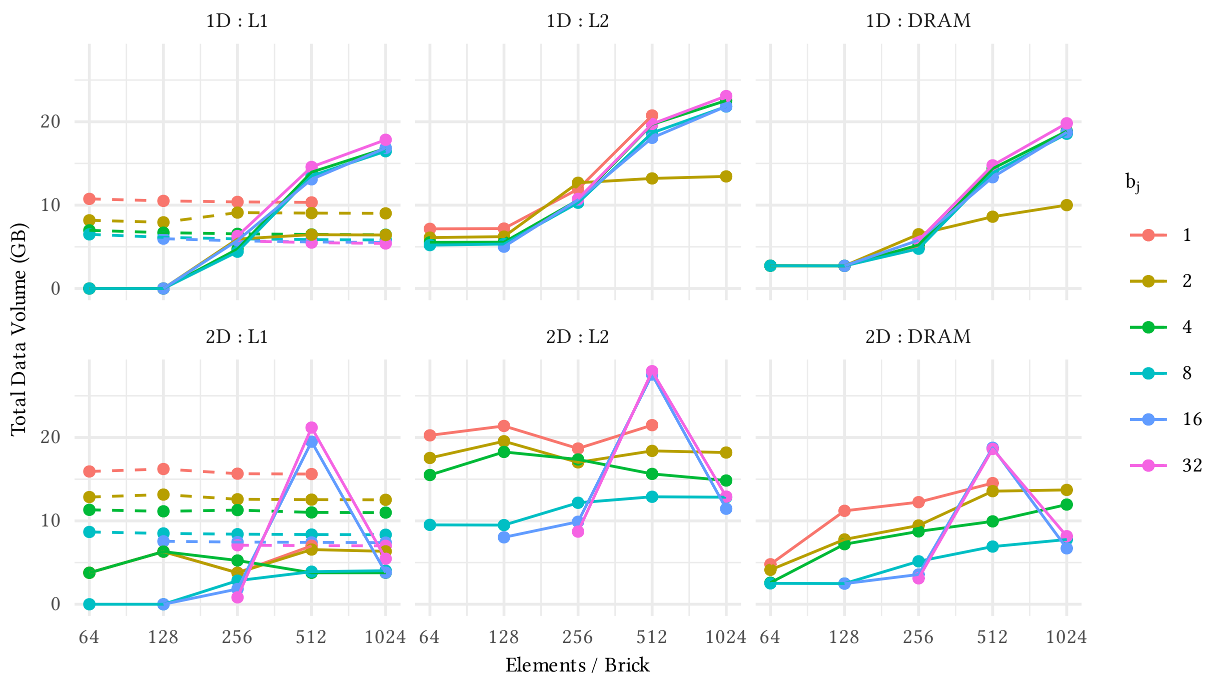

Fig. 4. Total data volume from/to L1, L2, and DRAM for both stencils as Brick size changes. Brick dimensions $b_i \times b_k \times b_\ell \times b_n$ are fixed at $2 \times 2 \times 2 \times 1$. Data Volume from/to L1 is split into global (user) data (dashed line) and local data (solid line). Since both kernels are bandwidth-bound, lower data volume corresponds to higher performance. Note that the user data moved is relatively constant with respect to Brick size, while the local data moved increases with Brick size, and determines the trend of data movement down to DRAM. Anomalous behavior begins to occur at large Brick sizes (e.g. 512 elements/Brick in 2D) due to unpredictable compiler decisions surrounding loop unrolling and register allocation. Note that the L2 traffic presented is only L2 $\leftrightarrow$ L1, as L2 $\leftrightarrow$ DRAM is equal to DRAM data volume.

