Breaking the Memory Bottleneck for Iterative Memory-bound Applications Via Persistent Kernels

Lingqi ZHANG[1,3,a)] Mohamed Wahib[2,b)] Peng Chen[3,c)] Jintao Meng[4,d)] Xiao Wang[5,e)]
Toshio Endo[1,f)] Satoshi Matsuoka[2,1,g)]

Abstract: Iterative memory-bound solvers commonly occur in HPC codes. Spatial blocking optimizations of iterative solvers are directed towards improving the data locality of the code executed within a single time step of the solver. Temporal blocking optimizations combine multiple consecutive iterations in a scheme that requires the resolution of neighborhood dependencies. We propose a novel data-locality optimization scheme for memory-bound iterative kernels: PERsistent KernelS (PERKS). In this scheme, we target the elimination or reduction of data movements occurring in-between time steps. We eliminate or reduce the traffic to the memory by caching a subset of the output in each time step on on-chip resources to be used as input for the following time step. PERKS can be generalized to any iterative solver: they are largely independent of the solver's implementation, and run independently on top of spatial/temporal blocking optimizations. We implement PERKS in CUDA since Nvidia GPUs provide low latency device-wide synchronizations and a large volume of on-chip resources, i.e., scratch-pad memory and register files. We explain the design principle of PERKS and demonstrate the effectiveness of PERKS for a wide range of iterative 2D/3D stencil benchmarks (geomean speedup of 2.35x for 2D stencils and 1.53x for 3D stencils).

Keywords: Iterative Solvers, Stencil, Conjugate Gradient Solvers, Persistent GPU Kernels

1. Introduction

Iterative solvers are ubiquitous in High Performance Computing (HPC). For example, iterative stencils [1–4] are used widely in numerical solvers for PDEs, iterative stationary methods are used for solving systems of linear equations (ex: Jacobi [5, 6] and Gauss–Seidel method [6–8]), and iterative Krylov subspace methods are typically used for solving systems of linear equations (ex: conjugate gradient [9, 10], BiCG [10, 11], and GMRES [10, 12]).

Given that iterative stencils and Krylov subspace solvers typically have low arithmetic intensity [3], significant research effort goes into optimizing them for data locality. Those effort include moving the bottleneck from memory to cache on CPUs [13, 14] or on-chip scratchpad memory on GPUs [15, 16]. Other works further push the bottleneck to be the register files [4, 17]. Those efforts become increasingly effective on GPUs, particularly so since the aggregate volume of register files and scratchpad memory capacity is increasing with newer generations of GPUs [18]. While the emphasis on optimizing iterative methods goes into reducing the memory traffic within each time step, it is important to note that a significant part of the time in iterative solvers goes into storing the output of each time step and then loading it again to use as an input in the following time step. For example, Figure 3 shows the time for inter-step data storing and loading to be a majority of runtime for a 2D 5-point stencil running on an Nvidia A100 GPU).

One opportunity to improve the data locality is to extend the lifetime of the solver across time steps and reduce the inter-step traffic to the memory.

In this paper, we propose a generic scheme for running iterative solvers to improve inter-step data locality. PERsistent KernelS (PERKS)[*1] are used to advance the solver over all, or some, of the time steps. PERKS requires two basic features to function. First, due to the spatial neighborhood dependencies in iterative solvers, a chip-wide barrier is required at the end of each time step (or several time steps when doing temporal blocking [3]). That is to assure that advancing the solution in time step k would only start after all threads finish advancing the solution in time step $k-1$. Second, on-chip resources to cache the output of each time step are also required. Both features, chip-wide barriers and cache memory, exist on CPUs. Therefore PERKS could be effective on CPUs. However, in this paper we focus on demonstrating PERKS on GPUs since: a) GPUs are more challenging to program and

1 Tokyo Institute of Technology, Tokyo, Japan
2 RIKEN Center for Computational Science, Kobe, Japan
3 National Institute of Advanced Industrial Science and Technology, Tokyo, Japan
4 Shenzhen Institutes of Advanced Technology, Shenzhen, China
5 Oak Ridge National Laboratory, Oak Ridge, U.S.A.
a) zhang.l.ai@m.titech.ac.jp
b) mohamed.attia@riken.jp
c) chin.hou@aist.go.jp
d) jt.meng@siat.ac.cn
e) Wangx2@ornl.gov
f) endo@is.titech.ac.jp
g) matsu@acm.org

[*1] In this paper, we use PERKS, interchangeably, to refer to our proposed scheme and as an abbreviation of PERsistent KERnelS.

hence demonstrating the effectiveness of PERKS on GPUs would pave the way for PERKS on CPUs, and b) GPUs are increasingly prevalent in the top tier HPC systems; the Top500 [19] list includes seven GPU-accelerated systems in the top ten (June 2022 list), and one third of the systems on the list in general use discrete GPUs.

In PERKS, we first move the time loop of the solver to be inside the kernel and use a device-wide barrier at the end of each time step to avoid race conditions arising from possible neighborhood dependencies in the problem domain. Next, we identify the cachable data in the solver: the most considerable portion of data (arrays) that is the output of time step $k-1$ and input to time step k. Finally, we change the code to use as much is available from both the GPU scratchpad memory and registers to cache the data, and reduce the traffic to the device memory.

The basic idea of PERKS and implementing PERKS is relatively simple, which we argue is essential for encouraging scientists and engineers to adopt PERKS in their iterative solvers implemented for GPUs, and other architectures as well. That being said, a challenging aspect that we address in this paper is a detailed analysis of how and why PERKS is effective. The analysis requires an understanding of the effect of concurrency on performance. More particularly, to gain a deep understanding of why PERKS are effective and the limitations of architectural features, we study the effect of pressure on resources (particularly registers and shared memory). On top of that, we examine the effect of reducing the device occupancy while maintaining high enough concurrency to saturate the device.

It is important to note that PERKS is orthogonal to temporal blocking optimizations. Temporal blocking relies on combining multiple consecutive iterations of the time loop to reduce the memory transactions between them. The dependency along the time dimension is resolved by either: a) redundantly loading and computing cells from adjacent blocks, which limits the effectiveness of temporal blocking to low degrees of temporal blocking [20–22], or b) using tiling methods of complex geometry (e.g. trapezoidal and hexagonal tiling) along the time dimension and restricting the parallelism due to the dependency between neighboring blocks [23, 24]. In contrast, the execution scheme of PERKS does not necessitate the resolution of the dependency along the time dimension since PERKS includes an explicit barrier after each time step, which allows for advancing the boundary cells in time. This means the PERKS model can be generalized to any iterative solver, regardless of whether the solver had neighborhood dependencies in the domain or not, and can be used on top of any version of the solver. In other words, iterative kernels written as PERKS do not compete with optimized versions of those iterative kernels. For instance, a stencil PERKS does not compete with kernels applying aggressive locality optimizations; the performance gain from PERKS is added to the performance gain from whatever stencil optimizations are used in the kernel. **As a matter of fact, the more optimized the kernel before it is ported to the PERKS execution scheme, the higher the speedup that would be gained by PERKS. That is since optimizations to the kernel proportionally increase the overhead of data storing and loading in between iterations, which PERKS aims to reduce.**

