

Porting Batched Iterative Solvers onto Intel GPUs with SYCL

Phuong Nguyen
phuong.nguyen@icl.utk.edu
University of Tennessee, Knoxville
USA

Pratik Nayak
nayak@kit.edu
Karlsruhe Institute of Technology
Germany

Hartwig Anzt
hantz@icl.utk.edu
University of Tennessee, Knoxville
USA

ABSTRACT

Batched linear solvers play a vital role in computational sciences, especially in the fields of plasma physics and combustion simulations. With the imminent deployment of the Aurora Supercomputer and other upcoming systems equipped with Intel GPUs, there is a compelling demand to expand the capabilities of these solvers for Intel GPU architectures.

In this paper, we present our efforts in porting and optimizing the batched iterative solvers on Intel GPUs using the SYCL programming model. These new solvers achieve impressive performance on the Intel GPU Max 1550s (Ponte Vecchio GPUs) which surpass our previous CUDA implementation on NVIDIA H100 GPUs by an average of 2.4x for the PeleLM application inputs. The batched solvers are ready for production use in real-world scientific applications through the Ginkgo library, complementing the performance portability of the batched functionality of Ginkgo.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**; • **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

SYCL, Performance Portability, Batched Linear Solvers, Intel GPUs

ACM Reference Format:

Phuong Nguyen, Pratik Nayak, and Hartwig Anzt. 2023. Porting Batched Iterative Solvers onto Intel GPUs with SYCL. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023)*, November 12–17, 2023, Denver, CO, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3624062.3624181>

1 INTRODUCTION

Batched iterative solvers have recently received a lot of attention due to their efficiency in solving batches of small and medium-sized sparse problems [8, 25, 26]. Similar to the monolithic problems, batched iterative solvers in particular outperform their direct counterparts if they can use the solution of a similar problem, for example, the previous system in a Picard loop, as the initial guess, which can dramatically shorten the iteration process. For a sequence of linear systems, direct solvers always have to start from scratch

with a complete (sparse) factorization for each problem. Generally, sparse direct solvers have the disadvantage that the fill-in in the factorization process is unknown a priori. In the batched case, this conflicts with the goal of packing all operations of the solve into a single kernel to reduce the main memory access. So batched sparse direct solvers will typically compose of two kernels with a memory allocation in-between, while batched iterative solvers can execute as a single kernel that can leverage data locality.

Batched routines were originally developed for NVIDIA GPUs, as these GPUs were the first to be used in scientific computing [4, 11, 16]. The batched kernels were written in NVIDIA's CUDA programming ecosystem [29]. In the last years, an increasing number of leadership systems are equipped with GPUs from other vendors, including AMD and Intel, there is a need to port these routines beyond the CUDA backend. SYCL drew our interests due to its performance efficiency and portability on different architectures, such as CPUs, GPUs, and FPGAs [19, 24]. SYCL is a cross-platform abstraction layer inspired by OpenCL [33]. Its underlying fundamental principles of portability and efficiency enable the composition of code, in a "single-source" style using completely standard C++, for heterogeneous architectures. SYCL's increasing popularity has led to the development of a diverse range of implementations within its ecosystem [7, 13]. Among them, the Intel oneAPI Toolkit provides a SYCL implementation and compilers which are highly efficient for Intel GPUs [23].

Historically, batched functionality was developed for scenarios where many small and independent problems had to be tackled in parallel with the same algorithm, with each small problem being too small to fully use the available compute resources. In that sense, batched functionality is suitable for data-parallel problems. Typical use cases for batched functionality are parallel applications of a linear operator as a dense or sparse batched matrix-vector multiplication [31], the parallel solution of a set of pairwise-independent linear systems [12], or the parallel singular value decomposition [15]. The use of these methods spans from high-order FEM schemes over tensor contractions in quantum Hall effects, astrophysics, metabolic networks, and quantum chemistry to image and signal processing. With the rise of machine learning and the heavy use of deep neural networks, the batched dense matrix-matrix multiplication [4] has become the most prominent use case for batched functionality. In many cases, the data-parallel problems arise by breaking down a large problem into many small problems that can be handled more efficiently if they are considered independently. A colorful example is the application of a block-Jacobi preconditioner that can be expressed either by applying a block-diagonal matrix to a global vector or by applying a set of small dense matrices to vector segments, with the latter allowing for a more localized kernel execution. In consequence, batched operations are traditionally designed to perform the same pre-defined sequence of operations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC-W 2023, November 12–17, 2023, Denver, CO, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0785-8/23/11...\$15.00
<https://doi.org/10.1145/3624062.3624181>

on all problems of the set. This allows for handling all problems in a SIMD-fashion: the problem-individual properties have no influence on the batched kernel execution. Given the strong demand for batched functionality and the possibility to leverage hardware characteristics in the performance optimization of batched functionality, the community has agreed on a de-facto batched BLAS interface convention [3] that mostly adheres to the vendor implementation of batched BLAS libraries [21, 30].

Batched dense functionality is often used in sparse linear algebra computations, for example when handling problems that contain small dense blocks, such as high-order finite element discretizations or supernodal factorizations. At the same time, there exists little work on batched sparse functionality. One of the earliest efforts on batched sparse functionality is the batched sparse matrix vector product kernel developed by Collins et al. [11]. However, the design and execution of the batched sparse matrix vector multiplication kernel is different from the traditional batched functionality as it uses different storage formats for the distinct sparse problems and launches suitable kernels – matching the storage format – via run-time polymorphism. With regard to batched sparse direct solvers, we mention in passing some work on batched tri-diagonal and penta-diagonal systems [17, 36]. However, these direct methods are restricted to batched tri- and penta-diagonal systems and typically utilize only one GPU thread per batch entry to solve it sequentially. While this approach is advantageous for certain types of problems, it does not utilize the fine-grained parallelism as is possible with iterative methods.

Recently, there have been some developments in the batched iterative solver, as an alternative to batched direct methods [5, 27]. In particular, for GPUs, it has been shown that the batched iterative methods can match/outperform the batched direct counterparts. Applications such as combustion and fusion plasma simulations need to solve hundreds of thousands of small to medium linear systems, each sharing a sparsity pattern. For the linear solution of these systems, placed inside a non-linear loop, it is advantageous to use an iterative solver, as that allows to incorporate an initial guess which can accelerate the linear system solution within the outer loop. Additionally, we might not need to solve the system to machine precision accuracy but can control the solution accuracy based on the parameters of the outer non-linear loop [5].

In this paper, we extend our previous GPU-native batched linear solvers to the SYCL ecosystem and showcase the performance on Intel GPUs. We compare the performance of the batched solvers on Intel GPUs against the NVIDIA GPUs, both on their vendor native programming models (SYCL and CUDA respectively). We explore the subtleties in the kernels and the differences required to optimize performance for both programming models. Focusing on a simple 3-point stencil discretization problem, we study scaling and demonstrate that we outperform the state-of-art for all problem sizes. We also use matrices derived from the Pele reaction flow simulation application that uses SUNDIALS [20] to solve the ODE linear systems, which lend themselves to batched solutions.