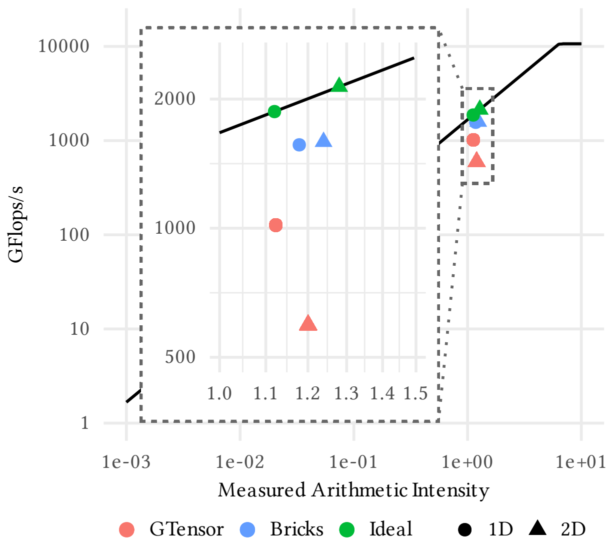

Fig. 5. Truncated window of a Roofline plot for stencils on a single device. Bricks delivers a 3.2% increase in arithmetic intensity in 2D and a 2.67× increase in performance. In 1D, Bricks achieves a 4.9% increase in arithmetic intensityand a 1.54× increase in performance.

phased tuning strategy: First, fix a Brick size and optimize auxiliary coefficient and input field reuse using the intra-Brick estimates from Section III-C. Then, increase the Brick size and repeat, stopping when performance no longer increases.

3) Tuning for GENE: We apply the procedure described above with the microbenchmarks taken from the GENE code. Since both 1D and 2D stencils involve reuse on j, we wish to choose b_j as large as possible. Consequently, we must set all other dimensions as small as possible, relying on inter-Brick reuse for the input fields. We begin with a small Brick size and increase it until performance degrades. For our grid, this corresponds to $b_j = 16$ (with $b_i = 2$, $b_k = 2$, and $b_\ell = 2$ since the stencils have radius 2 along the i, k, and ℓ axes). This process achieves performance (GFLOP/s) within 0.02% of the top Brick shape for both stencils.

Having seen that the Bricks layout is indeed tunable without the need for a deep understanding of the GPU architecture, we next compare it to GTENSOR in Figure 5. We see that both implementations are very close to the roofline, with Bricks achieving a 2.67× speedup for the 2D stencil and a 1.53× speedup on the 1D stencil. For both stencils, Bricks achieves theoretically minimal data movement[2].

Note that, since GTENSOR already achieves near the data movement lower bound Bricks has almost no room to improve on data movement alone. In fact, Bricks only achieves a 4.9% increase in AI for the 1D stencil and a 3.2% increase in AI for the 2D stencil. The speedup is likely due to achieved occupancy (average percentage of warps active during the computation). GTENSOR ranged in occupancy from 90% to 95% (for the 2D and 1D stencils), while Bricks remained in the 46% to 50% range across all Brick shapes.

[2]Nsight Compute slightly undercounts data movement, which accounts for the larger than theoretical arithmetic intensity in 1D

Somewhat counter-intuitively, the lower occupancy can explain the higher performance. In an application of Little's Law [18], once the achieved parallelism is sufficient to meet the bandwidth needs, higher parallelism does not provide an additional benefit. Rather, the large number of concurrent warps take up extra cache space. This results in a 2.55× reduction in L1 data movement for the 2D stencil (19.25 GB to 7.55 GB) and a 1.98× reduction in L1 data movement for the 1D stencil (11.82 GB to 5.98 GB), roughly corresponding to the increase in performance over GTENSOR.

D. FFTs on NVIDIA A100

Bricks are a high-performance solution for stencil-type kernels, but one might be concerned that they negatively affect the performance of other crucial kernels, slowing the entire application. To test this, we used the cuFFT callback feature to run the NVIDIA library directly on the Bricks layout. A critical consideration is that performance is not competitive without device link-time optimization, available starting in CUDA 11.2, due to function call overhead.

For a Brick layout, the performance of the FFT should only depend on the Brick shape in the j axis, and possibly the shape in the i axis since it affects the stride. We found that cuFFT's performance was independent of the shape in the i axis, so we only discuss effects of the j direction.