The contributions in this paper are as follows:

- We introduce the design principles of PERKS, provide analyses of the potential of PERKS, and how to effectively port iterative solvers to PERKS.
- We implement a wide range of iterative 2D/3D stencil benchmarks and a conjugate gradient solver as PERKS in CUDA. It is important to note that iterative stencils and Krylov subspace solvers are the backbones of numerous scientific and engineering codes. We include an elaborate discussion on the implementation details and performance-limiting factors such as the domain sizes, concurrency, and resource contention.
- Our PERKS-based implementation achieves geometric means speedups of 2.35x for 2D stencils and 1.53x for 3D stencils using highly optimized baselines comparable to state-of-the-art 2D/3D stencil implementations, with A100 and V100. The source code of all PERKS-based implementations in this paper is available at the following anonymized link: http://shorturl.at/cdjmX.

2. Background and Motivation

2.1 Iterative algorithms

In iterative algorithms, the output of time step k is the input of time step $k+1$. Iterative methods can be expressed as:

$$x^{k+1} = F(x^k) \tag{1}$$

When the domain is mapped out to processing elements, there are two points to consider:

- Spatial dependency necessitates synchronization between time steps, or else advancing the solution in the following time step might use data that has not yet been updated in the previous time step.
- In time step $k+1$, each thread or thread block needs input from the output of itself in time step k (i.e. temporal dependency). This gives the opportunity for caching data between steps to reduce device memory traffic.

In the following sections, we briefly introduce iterative stencils and Krylov subspace methods. Throughout the paper, we use them as motivation examples, and we use them to report the effectiveness of our proposed methods, given their importance in HPC scientific and engineering codes.

2.1.1 Iterative Stencils

Iterative stencils are widely used in HPC. According to Bastian et al. [25], stencil applications represent **49**% of workloads in a wide range of HPC centers. Take 2D Jacobian 5-point stencil (2d5pt) as an example:

$$\begin{aligned} x(i,j)^{k+1} =& N * x(i,j+1)^k + S * x(i,j-1)^k + \\ & C * x(i,j)^k + W * x(i-1,j)^k + E * x(i+1,j)^k \end{aligned} \tag{2}$$

Computation of each point at time step $k+1$ requires the values of the point itself and its four neighboring points at time step k.

Two blocking methods are widely used to optimize iterative

stencils for data locality: *Spatial Blocking* [26, 27] and *Temporal Blocking* [3, 28].

In spatial blocking on GPUs, we split the domain into sub-domains, where each thread block can load its sub-domain to the shared memory to improve the data reuse. In the meantime, we require redundant data accesses at the boundary of the thread block to data designated for adjacent thread blocks.

In iterative stencils, each time step depends on the result of the previous time step. One could advance the solution by combining several time steps. The temporal dependency, in this case, is resolved by using a number of halo layers that match the number of combined steps. The amount of data that can be computed depends on the stencil radius (*rad*) and the number of time steps that are combined (b_t). In overlapped temporal tiling [29–31], this region can be represented as $2 \times b_t \times rad$ (*halo* region). Methods based on this kind of blocking are called *overlapped temporal blocking* schemes.

2.2 CUDA Programming Model

CUDA's programming model includes: *threads*, the basic execution unit (32 threads are executed together as a *warp*); *Thread block (TB)*, which is usually composed of hundreds of threads; *grid*, which is usually composed of tens of thread blocks.

On-chip memory in a streaming multiprocessor (SMX) includes: shared memory (scratchpad memory), L1 cache, and register file (RF) and. Off-chip memory includes global memory and L2 cache. Data in global memory can reside for the entirety of the program, while data in on-chip memory has the lifetime of a kernel. The shared memory is shared among all threads inside a thread block.

2.2.1 GPU Device-wide Synchronization:

Synchronization in GPUs was limited to groups of threads: thread blocks in CUDA (or a work group in OpenCL). Starting from CUDA 9.0, Nvidia introduced cooperative group APIs [32] that include an API for device-wide synchronization. Before introducing grid-level synchronization, the typical way to introduce device-wide synchronization was to launch sequences of kernels in a single CUDA stream. Zhang et al. [33] conducted a comprehensive study to compare the performance of both methods. The result shows that the latency difference between explicit device-wide synchronization versus implicit synchronization (via repetitive launching of kernels) is negligible in most kernels.

2.3 Motivational Example

We use a motivational example of a double precision 2D 9-point Jacobian stencil to motivate implementing iterative solvers as PERKS. (1). Why PERKS: Optimizations for iterative methods focus on a single step to speed up iterative solvers. Single-step optimizations move the performance of the kernel closer to the highest possible attainable performance on the roofline model, yet will not influence the operational intensity. As Figure 1 shows, optimizations used for the 2D 9-point stencil move the performance vertically at the same operational intensity value of the kernel. Temporal blocking schemes can move the operational intensity, horizontally, to the right side of the roofline, yet resolving the neighborhood dependencies introduces redundancy [22, 29, 30] or hard-to-parallelize complex geometrical tile shapes [24, 34], and can cause register pressure [3]. In PERKS,

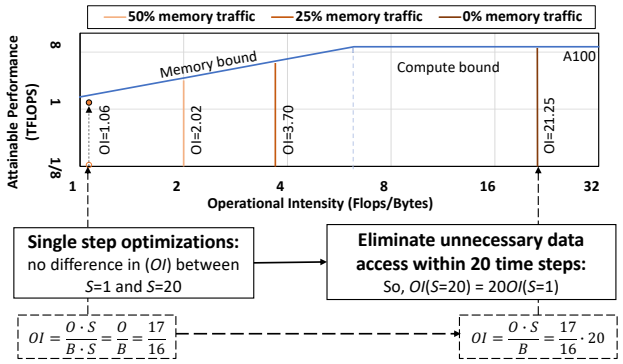


Fig. 1: The roofline model of a double precision 2D 9-point Jacobian stencil kernel, running $S = 20$ time steps, with a domain size of 3072^2 on A100 GPU. Optimization per-time step uses shared memory to improve locality [16]. Optimizations only improve the iterative stencil kernel to get closer to the peak performance. Reducing memory traffic between time steps can increase the performance by increasing the operation intensity (OI). We plot different operational intensities for a version of PERKS that reduces the data traffic in-between 20 time steps to 50%, 25%, and 0%.