In Section 2, we provide some background on batched solvers, their need in applications, and aspects that need to be considered for batched iterative solvers. We also briefly detail the SYCL programming model and its features and the Intel GPU hardware characteristics. In Section 3, we detail the design of our batched sparse

iterative solvers. In particular, we showcase our batched iterative solver design that comes with the flexibility of using different preconditioners, stopping criteria, and sparse matrix storage formats, and monitor the solver convergence for each system in the batch individually. We also elaborate on the specific SYCL optimizations that enable us to maximize performance on the Intel GPU.

After presenting all implementation details, in Section 4 we investigate the hardware performance, and time-to-solution of the batched sparse iterative solver technology. This includes the evaluation for benchmark problems arising in real-world PeleLM combustion applications. We also briefly discuss the performance portability and productivity of the implemented SYCL-based solvers. In Section 5, we summarize our findings and provide a roadmap for the extension of the batched sparse iterative functionality that we plan to provide in the Ginkgo open-source library [9, 10].

In summary, we make the following contributions:

- Successfully porting the batched iterative solvers onto the Intel GPUs using the SYCL programming model.
- Performance tuning for the ported solvers on the Intel GPUs for a wide range of matrix sizes.
- Performance evaluation for two different paradigms: a three-point stencil matrix used to study scaling behaviour, and matrices from the PeleLM application.
- A thorough evaluation of the performance of the batched iterative solvers on Intel GPUs and their comparison against the latest NVIDIA H100 GPU, both using the vendor native programming models.

2 BACKGROUND

Consider applications such as a chemical reaction or astrophysical simulations, which aim to evolve the reactive flow in time on a discretized 3D mesh. These simulations typically operator split the reactions from the hydrodynamics and hence require a solution of many independent chemical reaction ODEs. The resulting chemical reaction equations are usually very stiff, requiring the usage of implicit time stepping schemes such as the Backward Differentiation Formula(BDF) [20]. In each time step, one needs to solve a non-linear system. Solving this non-linear system with, for example, a Newton iteration requires the solution of linear systems. These linear systems characterize the reaction of species in the domain. As the species in the domain are the same across all cells and the reaction matrix is defined for the species, for each spatial discretization cell, we need to solve a linear system, with the linear system for all cells sharing the sparsity pattern.

Many other applications such as fusion plasma simulations or finite element simulations also require the solution of independent linear systems [25], and particularly within a non-linear iteration loop.

2.1 Batched iterative solvers

In contrast to batched direct solvers, batched iterative solvers provide the possibility to vary the solution accuracy, which can be beneficial to reduce the runtime of the non-linear iteration. Additionally, iterative solvers can incorporate solution information in the form of an initial guess, which can accelerate the overall time to solution.

This tunable accuracy and initial guess capabilities come at the cost of complexities in design and implementation. Iterative solvers do not follow a pre-defined execution, but the number of iterations depends on the matrix properties, the stopping criterion, and the precision format being used. Due to the nature of iterative solvers, the design needs to take into account the efficient composition of kernels with the plethora of parameters that are necessary for an efficient iterative solution.

With hierarchical architecture such as GPUs, the design needs to minimize memory movement, maximize local memory usage, and the overall occupancy of the GPU to maximize the performance. This optimization typically has to account for the target problem or the target problem class. We here focus on linear systems of small to medium size (of the order of 10 to 2,000 rows) and system matrices sharing the same sparsity pattern.

2.2 Overview of Ponte Vecchio GPUs architecture

To motivate the design of batched iterative solvers for Intel GPUs, we provide some technical background on the Intel GPU architecture.

The X^e-core is the smallest thread-level building block of the PVC GPU. Each X^e-core consists of eight X^e Vector Engines (XVEs), each of which have a 512-bit register, and can therefore perform 256 FP32 or FP64 operations per cycle. Each X^e-core can execute eight multithreads simultaneously, with each hardware thread having 4096 bytes of private memory in the form of 128 general purpose registers.

Each PVC GPU consists of 2 stacks which are connected via a *Stack-to-Stack* link. Each stack has its own dedicated resources: 4 X^e-slices, a Level 2 Cache, and a directly connected 64 GB of High Bandwidth Memory (HBM). Each X^e-slice consists of 16 X^e-cores. Overall, each PVC GPU has 128 X^e-cores and 128 GB of HBM.

Even though the two stacks have separately connected HBMs, this HBMs can be accessed directly by the other stack. This feature enables fast and efficient communication between the two stacks, via the HBM.

In practice, when running an application, this two-stack GPU can be seen as a single GPU device. The GPU driver and the runtime work together to automatically distribute the workloads across the two stacks. This describes the so-called *implicit scaling mode*. In contrast to the *implicit scaling mode*, the *explicit scaling mode* allows users to explicitly allocate the workloads and memory placement for each stack. Each stack then executes its own workloads.

As one may find more familiar with NVIDIA GPU terminology, the mapping between NVIDIA's terminology and Intel's terminology is provided in Table 1. Overall, with the exception that the PVC GPUs consist of 2 stacks, all other terminologies can be directly paired with the NVIDIA terminology.

2.3 SYCL Programming Model

SYCL is a Khronos Group language standard that enables developers to express data-parallel computations using standard C++ templates and lambda functions, abstracting the underlying hardware complexity and allowing seamless execution on diverse accelerators.

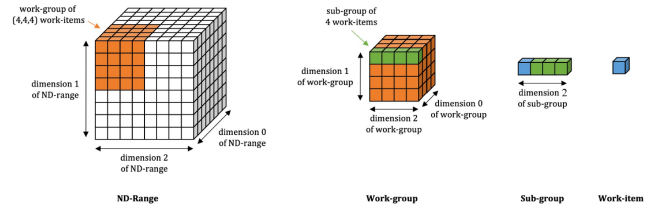


Figure 1: Hierarchy of the SYCL kernel index space [32].

Table 1: GPU architecture terminology mapping [23]

CUDA Capable GPUs	Ponte Vecchio GPUs
CUDA Core	XVE
Streaming Multiprocessor	X ^e -Core (XC)
Processor Cluster	X ^e -Slice
N/A	X ^e -Stack

Table 2: Execution model mapping from CUDA to SYCL [23]

CUDA	SYCL
Thread	work-item
Warp	sub-group
Block	work-group
Grid	ND-range

The SYCL kernel consists of the main kernel computation which is expressed as a C++ lambda function, the argument values associated with the kernel, and the parameters that define an index space. The kernel index space is often defined via an ND-Range.