In these experiments we use a $72 \times 32 \times 24 \times 24 \times 32 \times 2$ array, and compute an FFT over the j-axis. We consider two approaches: using cuFFT callbacks to perform an FFT directly on the j axis, or transposing the i-j axes before and after an FFT. Since cuFFT does not currently support an FFT directly on a "middle" axis (i.e. j in ijk), either a transpose or using callbacks is necessary even for array layouts. The GENE code, for instance, uses a transpose.

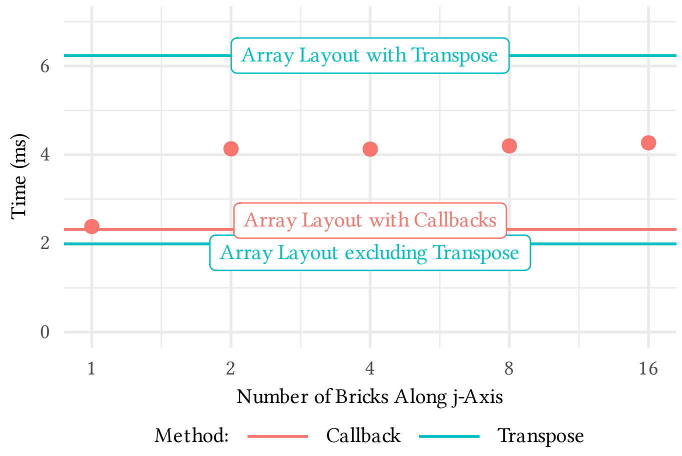

Fig. 6. Time to compute FFT as a function of the number of Bricks along the j-axis. Horizontal lines correspond to cuFFT on an array layout. The top line times an FFT with a transpose before and after (the approach taken by GENE), the bottom line times the same FFT excluding transpose time, and the middle line times the implementation using callbacks. The Bricks layout is just as performant as the array layout for a single Brick/FFT, and causes a slowdown between 1.83× and 1.93× using callbacks with more than one Brick/FFT.

We performed this computation for various Brick sizes to investigate how the FFT performance scaled as the number of Bricks along the FFT-axis increased (we begin with $2 \times 32 \times 2 \times 2 \times 1 \times 1$-shaped Brick, and repeatedly shift data from b_j into b_m). For each Brick shape, we timed 100 runs of the cuFFT computation after 10 warm-up iterations. We repeated this procedure 25 times to obtain a 95% confidence interval, presented in Appendix C. Figure 6 displays the results.

To understand the potential optimal performance, consider the bottom-most blue horizontal line, which shows the average time to compute 32-point FFTs on a (pre-transposed) $32 \times 72 \times 24 \times 24 \times 32 \times 2$ array along the i axis. Since 32 is a small power of 2 and in the contiguous direction, we expect this is an lower bound on time using cuFFT, taking only 1.99 ± 0.002 ms. Note, the transpose in Figure 6 has significant overhead.

No matter how much the transpose is optimized, streaming 1.36 GB twice at the A100's peak bandwidth of 1,552.2 GB/s still takes at least 1.75 ms, slowing computation by $1.88\times$.

While the callbacks add software complexity, they avoid the overhead of a transpose. The red horizontal line shows the performance of cuFFT using callbacks on an array layout. This implementation takes only 2.32 ± 0.02 ms, within 17% of the best performance cuFFT could potentially offer. Bricks with callbacks is just as performant when a single Brick spans the j-axis. The cost for using multiple Bricks in the FFT nearly doubles when moving from one to two Bricks in the direction of the FFT. However, there is very little additional cost in moving from two Bricks up to sixteen, with a slowdown ranging from $1.83\times$ to $1.93\times$ relative to a single domain-spanning brick.

It is clear that using Bricks with cuFFT comes at a cost. However, the built-in ability to use callbacks makes Bricks already an improvement over transpose-based implementations. Further, we are relying on cuFFT to optimize the accesses–a software package with no knowledge of the Bricks layout. Since cuFFT has no prior knowledge over what the callbacks are doing, it must make some assumptions to remain performant. The near optimality of the array layout using callbacks suggests that cuFFT may be assuming the elements of each FFT are laid out at a constant stride.