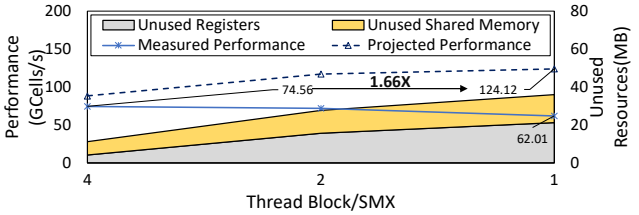


Fig. 2: Performance of a double precision 2D 9-point Jacobian stencil kernel (3072^2) for different thread blocks per streaming multiprocessor (TB/SMX) on A100. Filled regions indicate unused resources. The projected performance assumes that all unused resources can be used to cache data. Using one TB/SMX and using all unused resources for caching can theoretically provide 1.66x speedup in this situation.

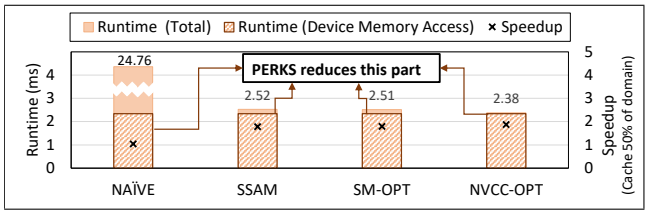


Fig. 3: Runtime (20 time steps) of double precision 2D 9-point Jacobian stencil (3072^2) with different state-of-the-art optimizations on A100 GPU. PERKS aims to reduce/eliminate the traffic to/from device memory in-between time steps. NVCC-OPT improves on NAIVE by enabling the auto-unrolling optimization provided by the latest NVCC compiler version. SSAM [4] uses register, while SM-OPT uses shared memory to improve locality [16]. We also plot the speedup of each implementation, assuming we cache 50% of the domain. The results show that the more optimized the baseline kernel, the more performance improvement we expect from caching.

we batch a sequence of time steps together and remove the unnecessary data access between time steps. The target data traffic to reduce is in-between time steps (i.e., outside the solver), and hence is not subject to the neighborhood dependency issue in temporal blocking schemes. Figure 1 demonstrates how this idea works for a real stencil benchmark running on an A100 GPU for 20 time steps. By caching more of the domain in-between time steps, the operational intensity moves more to the right side of the roofline to be compute-bound. This also demonstrates how PERKS is orthogonal to the per-time step optimizations; PERKS would improve the performance (by moving horizontally on the roofline) regardless of how optimized the baseline algorithm is at its operational intensity. (2). The prospect of PERKS: Latency across all operations/instructions in newer generation GPUs has been significantly dropping [35]. As a result, often fewer numbers of warps are enough for CUDA runtime to hide the latency effectively and hence maintain high performance at low occupancy [36]. In Figure 2, we vary the number of thread blocks per streaming multiprocessor (TB/SMX) and plot its performance (left Y-axis). For each TB/SMX configuration, we plot on the right Y-axis the unused resources (shared memory and registers). As the figure shows, even when $TB/SMX = 4$, more than 11.2MB of shared memory and register files are not in use. When TB/SMX decreases, the performance is slightly fluctuated (74.6-62.0 GCells/s [*2]) while the freed shared memory and registers gradually increase. By reducing the TB/SMX to its minimum while maintaining enough concurrency to sustain the performance level, the projection from performance gain when caching a subset of the results in unused resources can improve the performance by more than 1.6x.

As Figure 3 shows, the amount of time required for moving the data from/to device memory in-between time steps for a stencil kernel remains constant. At the same time, the compute part decreases as the more optimized the stencil implementation is. The prospect of PERKS is to reduce/eliminate this data movement time that dominates the runtime in highly optimized stencil implementations. Finally, while temporal-blocking schemes do also reduce the data movement to some extent, they can not be generalized to all iterative solvers. Additionally, resolving the temporal and spatial dependency adds compute overhead and can also lead to increased register pressure that limits the degree of temporal blocking on GPUs [3].

3. PERKS: Persistent Kernels to Improve Locality

3.1 Overview of PERKS

PERsistent Kernels (PERKS) is a generic scheme for running iterative solvers to improve data locality by taking advantage of the large capacity of on-chip resources. PERKS relies on chip-wide synchronization and on-chip caching resources: user-managed (e.g., scratchpad memory) or transparent (e.g., cache memory). Nvidia GPUs are widely used accelerators in HPC systems (more than 30% of systems in Top500 [19]), and they are equipped with both features. Nonetheless, PERKS, in principle, can be applied on any processor with support for chip-wide synchronization and on-chip caching resources, be that a CPU or a discrete accelerator.

[*2] GCells/s denotes giga-cells updated per second.

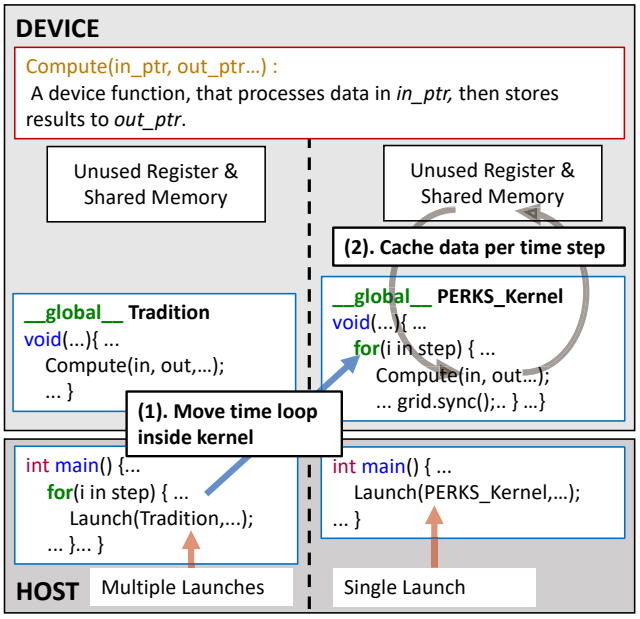


Fig. 4: Changing a traditional iterative CUDA kernel (time loop on the host) to PERKS: 1) move the time loop from the host code to the kernel code and use grid synchronization between time steps. 2) cache data between time loops on the unused shared memory and register files. The compute portion of the kernels does not notably change, i.e., no requirement to change the original algorithm when using PERKS.

Figure 4 shows an example of applying PERKS with CUDA. As Figure 4 illustrates, in PERKS, we move the time stepping loop from the host to the device, and use CUDA's grid synchronization API as a device-wide barrier at each time step. We then use the free register files and shared memory to reduce traffic to/from the device memory by caching the domain (or a subset of it) in-between time steps.

3.2 Assumptions and Limitations

The techniques discussed in this paper are based on the following assumptions about the applications.

Target Applications: In this paper, we target iterative kernels that are bounded by memory bandwidth. While execution in a PERKS fashion makes no assumptions on the underlying implementation, optimal PERKS performance can sometimes require minor adaptations to the kernel. Finally, despite not reporting results for compute-bound iterative kernels, it is important to note that compute-bound iterative kernels could potentially also benefit from becoming PERKS, if the kernel generates memory traffic in-between iterations that CUDA runtime can not effectively overlap with computation.