Figure 1 illustrates the index hierarchy of the kernel instance, in which the smallest kernel execution unit is called a *work-item*. Multiple consecutive work-items can be organized into a 1-dimensional set called a *sub-group*. The computation of a sub-group can be processed by one or few SIMD operations on a XVE. Additionally, one can also perform collective operations such as broadcast, shuffle, reduction, etc within a sub-group.

A *work-group* consists of a 1-, 2-, or 3-dimensional set of consecutive sub-groups. Each work-group has a local memory which is shared among all the work-items within a work-group. Depending on the implementation and the hardware availability, this Shared Local Memory (SLM) can be mapped into different physical memories. On Intel GPUs, the SLM is allocated on the L1 cache. Typically, the work-items in a work-group are executed together on a X^e-core. Depending on the work-group size and availability of the L1 cache, each X^e-core can handle multiple work-groups at a time. Additionally, the work-items within a work-group can be synchronized via local memory fences but synchronization across different work-groups is not possible in SYCL.

As one may be more familiar with CUDA terminologies, the execution model mapping between CUDA and SYCL can be found in Table 2.

Table 3: Batched feature support in Ginkgo

Mat. formats	Solvers	Preconditioners	Stop. criteria
BatchDense	BatchCg	BatchJacobi	Absolute
BatchCsr	BatchBicgstab	BatchIlu	Relative
BatchEll	BatchGmres	BatchIsai	
	BatchTrsv		

3 IMPLEMENTING BATCHED ITERATIVE SOLVERS

In this section, we discuss the design and the interface of batched sparse iterative solvers. Our goal is to develop batched sparse iterative solvers to be flexible in terms of accepting a preconditioner transforming the linear systems $A_i x_i = b_i, i = 1 \dots n$ into the preconditioned system $M_i A_i x_i = M_i b_i, i = 1 \dots n$ with M_i being a preconditioner adjusted to the specific system A_i , but all preconditioners M_i being of the same preconditioner type.

The objective of a batched solver interface is to leverage the data parallelism present in the batched problem at hand and map it to the hardware parallelism available. Batched solvers are favorable in cases where the individual matrices are relatively small, and a large numbers of these independent linear systems needs to be solved. The criteria that influence the design and implementation are the following:

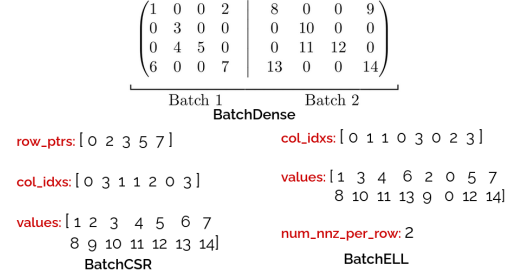
- (1) The size of the individual batch entries: the number of rows and the number of non-zeros.
- (2) The number of linear systems to be solved.
- (3) Common sparsity patterns between the batched matrices, if any.
- (4) Properties of the batched linear systems that influence convergence (condition numbers, etc.)

To enable support for a wide variety of applications, Ginkgo supports different batched matrix formats, solvers, and preconditioners, as shown in Table 3. We note that due to the templated design, any of the columns can be combined with another, with only a few exceptions (such as **BatchIsai** needing the **BatchCsr** matrix format).

3.1 Batched matrix formats

Sparse matrices typically store an array of non-zero values, as well as integer arrays encoding the sparsity pattern. For our problem space, all the matrices in the batch share the same sparsity pattern. Therefore, to minimize memory requirements, we store only one copy of the sparsity pattern for the batched matrix formats. We additionally implement two batch matrix formats, one general format, **BatchCsr**, and a specialized format, **BatchEll** in addition to the dense matrix format, **BatchDense**.

The **BatchCsr** matrix format is based on the popular Compressed Sparse Row (CSR) matrix storage format, where one stores an array of column indexes per row corresponding to each non-zero value in the matrix. An accumulated sum of the number of non-zeros per row is additionally necessary. This matrix format is suitable for general matrices with large variations in the number of non-zeros per row and performs generally well for most sparsity patterns.

**Figure 2: Batch Matrix Storage formats - BatchDense, BatchCsr and BatchEll**

The **BatchCsr** is an extension of this format where we store the column indexes and the row pointers for only one matrix and store the values of all the matrices.

For matrices that have a similar number of non-zeros in every row, we can optimize the storage by padding the rows to a uniform number of non-zeros per row, removing the need for a pointers array. This also gives us additional advantages in terms of coalesced accesses. The **BatchEll** matrix format stores one set of column indexes and the values of all the batch entries. In contrast to **BatchCsr**, we store the column indexes and the values in column-major allowing for coalesced accesses which is suitable for GPUs.

Figure 2 visualizes the schematic and the storage requirements of **BatchCsr** and **BatchEll** compared to the **BatchDense** format. With batched sparse matrix formats, the additional cost of storing the indexes and the pointers can be easily amortized over an increasing number of systems in the batch. The storage requirements therefore are:

- (1) **BatchDense**: $\text{num_matrices} \times \text{num_nnz_per_matrix}$
- (2) **BatchCsr**: $[\text{num_matrices} \times \text{num_nnz_per_matrix}] + [(\text{num_rows} + 1) \times 1] + [\text{num_nonzeros_per_matrix} \times 1]$
- (3) **BatchEll**: $[\text{num_matrices} \times \text{num_nnz_per_matrix}] + [\text{num_nnz_per_row} \times \text{num_rows} \times 1]$

3.2 Batched solver kernels

Iterative solvers such as CG, BiCGSTAB, GMRES can be easily composed of BLAS 1, BLAS 2, and sparse matrix vector operations [34]. Ginkgo supports different batched versions of the iterative solvers, suitable for matrices with different properties and these are listed in Table 3. With our problem space consisting of small to medium-sized linear systems, and our aim to optimize the compute and memory usage, we map one work-group to one linear system. This enables us to write efficient kernels for each linear system without worrying about global synchronization between workgroups (as each linear system is independent and requires no communication).

With the sparse matrix vector product being the workhorse of the Krylov solvers, we implement tuned SpMV kernels for each batched matrix format. For the **BatchCsr** matrix, we implement a sub-group to row-based mapping for matrices which provides good performance for general matrices. For matrices that are more balanced and have only a few nonzeros per row, the **BatchEll**

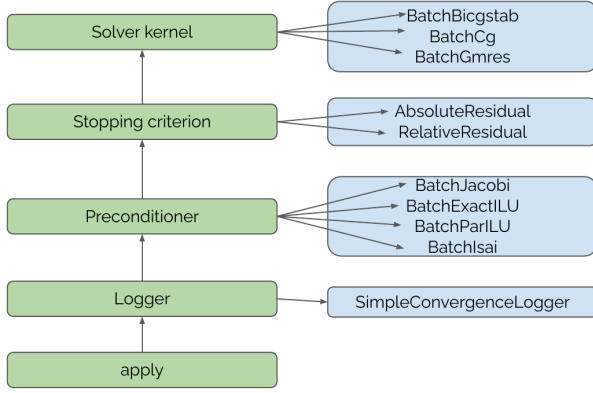


Figure 3: Multi-level dispatch mechanism

matrix format is more suitable, which handles one row per work item removing the need to communicate between thread using sub-warp reductions [25].