V. Discussion and Conclusions

We have demonstrated that Bricks can achieve near-theoretical peak performance competitive with custom solutions for high-dimensional stencil computations on GPUs. This performance can be understood through interpretable models based on the Brick shape parameters for three kernels (1D, 2D stencils and FFT) on the A100 GPU architecture.

Stencil computations on high-dimensional arrays where the dimensions contain moderate powers-of-two often produce large numbers of conflict misses as striding in least contiguous dimensions exposes the product of all the powers-of-two across all the lower dimensions. We observe that Bricks delivers near optimal cache performance for stencil computations and can thus infer few if any conflict misses. This suggests rearchitecting memory subsystems and memory access for brick-based computations in order to maximize performance and energy efficiency. That is, for a given cache capacity, Brick-based data layouts would allow computer architects to reduce cache associativity and increase cache line size while preserving the number of sets in the cache. The former reduces cache access energy (fewer tags to compare) while the latter improves memory access efficiency (more data per request, possibly reaching parity with the DRAM bank size). However, as the hierarchy of reuse distances will still be present with Brick-based layouts (albeit to a lesser degree), replacing caches with scratch pads is beneficial on computations with low dimensionality.

Because computations on the elements within bricks are performed in a data parallel manner, the adoption of Bricks as a data layout incentivizes longer vectors (bound by grid dimensionality and the depth of data that must be communicated between processes in each dimension). Alternatively, it could enable asynchronous, DMA-like software prefetching to mitigate the complexity of expressing memory parallelism instead of out-of-order execution, extreme multithreading, or hardware stream prefetchers.

Our investigation of cuFFT on Bricks in Section IV-D used a standard vendor library with *no knowledge* of the Bricks layout. While an FFT across multiple Bricks is not accessed at a constant stride, it is accessed in a regular fashion across each Brick. Increasing callback support to pass additional layout information, such as the FFT stride inside of a Brick or the number of FFTs performed per Brick, could prevent overhead from using the Bricks layout. Improving Bricks support for non-stencil operations – like FFTs, tensor contractions or linear algebra – widens its applicability, further abstracting the underlying Bricks layout and allowing users to simply rely on its DSL and code generation.

In future work, we also intend to apply similar methods to other high-dimensional problems such as QCD [19] in order to test the generalizability of our results.

Acknowledgements

This research was supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231, the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number(s) DE-SC0021110, and used resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] T. Zhao, S. Williams, M. Hall, and H. Johansen, "Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 59–70. [Online]. Available: https://doi.org/10.1109/P3HPC.2018.00009

[2] T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told, "The global version of the gyrokinetic turbulence code GENE," *Journal of Computational Physics*, vol. 230, no. 18, pp. 7053–7071, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999111003457

[3] F. Jenko, W. Dorland, M. Kotschenreuther, and B. N. Rogers, "Electron temperature gradient driven turbulence," *Physics of Plasmas*, vol. 7, no. 5, pp. 1904–1910, May 2000. [Online]. Available: https://doi.org/10.1063/1.874014

[4] M. Wolfe, "More iteration space tiling," in *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, 1989, pp. 655–664. [Online]. Available: https://doi.org/10.1145/76263.76337

[5] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-Polyhedral optimization in LLVM," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1. [Online]. Available: https://doi.org/10.1142/S0129626412500107

[6] 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," *SIGPLAN Not.*, vol. 48, no. 6, p. 519–530, jun 2013. [Online]. Available: https://doi.org/10.1145/2499370.2462176

[7] "OPENMP API Specification: Version 5.1 November 2020," November 2020. [Online]. Available: https://www.openmp.org/spec-html/5.1/openmpsu53.html

[8] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[9] D. S. Hollman, B. A. Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland, and C. R. Trott, "mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 60–70.