Impact on Optimized Kernels: It is crucial to note that PERKS is orthogonal to the optimization level applied to the compute part of the kernel. As a matter of fact, the more optimized the baseline kernel, the more performance improvement we expect from PERKS. Because optimizations reduce the time

per iteration, i.e., a single kernel invocation, while the time to store/load to global memory in-between iterations remains the same. *To summarize, PERKS reduces what amounts to be Amdahl's law serial portion of the solver, and hence the more optimized a code, i.e., a faster parallel portion, the higher the speedup that would be attributed to PERKS.*

PERKS in Distributed Computing: PERKS in this paper is demonstrated on a single GPU. In distributed applications that require halo regions (e.g., stencils), PERKS can potentially be used on top of communication/computation overlapping schemes [37, 38]. In overlapping schemes, the boundary points that are computed in a separate kernel would not be cached, while the kernel of the interior points would run as PERKS to cache the data of the interior points. PERKS could also be used with communication-avoiding algorithms (e.g., communication-avoiding Krylov methods [39])

Use of Registers: PERKS uses registers and shared memory for caching data in-between time steps. It should be noted that there are no guarantees that the compiler releases all the registers after the compute portion in each iteration is finished (with Nvidia's nvcc compiler we did not observe such inefficiency). If such register reuse inefficiency exists, imperfect register reuse by the compiler could result in fewer registers being available for caching and leaves only shared memory to be used for caching. PERKS would not be effective if the target kernel consumes all on-chip resources (both register file and shared memory) even in its minimal occupancy.

Iterative Solvers as PERKS: While this paper's focus is to demonstrate PERKS model for iterative stencils and Krylov subspace methods (conjugate gradient), the discussion in this section (and paper in general) is applicable to a high degree for other types of iterative solvers. That is since PERKS is not much concerned with the implementation of the solver, and only loads/stores the domain (or a subset of it) before/after the solver part in the kernel, under resource constraints. Iterative solvers that use the same flow expressed in Figure 4 can, in principle, be ported to PERKS (with relative ease). Generally speaking, the porting process is as follows: move the time step outside the kernel to be inside the kernel, add grid synchronization to ensure dependency, and store/load a portion of the input or output to cache: either shared memory and/or register (using register arrays). More details on porting kernels to PERKS in Section 4.1.

4. Porting Solvers to PERKS

Transforming the existing iterative solvers to PERKS is fairly straightforward. This section first explains briefly how end-users can transform or port their iterative solvers to PERKS. Next, we elaborate on how we implemented memory-bound iterative methods (namely 2D/3D stencils and a conjugate gradient solver) as PERKS.

4.1 Transforming Kernels to PERKS: the End-user Perspective

4.1.1 Identifying the minimal concurrency of the kernel

The end-user only needs to reduce the device occupancy to minimum **(while maintaining performance)** via manual tuning of the kernel launch parameters or using auto-tuning tools [40–42].

4.1.2 Porting of Kernel to become PERKS

As Listing 1 shows, PERKS does not modify computation, and the manually written code to move the time loop inside the kernel and load/store to cache is straightforward. Alternatively, though outside this paper's scope, we point out the possibility of simplifying the process of converting a kernel to PERKS by using source-to-source translation, C++ templates, or Domain Specific Languages.

4.1.3 What to Cache

The end-user can use a profiler, offline, to decide on what data arrays to cache by identifying the arrays that generate the most traffic to/from global memory. In many iterative solvers, profiling is not even needed since the algorithm clearly implies the main data array(s) causing the highest traffic (e.g., the matrix A in conjugate gradient and the domain in stencil applications).

4.1.4 Where to Cache

The end-user would simply use the unused shared memory for caching. For additional performance benefits, advanced users can choose to also cache in registers by manually identifying the adequate number of registers that can be used for caching, without causing register spilling (we provide a python script to automate this process), or by following the trace of existing on-chip resources management research [43, 44]. We anticipate the possibility of automating this step by source-to-source translation or Domain Specific Languages so that this step of using on-chip resources could be as easy as adding a persisting range in the domain, similar, in principle, to the method of using L2 cache residency control in A100 [45].

4.2 Transforming Stencil Kernels to PERKS

Our 3D stencil implementation uses the standard shared memory implementation where 2D planes (1D planes in 2D stencils) are loaded one after the other in shared memory. Each thread computes the cells in a vertical direction [2, 46]. In our PERKS implementation, before the compute starts, planes that already have the data cached from the previous time step do not load from global memory. We do not interfere with compute; only after the compute is finished that we store the results in the registers/shared memory. As Listing 1 shows, after adjusting to handle the input and output of the computation part of the kernel. To ensure coalesced memory accesses in the halo region, we transpose the vertical edges of the halo region in global memory. Finally, if the original kernel uses shared memory [2, 46] or registers [4] to optimize stencils, we use the version of the output residing in shared memory or registers at the end of each time step as an already cached output. This way, we avoid an unnecessary copy to shared memory and registers we would use for caching.

5. Why PERKS is Effective

This section includes an analysis of the effectiveness of PERKS in a practical setting. The analysis in this section is based on the latest Nvidia GPUs. Nonetheless, the analysis can be expanded to other architectures with relative ease. We explain how to effectively reduce concurrency in a regression-free manner to improve

Listing 1: 2D 5-pt stencil implemented in PERKS.

```
__global__ void 2d5pt_PERKS(ptr_in, ptr_out){...
for(k=0; k<timestep; k++){...
  switch (Source(ptr_in)){
  case FromSM:  load(sm_cache,sm_in);      break;
  case FromReg: load(reg_cache,sm_in);     break;
  default:      load(ptr_in,sm_in);}
  2d5pt_Compute(sm_in,reg_out);
  switch (Destination(ptr_out)){
  case ToSM:    store(reg_out,sm_cache);   break;
  case ToReg:   store(reg_out,reg_cache);  break;
  default:      store(reg_out,ptr_out);}
  ...
  //resolve dependency in halo region of TBs
  //part of the baseline code; omitted for space
  grid.sync();
  ...}
...}

__device__ void 2d5pt_Compute(sm_in, reg_out){
  x = threadIdx.x;
  t[IPT+2]; //IPT: items per thread
  for(y=0; y< IPT+2; y++)
    t[y]=sm_in[x, y+ind_y-1];
    for( y=0; y< IPT; y++){
      reg_out[y]=sm_in[x+ind_x-1,y+1+ind_y]*WEST
        +sm_in[x+ind_x+1,y+1+ind_y]*EAST
        +t[y-1+1]*SOUTH
        +t[y+1]*CENTER
        +t[y+1+1]*NORTH;
    }
}
```

the performance of PERKS.