In addition to the SpMV kernel, kernels such as dot, scalar addition, and norm are also implemented. Reduction operations such as dot and norm are implemented using the reduction over the whole work-group which is a primitive function provided by SYCL. For small matrices, it is more efficient to implement the reduction within a subgroup since we do not need to read/write through the SLM. These reduction operations were implemented in a different fashion compared to our CUDA-based solvers as in CUDA only warp-level reductions are used as no efficient thread-block level reduction operations are available.

We note that these building blocks are all device kernels and in-lined, involving no host-device transfers. This enables the compiler to optimize the entire solver kernel as a whole. Additionally, the SpMV, scalar operations, dot, and norm kernels are shared between the different solvers, reducing code duplication and improving code sustainability.

3.3 Multi-level dispatch mechanism

We design a multi-level dispatch mechanism as shown in Figure 3 that preserves flexibility, enabling the runtime choice between the different matrix formats, solvers, stopping criteria and preconditioners.

3.4 Minimizing kernel launch latency

For batched solvers, the time to solution for one batch item can be very small, particularly for small linear systems. Launching one kernel for each batch item or for even a few items at once is therefore intractable. To minimize kernel launch overhead, we gather all functionality in a single kernel handling all items of the batch.

The strategy of packing all functionality into a single kernel enables the compiler to optimize the templated kernel as a single instance after the multi-level dispatch mechanism has instantiated the different kernel options (precision format, matrix format, preconditioner, stopping criterion). The modern C++ templating mechanism in this case not only avoids code complexity, but also

Algorithm 1 The **BatchCg** solver.

```

1: for  $b < N_{batches}$  do
2:    $r \leftarrow b - Ax, z \leftarrow Mr, p \leftarrow z, t \leftarrow 0$ 
3:    $\rho \leftarrow r \cdot z, \alpha \leftarrow 1, \hat{\rho} \leftarrow 1$ 
4:   for  $i < N_{iter}$  do
5:     if  $|\rho| < \tau$  then
6:       break
7:     end if
8:      $t \leftarrow Ap$ 
9:      $\alpha \leftarrow \frac{\rho}{p \cdot t}$ 
10:     $x \leftarrow x + \alpha p$ 
11:     $r \leftarrow r - \alpha t$ 
12:     $z \leftarrow \text{PRECOND}(r)$ 
13:     $\hat{\rho} \leftarrow r \cdot z$ 
14:     $p \leftarrow z + \frac{\hat{\rho}}{\rho} \cdot p$ 
15:     $\rho \leftarrow \hat{\rho}$ 
16:   end for
17: end for

```

kernel branching which is prohibitively expensive when handling small problems.

The kernel execution then handles the solution of all items in the batch with the same solver configuration.

3.5 Maximizing local memory usage

Maximizing the performance of the batched solvers requires efficient usage of both the compute and memory hierarchy. It is essential to keep frequently used data in SLM since accessing this memory has a lower latency than the global memory.

In our implementation, we map one linear system into one work-group, i.e. each work-group solves one linear system at a time. Within that, each work-group keeps its intermediate vectors which requires for the iterative solvers to solve the system separately. The sizes of these vectors depends on the size of the batch item matrix. To reduce the latency of memory accesses, it is beneficial to allocate these intermediate vectors on the SLM. The pre-conditioned matrix and a copy of the result vector x are also allocated on the SLM as they are repeatedly used in the solvers kernel. Besides, the system matrix and the right-hand side are read-only data that needed to be fetched from the global memory in every iteration, but their sizes are relatively large for being kept inside the SLM. Therefore, caching these data into another level cache, for example, L2, is favorable.

For medium to large matrix sizes, allocating all these objects on the SLM is impossible as the size of the SLM is limited. For each batched iterative solver type, we prioritize these intermediate vectors based on its usage frequency and sizes. Based on this priority, the solvers dynamically determine at runtime how many vectors can be allocated on the SLM, given the input matrix size and the available SLM memory on the device. The host then selects and dispatches the appropriate kernel which allocates the required amount of SLM and assign these objects accordingly.

The priority for storing those objects on the SLM can be illustrated through the **BatchCg**, for example. Its algorithm can be found in Algorithm 1. Based on the usage frequency and the size of these objects, the priority we assign in decreasing order is: r, z, p, t, x .

The preconditioner workspace is also allocated on the SLM if the SLM is still available.

3.6 Optimizations based on the matrix size

It is important for the batched solvers to have good performance across a wide range of matrix sizes so that it meets the needs of different real-world applications. In this section, we discuss the optimizations techniques we used for our SYCL-based batched solvers relying on the size of the input matrices.

The performance of GPU kernels often depends highly on the execution configuration. Since we assign one linear matrix system to one work-group, it is beneficial to select the work-group size based on the input matrix size. In our implementation, the work-group size is chosen dynamically at runtime depending on the number of rows of the input matrix, as follows:

- The work-group size should not exceed the maximal work-group size supported by the device.
- The work-group size should be divisible by the sub-group size.
- The work-group size should be at least equal to the number of rows so that the *SpMV* kernel performs efficiently.

Thus, for small matrices, in the case when the number of rows is divisible by the sub-group size, the number of rows is chosen as the work-group size. Otherwise, we choose the work-group size equals to the next round-up number by sub-group size of the number of rows. This round-up strategy shows a performance improvement for some input cases, as shown in Table 6 (Appendix A). The maximal work-group size is selected in the case of large input matrix cases. These selecting work-group strategies not only increases the thread utilization within a work-group but also increases the number of possible work-groups that can be scheduled on the same XVE at a time, enhancing the GPU occupancy.

Besides the work-group size, the sub-group size also plays an important role in the performance of the GPU kernels in SYCL. In fact, the Intel PVC GPUs support two different sub-group sizes: 16 and 32. For our batched solver implementations, empirically, we measured that a sub-group size of 16 is better for the small matrices, while a size of 32 performed better for larger input matrices. Thus, the SYCL-based batched solvers are implemented in such a way that they can choose the sub-group size, in addition to the work-group size, dynamically, at runtime. In practice, since SYCL only allows to enforce the sub-group size with a compile-time known number, we use *C++ templating* to instantiate kernels with all possible sub-group size values and the proper kernel is selected at runtime based on the matrix size.

Additionally, as the matrix size also dictates whether the reduction operations use a single sub-group or the whole work-group, we implemented our batched solvers in such a way that the selection can happen at runtime for these reduction operations as well. Again, we use the *C++ templating* ideas to minimize runtime overhead.