[10] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen, "Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356210

[11] T. Zhao, M. Hall, H. Johansen, and S. Williams, "Improving Communication by Optimizing On-Node Data Movement with Data Layout," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2021, p. 304–317. [Online]. Available: https://doi.org/10.1145/3437801.3441598

[12] W. Lee, M. A. Dorf, M. R. Dorr, R. H. Cohen, T. D. Rognlien, J. A. F. Hittinger, M. V. Umansky, and S. I. Krasheninnikov, "Verification of 5D continuum gyrokinetic code COGENT: Studies of kinetic drift wave instability," *Contributions to Plasma Physics*, vol. 58, no. 6-8, pp. 445–450, Apr. 2018. [Online]. Available: https://doi.org/10.1002/ctpp.201700161

[13] K. Germaschewski, B. Allen, T. Dannert, M. Hrywniak, J. Donaghy, G. Merlo, S. Ethier, E. D'Azevedo, F. Jenko, and A. Bhattacharjee, "Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU," *Physics of Plasmas*, vol. 28, no. 6, p. 062501, 2021. [Online]. Available: https://doi.org/10.1063/5.0046327

[14] A. Arakawa, "Computational design for long-term numerical integration of the equations of fluid motion: Two-dimensional incompressible flow. Part I," *Journal of Computational Physics*, vol. 1, no. 1, pp. 119–143, 1966. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0021999166900155

[15] 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 *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. USA: IEEE Computer Society, 2010, p. 1–13. [Online]. Available: https://doi.org/10.1109/SC.2010.2

[16] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The Potential of the Cell Processor for Scientific Computing," in *Proceedings of the 3rd Conference on Computing Frontiers*, ser. CF '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 9–20. [Online]. Available: https://doi.org/10.1145/1128022.1128027

[17] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[18] H. Kim, "Probability, Statistics, and Random Processes for Engineers," *Technometrics*, vol. 46, no. 3, pp. 360–361, 2004. [Online]. Available: https://doi.org/10.1198/004017004000000400

[19] M. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi, "Solving lattice QCD systems of equations using mixed precision solvers on GPUs," *Computer Physics Communications*, vol. 181, no. 9, pp. 1517–1528, Sep. 2010. [Online]. Available: https://doi.org/10.1016/j.cpc.2010.05.002

[20] C. Angerer, "CUDA Pro Tip: Use cuFFT Callbacks for Custom Data Processing," Sep 2014. [Online]. Available: https://developer.nvidia.com/blog/cuda-pro-tip-use-cufft-callbacks-custom-data-processing/

Appendix A
Arithmetic Intensity Computations

In this section we derive the arithmetic intensity estimates mentioned in Section III-A. For each point on the grid, the 1D stencil performs three complex multiplies (18 FLOPs), five real-complex multiplies (10 FLOPs), and five complex additions (10 FLOPs). With $I \times J \times K \times L \times M \times N = 72 \times 32 \times 24 \times 24 \times 32 \times 2$,

$$\frac{38(I-4)JKLMN}{16 * (IJKLMN + (I-4)JKLMN + J)} \approx 1.12.$$

The 2D stencil performs 13 real-complex multiplies (26 FLOPs) and 12 complex additions (24 FLOPs) at each grid point, so its arithmetic intensity is

$$\frac{50IJ(K-4)(L-4)MN}{16 * (IJKLMN + IJ(K-4)(L-4)MN)} \approx 1.28.$$

Finally, the 1D FFT along the j axis performs $IKLMN$ independent FFTs of size J, with a stride of I between each element of an FFT. Since cuFFT is an "in-place" FFT [20], it has an arithmetic intensity of $\frac{5}{32}\log_2(J) = 0.78$ [16].

Appendix B
Brick Reuse

In this section, we derive the reuse estimates in Equations 2, 5, and 6. First, we establish some notation. In what follows, for two integers $a, b \in \mathbb{Z}$, we define

$$[a : b] = \{i \in \mathbb{Z} \mid a \leq i < b\}.$$

Given two sets A and B, we write $A + B$ for the Minkowski sum $\{a + b : a \in A, b \in B\}$.

Let I be the index space of the Brick on which the stencil is being computed. Note that $size(I + \Delta)$ many input elements are read in, and a total of $size(I) \cdot size(\Delta)$ input elements are used in the computation. Since $size(I)$ is the size of the Brick, our reuse factor is

$$\frac{size(\Delta) \prod_{d=0}^{D-1} b_d}{size(I + \Delta)}. \tag{8}$$

However, this greatly simplifies if there are any dimensions along which Δ is constant. Then, we can "project out" dimension d from the set $I + \Delta$.