In PERKS, low occupancy is desirable since this releases on-chip resources (scratchpad memory and registers) to be used for caching more data. Yet low occupancy should not be done in a manner that drops the performance. Volkov [36] explored methods to achieve high performance with low occupancy. Specifically, Volkov reported that performance is not only dictated by Thread Level Parallelism (TLP). We use $\mathbb{C}_{sw}(\mathbb{OP})$ to represent the parallelism exposed by the kernel where $\mathbb{C}_{sw}(\mathbb{OP})$ is the minimum number of concurrently executable instructions of the operation $\mathbb{OP}$ exposed by the launched kernel.

The kernel saturates the device only when the minimal concurrency exposed by the kernel is higher than the max concurrency supported by the hardware ($\mathbb{C}_{hw}$).

The hardware concurrency is dictated by $\mathbb{C}_{hw}$ by throughput $\mathbb{THR}$ and latency $\mathbb{L}$ [36], according to Little's Law [47]:

$$\mathbb{C}_{hw} = \mathbb{THR} \cdot \mathbb{L} \tag{3}$$

The throughput $\mathbb{THR}$ for different data access operations are available in the official documentation of Nvidia GPUs [48, 49]. We measure the latency $\mathbb{L}$ with commonly used microbenchmarks [50–52].

We only summarize the concurrency findings. For the global memory access operations at the SMX level, relying only on TLP requires minimal occupancy (for $\mathbb{C}_{sw} \geq \mathbb{C}_{hw}$) of 31.25% (P100), 25% (V100), and 37.5% (A100) to fully saturate the memory bandwidth. Reducing the number of launched threads to meet this minimal occupancy releases the on-chip resources to be used for caching in PERKS. In addition, it is worthwhile to mention that many well-tuned kernels usually rely more on Instruction Level Parallelism (ILP) to drive the concurrency, which means even lower occupancy can be tolerated [53–56].

Table 1: Stencil benchmarks. A detailed description of the stencil benchmarks can be found in [17, 28]

Benchmark(Stencil Order, FLOPs/Cell)			
2d5pt(1,10)	2ds9pt(2,18)	2d13pt(3,26)	2d17pt(4,34)
2d21pt(5,42)	2ds25pt(6,59)	2d9pt(1,18)	2d25pt(2,50)
3d7pt(1,14)	3d13pt(2,26)	3d17pt(1,34)	3d27pt(1,54)
poisson(1,38)	—	—	—

6. Evaluation

6.1 Hardware and Software Setup

The experimental results presented here are evaluated on the two latest generations of Nvidia GPUs: Volta V100 and Ampere A100 with CUDA 11.5 and driver version 495.29.05.

We run each evaluation ten times for all iterative stencils and conjugate gradient experiments, and report the run with the highest performance.

6.2 Benchmarks and Datasets

6.2.1 Stencil Benchmarks

To evaluate the performance of PERKS-based stencils, we conducted a wide set of experiments on various 2D/3D stencil benchmarks (listed in Table 1). The baseline implementation uses state-of-the-art optimizations such as shared memory and heavy unrolling (to counter the reduction in over-subscription). We report the performance (CGells/s) of the baseline implementation for all benchmarks in Table 2. The baseline performance is on-par with (and often exceeds) state-of-the-art GPU-optimized stencil codes reporting the highest performance across different stencil benchmarks. Namely, SSAM [4], register-optimized stencils [57, 58], StencilGen [59], and temporal blocking AN5D [3].

We use the test data provided by StencilGen [59]. We tested three PERKS implementations: PERKS (sm) that only uses shared memory to cache data; PERKS (reg) that only uses register to cache data; and PERKS (mix) that uses both shared memory and registers to cache data. Due to space limitations, we report only the peak performance among those three PERKS variants.

6.3 Sizes of Domains and Problems

PERKS intuitively favors small domain/problem sizes. However, for a fair evaluation of PERKS, we can not choose arbitrarily small domain sizes; we need domain/input sizes that fully utilize the compute capability of the device. We conducted an elaborate set of experiments for every individual stencil benchmark to identify the minimum domain size that would fully utilize the device. We use this domain size to represent large domains when PERKS can only partially cache the domain. Note that domain/problem sizes that are beyond domain/problem sizes that could fully utilize the device are effectively serialized by the device once we go beyond peak concurrency sustainable by the device. Table 2 summarizes the domain sizes (marked as 'P') for stencil benchmarks that would provide a base for a fair comparison. We also test small domains where the whole domain can be cached by PERKS (marked 'F' in Table 2).

Table 2: Speedup of PERKS over baseline (non-persistent kernels) for 2D/3D stencil benchmarks on A100 and V100 GPUs. For benchmarks two problem sizes are reported: 1) for domain sizes large enough to saturate the device (i.e. weak scaling simulations), and 2) for domain sizes that can be Full in PERKS (i.e. strong scaling simulations); Label Explanation: *BM* = Benchmarks, *$* = Cache, *P* = Partially Cache, *F* = Fully Cache, *% $ed* = Percentage Cached, *Perf.* = Performance in Giga-cells Updated per Second, ↑ = Speedup, GM: Geometric Mean.