Overall, due to these optimizations, the batched solvers switch between different paths in the kernel launch stage, and the selected kernel depends on the matrix size. Since the thresholds between small and large matrix sizes are different for different GPUs capabilities, these thresholds need to be determined experimentally for

Table 4: Reference for data inputs

Input case	# Unique matrices	Matrix size	# Nnz/matrix
<i>3pt stencil</i>	-	-	$3 \times n_{rows}$
<i>drm19</i>	67	22 x 22	438
<i>gri12</i>	73	33 x 33	978
<i>gri30</i>	90	54 x 54	2560
<i>dodecane_lu</i>	78	54 x 54	2332
<i>isooctane</i>	72	144 x 144	6135

Table 5: GPUs specifications

	A100	H100	PVC-1S	PVC-2S
FP64 Peak (TFLOPs)	9.7	26	22.9	45.8
HBM BW Peak (TB/s)	1.6	2.0	1.6	3.2
Shared Local Mem. (KB)	192	228	128	128

each targeted device before using these solvers to ensure optimal performance for that architecture.

4 PERFORMANCE EVALUATION

In this section, we evaluate runtime and scalability of the batched solvers on the latest Intel GPUs with SYCL and benchmark the performance against the batched solver implementation for NVIDIA GPUs with CUDA.

4.1 Experimental setup

To evaluate our SYCL-based batched iterative solvers, we consider two classes of inputs, summarized in Table 4:

- Matrices generated from a 3-point stencil, whose number of rows can be increased as necessary to study the scaling behaviour of the batched solvers.
- Matrices from the PeleLM+SUNDIALS application, which consists of matrices derived from different reactive flow simulations. Each mechanism gives us a different matrix set and we have as many items in the batch as the number of cells in the mesh. As we would like to have a smaller test case, we extract the matrices from the application for a few cells and replicate it to emulate the solution for a larger mesh. The matrices for this application are fairly small (22 rows to 144 rows) and relatively dense. More information on this dataset is available in [5].

In the experiment, all input matrices are stored in the **BatchCsr** format. We study the performance for two batched iterative solvers, the **BatchCg** and the **BatchBicgstab**. Additionally, the PeleLM+SUNDIALS matrices use a scalar Jacobi preconditioner to accelerate convergence.

We compare the batched iterative solvers on three different GPUs:

- (1) NVIDIA A100 80GB PCIe using CUDA 11.8.0
- (2) NVIDIA H100 PCIe Gen 5 using CUDA 11.8.0
- (3) Intel Data Center GPU Max 1550 (PVC) using Intel(R) oneAPI DPC++/C++ Compiler 2023.2.0

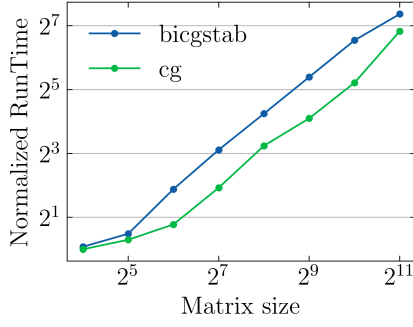
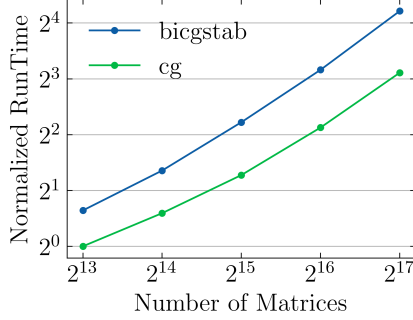
(a) W.r.t matrix sizes, 2^{17} matrices(b) W.r.t number of matrices, the matrix size (64×64)

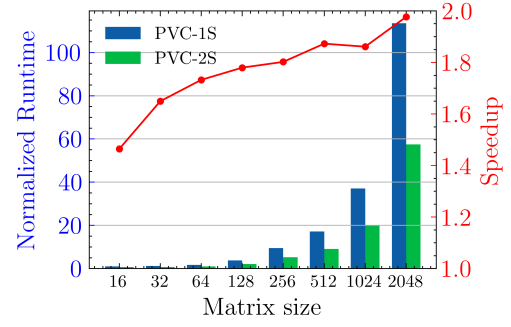
Figure 4: Scaling of the SYCL batched solvers on 1 stack of the PVC GPU with respect to the problem sizes using the synthetic input. The runtimes of the solvers scale almost linearly with the problem size.

Some salient features of the GPUs that are relevant in our case is tabulated in Table 5.

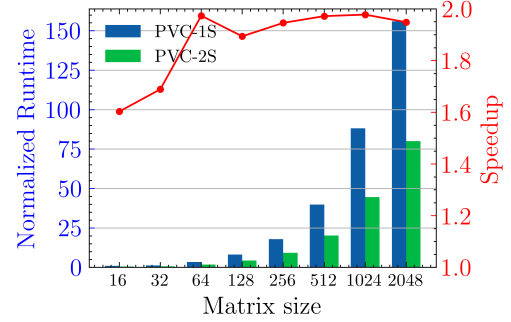
4.2 Scaling evaluation with the synthetic input

Using a standard 3-point stencil problem, we can generate a batch of symmetric, positive definite (SPD) matrices that allows us to do scaling experiments in both the matrix size and the batch size. Figure 4 shows the scaling behaviour using 1 stack of the Intel GPU for both the **BatchCg** and the **BatchBicgstab** solver. In Figure 4a, we fix the batch size to 2^{17} and increase the matrix size (number of rows) for each of the batch items. As expected, the overall runtime increases linearly with the matrix size. In Figure 4b, we increase the number of items in the batch from 2^{13} to 2^{17} for a individual problem size of 64×64 and again observe a linear increase in the run-time. This means that we are able to fully saturate the GPU, and additional linear systems need to wait for the current systems to be completed. Overall, the great scalability of our batched solvers would allow them to address different potential simulations which are problem-size dependent.

The Intel GPU consists of two separate stacks that can be viewed as a single GPU or as two separate GPUs, as previously mentioned. The batched solvers, being embarrassingly parallel, can take advantage of this feature via the implicit scaling mode. It can be done automatically as the Intel GPU driver can split the workloads, i.e. number



(a) Batch CG



(b) Batch BiCGSTAB

Figure 5: Performance comparison of the SYCL batched solvers on 1- and 2-stacks of the PVC GPUs with respect to different matrix sizes. The synthetic input set is used with 2^{17} matrices. For the speedup, the performance of 1-stack is used as a baseline. With implicit scaling on 2-stacks, the two solvers achieve between 1.5x – 2.0x speedup, and the larger matrix size, the higher speedup.

of matrices, and schedule them on the two stacks without explicit requests from the users. This implicit scaling behaviour and its benefits are shown in Figure 5 for both **BatchCg** and **BatchBicgstab** solvers. We observe on average a 1.8x speedup for **BatchCg** and 1.9x for the **BatchBicgstab** solver going from 1 stack to 2 stacks, revealing that the batched solvers indeed provide the embarrassing parallelism that can be harnessed by the GPUs. The speedup, however, is lower than 2x due to the NUMA effects, as the memory allocation is not perfectly split across the two stacks with the *implicit scaling* mode. Nevertheless, the reasonable achieved speedup suggests that we can easily scale to multiple GPUs as distributing these batched matrices over the MPI ranks is trivial and no additional communication is necessary.