More concretely, given shifts $\Delta = \{\boldsymbol{\delta}^1, \ldots, \boldsymbol{\delta}^n\}$ and $Z = \{z_0, \ldots, z_{m-1}\}$ with $0 \leq z_0 < z_1 < \cdots < z_{m-1} < D$, define $\boldsymbol{\delta}_Z^n = (\delta_{z_0}^n, \delta_{z_1}^n, \ldots, \delta_{z_{m-1}}^n)$, $\Delta_Z = \{\boldsymbol{\delta}_Z^1, \ldots, \boldsymbol{\delta}_Z^N\}$, and $I_Z = \bigtimes_{d=0}^{m-1}[0 : b_{z_d}]$. If Δ is constant along dimensions $[0 : d] - Z$, then

$$size(I + \Delta) = size(I_Z + \Delta_Z) \cdot \prod_{d \notin Z} b_d.$$

Therefore, defining $ax(\Delta)$ to be the largest set along which Δ is non-constant for each $d \in ax(\Delta)$, the reuse factor is

$$R = \frac{size(\Delta)}{size(I_{ax(\Delta)} + \Delta_{ax(\Delta)})} \prod_{d \in ax(\Delta)} b_d. \tag{9}$$

Where Δ is clear, we write ax for $ax(\Delta)$. To estimate $size(I_{ax} + \Delta_{ax})$, we use the radius of the stencil. Define $rad(\Delta)$ to be the D-dimensional vector whose ith entry is $\max\{|\delta_i| : \delta \in \Delta\}$. Then,

$$\begin{aligned} \max\left(\prod_{d \in ax} b_d, size(\Delta)\right) &= \max(size(I_{ax}), size(\Delta_{ax})) \\ &\leq size(I_{ax} + \Delta_{ax}) \\ &\leq \prod_{d \in ax} (b_d + 2rad(\Delta)_d). \end{aligned}$$

Plugging this into Equation 9, our reuse factor R is between

$$size(\Delta) \prod_{d \in ax} \frac{b_d}{b_d + 2rad_d(\Delta)} \leq R \leq \min\left(size(\Delta), \prod_{d \in ax} b_d\right).$$

Dividing each fraction on the left-hand side by b_d, we obtain Equation 2.

To obtain Equation 5, assume that each input/output field-Brick is of size B, that the axes of the auxiliary field depend on a subset of the axes of the input field, and the auxiliary field in question is stored in bricks of size B_{aux} with dimensions that match the input field brick along shared axes. Suppose there are N total output elements. Then, the number of auxiliary elements loaded is

$$\sum_{b=0}^{N/B-1} B_{aux} = \frac{N}{B} \cdot B_{aux},$$

Which is exactly Equation 5. Observing that each of the N_{aux} auxiliary elements is used N/N_{aux} times, we arrive at Equation 6 by dividing the number of uses, N, by the number of auxiliary elements loaded.

Appendix C
Results Uncertainty

We've included uncertainty estimates from the FFT experiments detailed in Section IV-D in Table I.

TABLE I
STANDARD DEVIATIONS OF REPORTED FFT RUNTIMES

Layout	Method	b_j	Mean (ms)	σ (ms)
Array	Transpose (FFT Only)	-	1.99E+00	4.63E-03
Array	Transpose	-	6.24E+00	5.39E-03
Array	Callback	-	2.32E+00	3.93E-02
Bricks	Callback	2	4.27E+00	3.66E-04
Bricks	Callback	4	4.20E+00	5.00E-04
Bricks	Callback	8	4.13E+00	4.16E-04
Bricks	Callback	16	4.13E+00	4.54E-04
Bricks	Callback	32	2.38E+00	4.40E-04
Bricks	Transpose	2	8.07E+00	4.64E-04
Bricks	Transpose	4	8.01E+00	2.65E-04
Bricks	Transpose	8	8.04E+00	3.74E-04
Bricks	Transpose	16	8.04E+00	2.72E-04
Bricks	Transpose	32	6.41E+00	8.61E-04