BM	$	Single Precision								Double Precision							
		A100				V100				A100				V100			
		Domain Size	% $ed	Perf.	↑	Domain Size	% $ed	Perf.	↑	Domain Size	% $ed	Perf.	↑	Domain Size	% $ed	Perf.	↑
2d5pt	P	4608 × 3072	59%	257.74	**1.66**	4096 × 2560	38%	137.30	**1.49**	2304 × 2304	88%	285.63	**3.69**	2048 × 1280	88%	241.63	**5.45**
	F	3072 × 2160	100%	636.79	**4.34**	2560 × 1536	100%	494.64	**5.56**	2304 × 1536	100%	335.42	**4.46**	2048 × 1120	100%	277.08	**6.44**
2ds9pt	P	4608 × 3072	64%	229.88	**1.51**	2560 × 2048	94%	301.32	**3.48**	2304 × 2304	79%	194.48	**2.70**	2048 × 1280	75%	108.49	**2.51**
	F	4608 × 1824	100%	463.25	**3.59**	2048 × 1760	100%	352.98	**4.20**	2304 × 1152	100%	235.70	**3.20**	1536 × 1280	100%	188.32	**4.39**
2d13pt	P	4608 × 3072	47%	197.83	**1.35**	2560 × 2048	94%	222.71	**2.64**	4608 × 3072	13%	84.40	**1.10**	2048 × 2048	47%	58.99	**1.40**
	F	4608 × 1440	100%	357.40	**2.86**	2560 × 1408	100%	266.54	**3.25**	4608 × 576	100%	181.19	**2.71**	2048 × 800	100%	141.26	**3.62**
2d17pt	P	4608 × 3072	50%	170.91	**1.28**	5120 × 4096	14%	99.45	**1.16**	3072 × 2304	47%	91.77	**1.34**	4096 × 2560	14%	47.52	**1.18**
	F	3072 × 2448	100%	280.19	**2.63**	4096 × 800	100%	207.62	**2.76**	2304 × 1440	100%	145.53	**2.30**	2560 × 576	100%	109.57	**3.18**
2d21pt	P	4608 × 3072	41%	151.78	**1.21**	2560 × 2048	75%	124.12	**1.60**	4608 × 3072	6%	73.44	**1.13**	5120 × 4096	0%	46.76	**1.12**
	F	4608 × 1536	100%	233.71	**2.10**	2560 × 1408	100%	178.40	**2.39**	3072 × 1008	100.00%	118.98	**2.29**	4096 × 320	100%	89.87	**2.63**
2ds25pt	P	4608 × 4608	13%	126.98	**1.05**	2048 × 2048	78%	105.91	**1.41**	4608 × 4608	2%	69.29	**1.22**	5120 × 4096	0%	46.79	1.21
	F	4608 × 672	100%	192.49	**1.81**	2048 × 1600	100%	149.50	**2.06**	4608 × 576	100%	96.98	**2.00**	4096 × 280	100.00%	61.96	2.13
2d9pt	P	3072 × 3072	98%	491.51	**3.44**	2560 × 2048	97%	388.20	**4.30**	2304 × 2304	83%	247.35	**3.27**	2048 × 1280	88%	189.82	**4.27**
	F	3072 × 2592	100%	547.10	**4.03**	2560 × 1664	100%	408.36	**4.61**	2304 × 1920	100%	289.90	**3.89**	1792 × 1280	100%	227.56	**5.15**
2d25pt	P	4608 × 3072	47%	187.21	**1.26**	2560 × 2048	91%	196.04	**2.29**	4608 × 3072	13%	82.56	**1.13**	2048 × 1280	63%	67.39	**1.58**
	F	4608 × 1536	100%	274.74	**2.04**	2560 × 1408	100%	212.94	**2.56**	4608 × 816	100%	139.38	**2.28**	2048 × 720	100%	105.13	**2.64**
GM(2D)	-	-	-	-	**2.04**	-	-	-	**2.60**	-	-	-	**2.19**	-	-	-	**2.63**
3d7pt	P	256 × 288 × 256	21%	159.95	**1.19**	256 × 160 × 256	27%	99.49	**1.21**	256 × 288 × 256	0%	77.58	**1.08**	128 × 128 × 128	47%	56.47	**1.67**
	F	126 × 192 × 256	100%	274.37	**2.23**	68 × 160 × 256	100%	212.58	**3.21**	39 × 288 × 256	100%	146.25	**2.24**	80 × 128 × 128	100%	98.33	**2.99**
3d13pt	P	256 × 288 × 256	0%	129.16	**1.12**	256 × 320 × 256	0%	86.94	**1.06**	256 × 288 × 256	2%	70.15	**1.18**	256 × 320 × 256	0%	41.04	**1.03**
	F	81 × 288 × 256	100%	152.15	**1.47**	32 × 320 × 256	100%	137.67	**2.46**	33 × 288 × 256	100%	92.62	**1.77**	8 × 320 × 256	100%	46.93	**1.90**
3d17pt	P	256 × 288 × 256	28%	138.53	**1.07**	160 × 160 × 256	45%	94.45	**1.34**	256 × 288 × 256	2%	75.16	**1.13**	160 × 160 × 256	13%	45.54	**1.40**
	F	84 × 288 × 256	100%	170.66	**1.51**	160 × 64 × 256	100%	131.01	**2.12**	36 × 288 × 256	100%	87.97	**1.53**	32 × 160 × 256	100%	64.43	**2.37**
3d27pt	P	256 × 288 × 256	14%	130.88	**1.01**	160 × 160 × 256	45%	94.32	**1.34**	256 × 288 × 256	5%	75.22	**1.14**	160 × 160 × 256	13%	45.69	**1.40**
	F	81 × 192 × 256	100%	162.22	**1.33**	160 × 64 × 256	100%	130.99	**2.12**	33 × 288 × 256	100%	86.89	**1.43**	32 × 160 × 256	100%	64.01	**2.35**
Poisson	P	256 × 288 × 256	14%	130.50	**1.01**	160 × 160 × 256	30%	94.70	**1.35**	256 × 288 × 256	2%	73.13	**1.12**	160 × 160 × 256	13%	45.78	**1.41**
	F	90 × 288 × 256	100%	164.00	**1.45**	160 × 64 × 256	100%	130.89	**2.12**	36 × 288 × 256	100%	87.59	**1.54**	32 × 160 × 256	100%	64.92	**2.39**
GM(3D)	-	-	-	-	**1.30**	-	-	-	**1.72**	-	-	-	**1.38**	-	-	-	**1.80**

6.4 Iterative 2D/3D Stencils

Table 2 shows the PERKS' speedups for both large domain and small domain sizes. The geometric mean speedup for 2D stencils is 2.11x in A100 and 2.61x in V100. The geometric mean speedup for 3D stencils is 1.34x for A100 and 1.76x for V100. It is worth reemphasizing that PERKS' speedups should not be compared to speedups from optimization applied to the baselines; PERKS' speedups are compounded over any speedups for optimizations applied to the baselines.

In large problem sizes, it is important to note three points: a) the benchmarks we use include both low-order and high-order stencils, b) the speedups are particularly higher on low-order stencils that are more commonly used in practice (ex: geomean of 1.57x speedup for up to 2^{nd} order 3D stencils on V100 and A100 in double precision), and c) the speedups we report are not limited to the –highly optimized– implementation we use as baseline; other stencil implementations, regardless of their internals, can also benefit from being transformed to PERKS.

6.5 Where to Cache: Shared Mem., Registers, or Both?

The intuition is that using both shared memory and registers would always be better (more cache-able space). The results show that this is usually the case. There can, however, be exceptions. For instance, in our observations, we see that for higher order stencils, using shared memory and registers is often not the ideal choice (presumably due to arising register pressure).

6.6 Discussion of the Results

We want to emphasize that for large problem sizes, PERKS achieves high performance. To illustrate it, we can see from Figure 5 and Figure 6, that by applying PERKS in V100, we get a geometric mean speedup of 1.71x, which is 98.6% of what one generation of hardware improvements in A100 provide (1.72x) . **Applying PERKS provides a performance gain comparable to migrating to the next generation hardware.**

7. Related Work

The concept of persistent threads and persistent kernels dates back to the introduction of CUDA. The main motivation for persistence at the time was load imbalance issues with the runtime warp scheduler [60, 61]. Later research focused on using persistent kernels to overcome the kernel invocation overhead (which was high at the time). GPUrdma [62] and GPU-Ether [63] expanded on the concept of persistent kernels to reduce the latency of network communication. As on-chip memory sizes increased, researchers began to capitalize on data reuse in persistent kernel. Most of them focused on specific applications, GPUrdma [62] proposed to keep the constant matrix in shared memory. Khorasani et al. [64] proposed to keep parameters in register. Zhu et al. [65] proposed a sparse persistent implementation of recurrent neural networks. To our knowledge, this work is the first to propose a methodological and generic blueprint for accelerating memory-bound iterative applications using persistent kernels.