4.3 Performance evaluation with the real application inputs

For benchmarking the performance of the batched solvers on the Intel GPUs, against the batched solvers on state of the art NVIDIA GPUs, we use matrices from the PeleLM+SUNDIALS application, described in Table 4. Since these matrices are non-SPD, **BatchCg** can not be used to solve these systems, thus only **BatchBicgstab** is

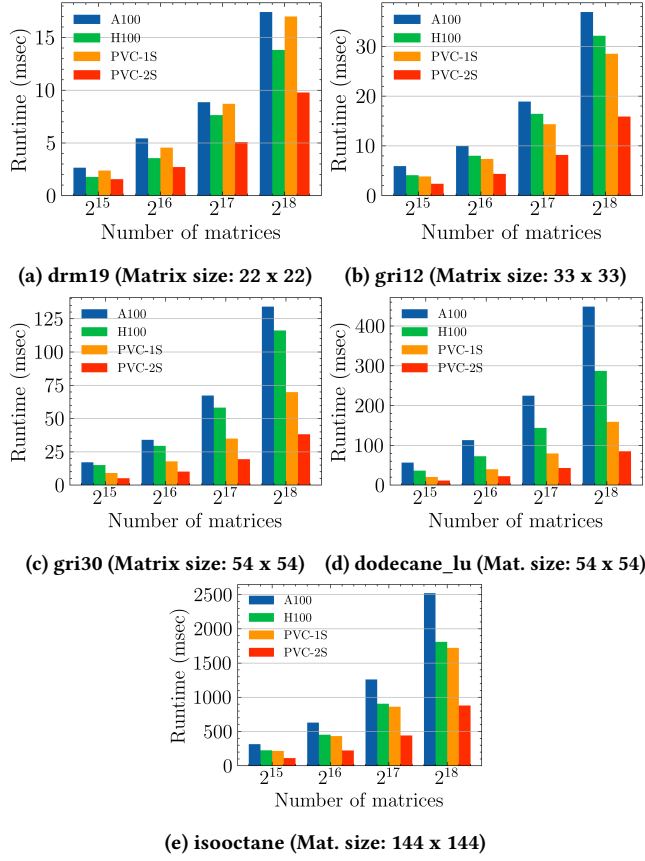


Figure 6: Runtime of the two batched solvers on the three GPUs with different input data from the PeleLM simulations. Overall, the SYCL solvers on the PVC GPUs outperform the ones on the NVIDIA H100 for all input cases.

evaluated in this section. The vendor-native programming models are used: SYCL-based solvers for Intel PVC GPUs and CUDA-based solvers for NVIDIA A100 and H100. The CUDA implementation of the batched iterative solver can be found in our previous paper [5].

Figure 6 presents the runtime comparison of the solvers for solving the five test problems on NVIDIA A100, NVIDIA H100, and Intel PVC GPUs. Overall, the batched solver on 2 stacks of the PVC GPUs outperforms the ones on the A100 and H100 GPUs significantly for all matrices and batch sizes. In addition, the figure also demonstrates that the SYCL-based batched solvers scale well on real application inputs in a similar fashion to the previous experiment with the synthetic input matrices.

Figure 7 provides a direct performance comparison of the batched solvers on the three GPUs. Except for the *gri12* case, all other input cases shows notable performance of the solvers on 1 stack of the PVC GPUs when compared with the NVIDIA GPUs. In average, the PVC-1S is 1.7x and 1.3x faster than the A100 and H100, respectively, across all input cases. Similarly, the PVC-2S outperforms the A100 and H100 by an average factor of 3.1 and 2.4, respectively.

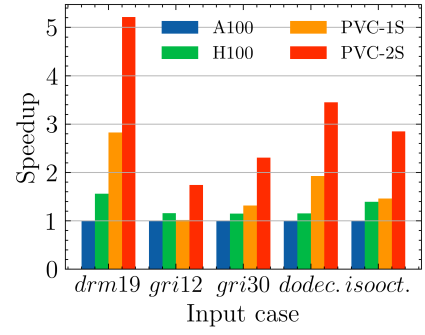


Figure 7: Normalized speedup comparison for different input cases with 2¹⁷ matrices and the runtime of A100 is the baseline. The PVC GPU with 1- and 2-stacks are 1.3 and 2.4 times faster than the H100, in average.

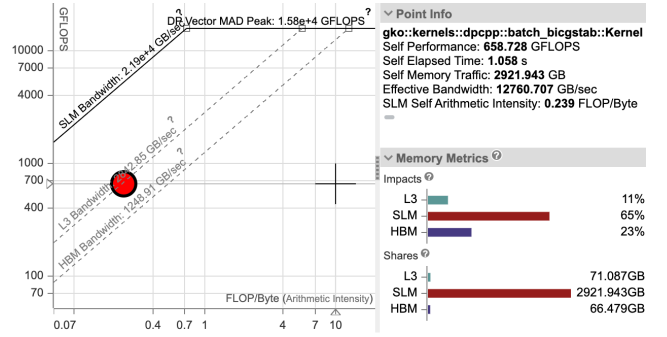


Figure 8: Roofline analysis and memory metrics of the BatchBigstab for the *dodecane_lu* input case with 2¹⁷ matrices on 1-stack of the PVC GPU. The solver performance relies heavily on the Shared Local Memory and has not reach the SLM Bandwidth Bound.

4.4 Roofline analysis

The performance of the SYCL-based batched solvers on the Intel GPUs is also evaluated using the Intel Advisor Tool.

We analyse the **BatchBigstab** solver for the *dodecane_lu* input case with the batch size of 2¹⁷ matrices on 1 stack of the PVC GPU.

The profiling results outline that the XVE Threading Occupancy is around 50% with the XVE Array Active stays around 40%. This means that the kernel workloads do not fully occupy all available XVE on the X^e-cores. This is expected, as in the solver kernel implementation, we let each work-group use the maximum amount of shared local memory available regardless of the work-group size. With this strategy, even if the thread-group size is smaller than the number of work-items a X^e-core can handle, there are not more thread-groups get scheduled on the same X^e-core due to the limit on the available shared local memory. In other words, we trade the XVE occupancy for increased amount of shared local memory usage in each work-group, which is more important to achieve a good performance for the batched iterative solvers, as previously discussed in Section 3.5.

Figure 8 presents the roofline performance chart and the memory metrics of the solver. The time breakdown for the memory subsystem shows that 65% of the time spent for memory transactions is spent on the SLM requests. In addition, there are almost 3 TB of data passes through the SLM which is much larger than the one passes through either the L3 or HBM. This means that the performance of the solver mainly relies on the efficiency of the SLM accesses which is expected, since the solver keeps all frequently accessed data in the SLM for this input case. In addition, 11% of memory accesses are from the L3 (which is actually the L2 Cache on the GPU stack) implying that the batch matrices and the right hand side (constant objects) are likely cached into the this last level cache, which facilitates the accessing of these matrix from the work-group. In the roofline analysis, we observe that the performance of the solver lies on the L3 Bandwidth roof which is relatively good. However, the solver does not yet reach the SLM Bandwidth roof. Further optimizations to improve SLM accesses, for example identifying possible bank-conflicts and resolving them, will be part of our future work.

4.5 Performance portability and productivity

SYCL itself is a portable programming model and several applications use SYCL to successfully support multiple hardware platforms, such as [18, 24]. For the Ginkgo project, the existence of such a portable programming model promises both performance portability and productivity for the library, potentially giving us the opportunity to focus our work on algorithm developments instead of supporting multiple backends. Nonetheless, porting the Ginkgo SYCL backend to other platforms posed several challenges, as discussed below.

Our first attempt was to port the Ginkgo SYCL backend, which includes both normal routines and batched routines, onto the NVIDIA GPUs. We used the *llvm* compiler with the SYCL extensions developed by Intel, as it supports SYCL on the Intel GPUs, AMD GPUs, and NVIDIA GPUs [22]. Some components of the Ginkgo SYCL backend employ routines from the Intel oneAPI Math Kernel Library (oneMKL) and the Intel oneAPI DPC++ Library (oneDPL) which are partially supported on the NVIDIA GPUs [1]. We experienced an issue with linking against those Intel libraries while compiling the Ginkgo SYCL backend for NVIDIA GPUs, an issue which is not resolved at the time of this paper. Since the implemented batched iterative solvers themselves do not use the routines from neither oneMKL or oneDPL, ideally, we could compile them without linking against these libraries. This can be done for portability evaluation purposes, though other routines may not be fully functional. However, even if this is done, the current state of the SYCL implementation on CUDA platforms for the used *llvm* compiler does not fully support complex floating-point functions [2], hence prohibiting us from porting the routines onto the NVIDIA GPUs.

With the success of the A64FX-based Fugaku system at Riken, we have seen high interest in sparse linear algebra libraries for ARM-based systems [6, 18, 35]. Additionally, SYCL compilers for ARM processors have been developed [7]. Therefore, we attempted to port the Ginkgo SYCL backend for the A64FX processor. For this, we used the OpenSYCL to compile our SYCL backend on

the Oukami Cluster at Stony Brook University via the ACCESS program [14]. We successfully verified the compiler and the setup environment with a simple standalone kernel - *vector-add*. For compiling Ginkgo SYCL backend, we faced two main issues. First, oneMKL and oneDPL do not support ARM processors. Linking against other ARM-supported math libraries requires re-writing all API calls, as there is not yet a standard for sparse linear algebra routines. Second, while OpenSYCL has made significant progress in the implementation, it has not achieved full SYCL conformance. The SYCL-based batched iterative solvers, therefore, are not yet portable to the A64FX processors.

Nevertheless, Ginkgo supports multiple hardware architectures including CPUs, NVIDIA GPUs, AMD GPUs, and Intel GPUs through OpenMP, CUDA, HIP, and SYCL backends. Hence, Ginkgo itself is a performance portable library from a user’s perspective, as it provides users sparse solvers across multiple platforms. Even though the portability of the SYCL backend in general and the SYCL-based batched iterative solvers in particular on other non-Intel GPUs platforms are not achieved at the time of the paper, the developed SYCL-based solvers hold promises for our future research endeavors in developing a portable backend as well as improving the productivity of Ginkgo developers.

5 CONCLUSION

We have presented our successful work on porting the Ginkgo’s batched iterative solvers onto Intel GPUs with the SYCL programming model. The design of the batched iterative solvers is delineated and the different optimization strategies are discussed. The SYCL-based batched iterative solvers shows nearly linear scaling with respect to the problem sizes and exhibit effective scaling on 2 stacks of the PVC GPU using the implicit scaling mode. The performance of the ported solvers on the Intel GPU Max 1550s surpasses the one on NVIDIA H100 GPU by an average factor of 2.4 for the matrices from the PeleLM application. Moreover, the solvers demonstrate exemplary the hardware resource utilization, as evidenced by the analysis conducted using the Intel Advisor Tool.

Appendices

A PERFORMANCE APPENDIX

Table 6: Percentage speed-up when rounding the work-group size to a multiple of the sub-group size

Batch size	32768	65536	131072	262144	Average
<i>drm19</i>	0.8	-0.2	0.6	-0.6	1.3
<i>gri12</i>	50.9	54.8	50.8	52.1	48.5
<i>gri30</i>	2.9	2.6	3.2	3.1	3.2
<i>dodecane_lu</i>	1.3	1.5	1.7	1.9	1.6
<i>isooctane</i>	0.0	-0.1	-0.0	0.1	0.1

B REPRODUCIBILITY APPENDIX

In order to ensure reproducibility of results, we provide the code and elaborate on the settings and parameters used to produce these results.

Obtaining the source code

The source code is open-source and available on the Ginkgo-Project Github (<https://github.com/ginkgo-project/ginkgo.git>). The code used for this paper is archived on Zenodo [28].

Building and installing Ginkgo

To build GINKGO with SYCL, the following components are necessary:

- (1) The CMake build platform, CMake-3.26.3 was used in this paper.
- (2) Intel oneAPI Toolkit installation, Intel oneAPI 2023.05.15.006 was used in this paper.

The Ginkgo library and the batched functionality use the same canonical CMake setup as elaborated in the Ginkgo documentation (https://ginkgo-project.github.io/ginkgo/doc/develop/install_ginkgo.html).

Benchmarking

The performance results from this paper can be reproduced following these steps:

- (1) Building GINKGO with SYCL backend:
 - Make a build directory: `$ mkdir build && cd build`
 - Configure GINKGO with the SYCL backend :
`$ cmake -DCMAKE_CXX_COMPILER=icpx -DGINKGO_BUILD_DPCPP=on ..`
 - Compile GINKGO: `$ make`
- (2) Building GINKGO with CUDA backend: follow the reproducibility appendix in [5].
- (3) Performance tests:
 - The benchmarks with the synthetic 3pt stencil input can be found in the directory:
`./ginkgo/examples/batched-solver.`
 - The benchmarks with input matrices from the PeleLM application can be found in the directory:
`ginkgo/examples/batched-solver-from-files.`
 - The benchmarking scripts for both input classes are provided in `run-test-dpcpp.sh` and `run-test-cuda.sh`.