8. Conclusion

We propose a persistent kernel execution scheme for iterative applications. We enhance performance by moving the time loop to the kernel and caching the intermediate output of each time step. We show notable performance improvement for iterative 2D/3D stencils for both V100 and A100 over highly optimized baselines. We further report notably high speedups in small domain/problem sizes, which is beneficial in strong scaling cases.

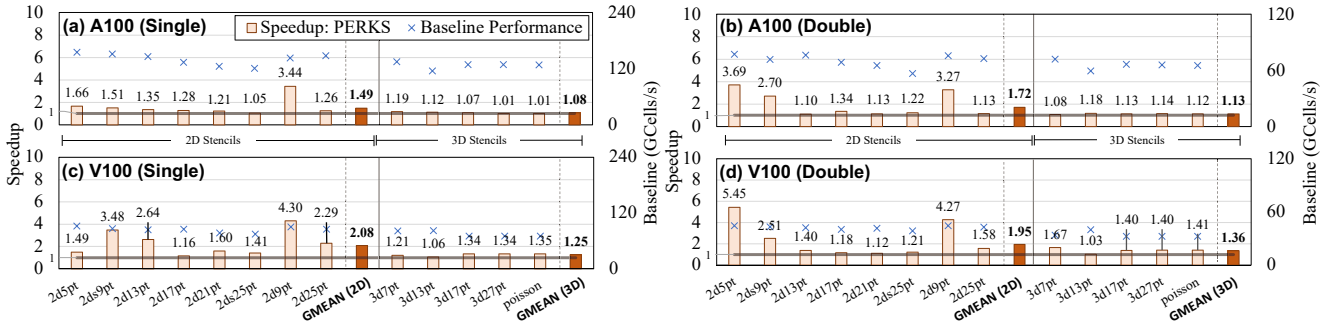


Fig. 5: PERKS speedup for 2D/3D stencil benchmarks on A100 and V100 GPUs. Baseline in this figure (and Figure 6) uses the widely-common stencil optimization employing shared memory to improve locality [2, 16]. Note that using higher performing stencil implementations as baseline would further improve PERKS speedup since the overhead of data storing/loading in between iterations proportionally increases when the implementation is more optimized (as explained in Figure 3)

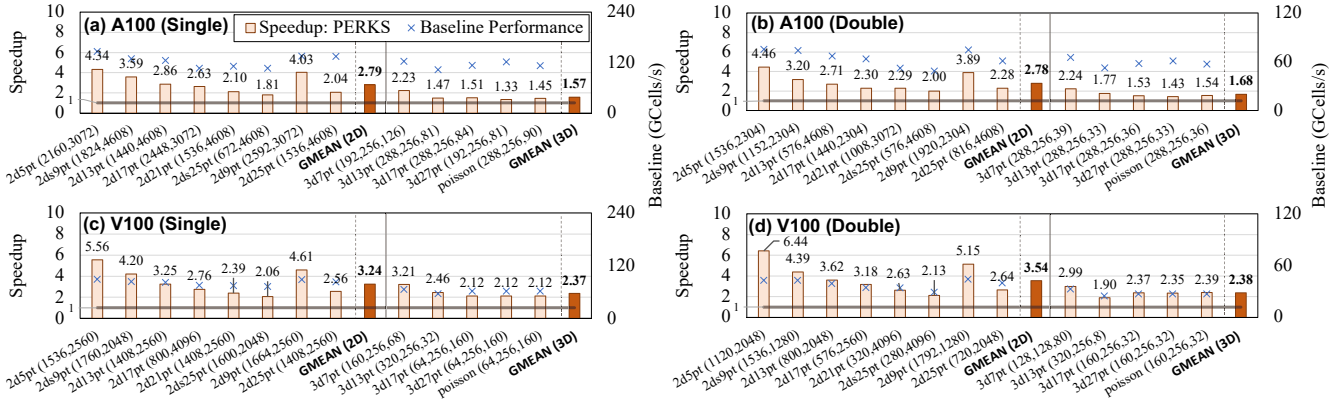
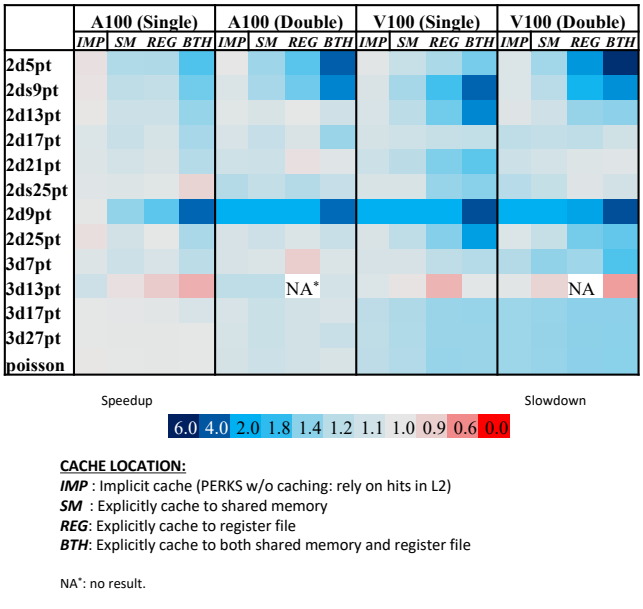


Fig. 6: Performance of PERKS for 2D stencils with small domain sizes on A100 and V100 GPUs (domain sizes are written next to the benchmark names). Small domain means that the whole domain can be cached (ex: in strong scaling cases).



CACHE LOCATION:
IMP : Implicit cache (PERKS w/o caching: rely on hits in L2)
SM : Explicitly cache to shared memory
REG: Explicitly cache to register file
BTH: Explicitly cache to both shared memory and register file

NA*: no result.

Fig. 7: Heatmap of speedup over non-PERKS baseline (SM-OPT) when caching data in different locations, for different stencils.

References

[1] J. Meng and K. Skadron, "A performance study for iterative stencil loops on gpus with ghost zone optimizations," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, 2011.

[2] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "On optimizing complex stencils on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 641–652, 2019.

[3] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "AN5D: automated stencil framework for high-degree temporal blocking on gpus," in *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*, pp. 199–211, 2020.

[4] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for gpu memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–81, 2019.

[5] A.-K. C. Ahamed and F. Magoulès, "Efficient implementation of jacobi iterative method for large sparse linear systems on graphic processing units," *The Journal of Supercomputing*, vol. 73, no. 8, pp. 3411–3432, 2017.

[6] A. Kochurov and D. Golovashkin, "Gpu implementation of jacobi method and gauss-seidel method for data arrays that exceed gpu-dedicated memory size," *Journal of Mathematical Modelling and Algorithms in Operations Research*, vol. 14, no. 4, pp. 393–405, 2015.

[7] H. Courtecuisse and J. Allard, "Parallel dense gauss-seidel algorithm on many-core processors," in *2009 11th IEEE International Conference on High Performance Computing and Communications*, pp. 139–147, IEEE, 2009.

[8] M. Fratarcangeli, V. Tibaldo, and F. Pellacini, "Vivace: A practical gauss-seidel method for stable soft body dynamics," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, pp. 1–9, 2016.

[9] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pp. 68–84, Springer, 2014.

[10] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A modern linear operator algebra framework for high performance computing," 2020.

[11] J. I. Aliaga, J. Pérez, and E. S. Quintana-Ortí, "Systematic fusion of cuda kernels for iterative sparse linear system solvers," in *European Conference on Parallel Processing*, pp. 675–686, Springer, 2015.
[12] R. Couturier and S. Domas, "Sparse systems solving on gpus with gmres," *The journal of Supercomputing*, vol. 59, no. 3, pp. 1504–1516, 2012.
[13] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pp. 463–482, 2011.
[14] J. Lifflander and S. Krishnamoorthy, "Cache locality optimization for recursive programs," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–16, 2017.
[15] L. Yuan, Y. Zhang, P. Guo, and S. Huang, "Tessellating stencils," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, (New York, NY, USA), Association for Computing Machinery, 2017.
[16] N. Maruyama and T. Aoki, "Optimizing Stencil Computations for NVIDIA Kepler GPUs," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations* (A. Größlinger and H. Köstler, eds.), (Vienna, Austria), pp. 89–95, Jan. 2014.
[17] 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*, pp. 1–44, 2019.
[18] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.
[19] "Top500," 2022. [Online; accessed 27-Mar-2021].
[20] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), p. 311–320, Association for Computing Machinery, 2012.
[21] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), p. 256–265, Association for Computing Machinery, 2009.
[22] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Sdslc: A multi-target domain-specific compiler for stencil computations," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, (New York, NY, USA), Association for Computing Machinery, 2015.
[23] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1285–1298, 2017.
[24] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for gpus: Automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, (New York, NY, USA), p. 24–31, Association for Computing Machinery, 2013.
[25] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 100–112, 2018.
[26] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, (New York, NY, USA), pp. 319–329, ACM, 1988.
[27] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, (New York, NY, USA), pp. 655–664, ACM, 1989.
[28] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan, "Effective resource management for enhancing performance of 2d and 3d stencils on gpus," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pp. 92–102, 2016.
[29] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 311–320, ACM, 2012.
[30] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 235–244, ACM, 2007.
[31] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 256–265, ACM, 2009.
[32] Nvidia, "Nvidia cuda runtime api," 2021.
[33] L. Zhang, M. Wahib, H. Zhang, and S. Matsuoka, "A study of single and multi-device synchronization methods in nvidia gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 483–493, IEEE, 2020.
[34] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, pp. 1285–1298, May 2017.
[35] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018.
[36] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10, p. 16, San Jose, CA, 2010.
[37] T. Endo, Y. Takasaki, and S. Matsuoka, "Realizing extremely large-scale stencil applications on gpu supercomputers," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 625–632, 2015.
[38] C. Pearson, M. Hidayetoğlu, M. Almasri, O. Anjum, I.-H. Chung, J. Xiong, and W.-M. W. Hwu, "Node-aware stencil communication for heterogeneous supercomputers," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 796–805, 2020.
[39] Y. Idomura, T. Ina, Y. Ali, and T. Imamura, "Acceleration of fusion plasma turbulence simulations using the mixed-precision communication-avoiding krylov method," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, p. 93, 2020.
[40] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code autotuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019.
[41] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.
[42] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
[43] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in gpus," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, IEEE, 2016.
[44] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into gpu on-chip memory resources," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 23–33, IEEE, 2015.
[45] Nvidia, "Nvidia a100 tensor core gpu architecture," 2021.
[46] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, (New York, NY, USA), p. 79–84, Association for Computing Machinery, 2009.
[47] J. D. C. Little, "A proof for the queuing formula: L = λw," *Operations Research*, vol. 9, pp. 383–387, 1961.
[48] Nvidia, "Programming guide," 2022.
[49] N. CUDA, "Nvidia a100 tensor core gpu architecture," 2021. [Online; accessed 20-July-2021].
[50] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246, IEEE, 2010.
[51] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern gpus," in *IFIP International Conference on Network and Parallel Computing*, pp. 144–156, Springer, 2014.
[52] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," in *ACM SIGPLAN Notices*, vol. 52, pp. 31–43, ACM, 2017.
[53] M. Kronbichler and K. Ljungkvist, "Multigrid for matrix-free high-order finite element computations on graphics processors," *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 1, pp. 1–32, 2019.
[54] O. Zachariadis, A. Teatini, N. Satpute, J. Gómez-Luna, O. Mutlu, O. J. Elle, and J. Olivares, "Accelerating b-spline interpolation on gpus:

Application to medical image registration," *Computer methods and programs in biomedicine*, vol. 193, p. 105431, 2020.

[55] K. Świrydowicz, N. Chalmers, A. Karakus, and T. Warburton, "Acceleration of tensor-product operations for high-order finite element methods," *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 735–757, 2019.

[56] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 678–689, IEEE, 2016.

[57] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Register optimizations for stencils on gpus," *SIGPLAN Not.*, vol. 53, pp. 168–182, Feb. 2018.

[58] P. S. Rawat, A. Sukumaran-Rajam, A. Rountev, F. Rastello, L.-N. Pouchet, and P. Sadayappan, "Associative instruction reordering to alleviate register pressure," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, (Piscataway, NJ, USA), pp. 46:1–46:13, IEEE Press, 2018.

[59] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance gpu code for stencil computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.

[60] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the conference on high performance graphics 2009*, pp. 145–149, 2009.

[61] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-gpu systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, 2010.

[62] F. Daoud, A. Watad, and M. Silberstein, "Gpurdma: Gpu-side library for high performance networking from gpu kernels," in *Proceedings of the 6th international Workshop on Runtime and Operating Systems for Supercomputers*, pp. 1–8, 2016.

[63] C. Jung, S. Kim, I. Yeom, H. Woo, and Y. Kim, "Gpu-ether: Gpu-native packet i/o for gpu applications on commodity ethernet," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pp. 1–10, IEEE, 2021.

[64] F. Khorasani, H. Asghari Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-register parameter caching for dynamic neural nets with virtual persistent processor specialization," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 377–389, 2018.

[65] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse persistent rnns: Squeezing large recurrent networks on-chip," *arXiv preprint arXiv:1804.10223*, 2018.