Our setup

The performance of the solvers on the Intel GPUs were measured on the Sunspot - the testbed system for the Aurora supercomputer - deployed by the Argonne Leadership Computing Facility at Argonne National Laboratory, US. Each node consists of 2x Intel Xeon CPU Max Series (Sapphire Rapids) and 6x Intel Data Center GPU Max Series (PVC). Compilers and related libraries are from the Intel oneAPI 2023.05.15.006 Toolkit.

The performance of the solvers on the NVIDIA GPUs were obtained from our in-house testing nodes at the University of Tennessee, Knoxville, as follow:

- 1 node: 2x Intel Xeon Silver 4309Y CPU and 1x NVIDIA H100 PCIe Gen5 80GB.
- 1 node: 2x AMD EPYC 7742 CPU and 8x NVIDIA A100 SXM4 80GB .

GCC-11.3.1 was used as the host compiler and CUDA Toolkit 11.8.0 was used as the device compiler on both nodes.

6 ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. In addition, this work used the Sunspot testbed under the CLOVER, XSDK projects at the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility, and the Ookami cluster at Stony Brook university through allocation CIS230121 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

Special thanks to Thomas Applencourt and Abhishek BaguSETTY for their valuable supports on working with SYCL and the Intel GPUs, Jens Domke for insights and approaches on porting code to A64FX, and John Pennycook for shepherding the paper.

REFERENCES

- [1] 2020. [CUDA] MKL compatibility. <https://github.com/intel/llvm/issues/1548>.
- [2] 2023. [SYCL][CUDA] ptxas fatal: complex floating-point functions. <https://github.com/intel/llvm/issues/8281>.
- [3] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Mawussi Zounon. 2021. A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines. *ACM Trans. Math. Softw.* 47, 3, Article 21 (June 2021), 23 pages. <https://doi.org/10.1145/3431921>
- [4] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. 2016. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9697)*, Julian M. Kunkel, Pavan Balaji, and Jack J. Dongarra (Eds.). Springer, 21–38. https://doi.org/10.1007/978-3-319-41321-1_2
- [5] Isha Aggarwal, Aditya Kashi, Pratik Nayak, Cody J. Balos, Carol S. Woodward, and Hartwig Anzt. 2021. Batched Sparse Iterative Solvers for Computational Chemistry Simulations on GPUs. In *2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 35–43. <https://doi.org/10/gn3xcg>
- [6] Christie Alappat, Nils Meyer, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, and Tilo Wettig. 2022. Execution-Cache-Memory modeling and performance tuning of sparse matrix-vector multiplication and Lattice quantum chromodynamics on A64FX. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6512. <https://doi.org/10.1002/cpe.6512> arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6512>
- [7] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL. In *Proceedings of the International Workshop on OpenCL (Munich, Germany) (IWOC '20)*. Association for Computing Machinery, New York, NY, USA, Article 8, 1 pages. <https://doi.org/10.1145/3388333.3388658>
- [8] Hartwig Anzt, Edmond Chow, Thomas Huckle, and Jack Dongarra. 2016. Batched Generation of Incomplete Sparse Approximate Inverses on GPUs. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 49–56. <https://doi.org/10.1109/ScalA.2016.011>
- [9] Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmaier, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. 2020. Ginkgo: A High Performance Numerical Linear Algebra Library. *Journal of Open Source Software* (Aug. 2020). <https://doi.org/10.21105/joss.02260>
- [10] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmaier, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Orti. 2020. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *arXiv:2006.16852 [cs]* (July 2020). arXiv:2006.16852 [cs]

- [11] Hartwig Anzt, Gary Collins, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017. Flexible Batched Sparse Matrix-Vector Product on GPUs. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (Denver, Colorado) (*Scala '17*). Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. <https://doi.org/10.1145/3148226.3148230>
- [12] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017. Variable-Size Batched LU for Small Matrices and Its Integration into Block-Jacobi Preconditioning. In *2017 46th International Conference on Parallel Processing (ICPP)*. 91–100. <https://doi.org/10.1109/ICPP.2017.18>
- [13] Thomas Appencourt, Brice Videau, Jefferson Le Quellec, Amanda Dufek, Kevin Harms, Nevin Liber, Bryce Allen, and Aiden Belton-Schure. 2023. Standardizing Complex Numbers in SYCL. In *Proceedings of the 2023 International Workshop on OpenCL* (Cambridge, United Kingdom) (*IWOCL '23*). Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. <https://doi.org/10.1145/3585341.3585343>
- [14] Timothy J. Boerner, Stephen Deems, Thomas R. Furlani, Shelley L. Knuth, and John Towns. 2023. ACCESS: Advancing Innovation: NSF's Advanced Cyber-infrastructure Coordination Ecosystem: Services & Support. In *Practice and Experience in Advanced Research Computing* (Portland, OR, USA) (*PEARC '23*). Association for Computing Machinery, New York, NY, USA, 173–176. <https://doi.org/10.1145/3569951.3597559>
- [15] Wajih Halim Boukaram, George Turkiyyah, Hatem Ltaief, and David E. Keyes. 2018. Batched QR and SVD Algorithms on GPUs with Applications in Hierarchical Matrix Compression. *Parallel Comput.* 74, C (May 2018), 19–33. <https://doi.org/10.1016/j.parco.2017.09.001>
- [16] Enda Carroll, Andrew Gloster, Miguel D. Bustamante, and Lennon Ó' Náraigh. 2021. A Batched GPU Methodology for Numerical Solutions of Partial Differential Equations. <https://doi.org/10.48550/arXiv.2107.05395> arXiv:2107.05395 [physics]
- [17] Enda Carroll, Andrew Gloster, Miguel D. Bustamante, and Lennon Ó' Náraigh. 2021. A Batched GPU Methodology for Numerical Solutions of Partial Differential Equations. *arXiv* 2107.05395 (2021). arXiv:2107.05395 [physics.comp-ph]
- [18] Luigi Crisci, Majid Salimi Beni, Biagio Cosenza, Nicolò Scipione, Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Andrea Beccari. 2022. Towards a Portable Drug Discovery Pipeline with SYCL 2020. In *International Workshop on OpenCL* (Bristol, United Kingdom, United Kingdom) (*IWOCL'22*). Association for Computing Machinery, New York, NY, USA, Article 5, 2 pages. <https://doi.org/10.1145/3529538.3529688>
- [19] Tom Deakin, Andrei Poenaru, Tom Lin, and Simon McIntosh-Smith. 2020. Tracking Performance Portability on the Yellow Brick Road to Exascale. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. <https://doi.org/10.1109/P3HPC51967.2020.00006>
- [20] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. 31, 3 (2005), 363–396.
- [21] Intel. 2021. Intel oneAPI Math Kernel Library. <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html>. Accessed: 2021-08-24.
- [22] Intel. 2023. oneAPI DPC++ compiler. <https://github.com/intel/llvm>.
- [23] Intel Corp. 2023. oneAPI GPU Optimization Guide. <https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top.html> Accessed: Aug 2023.
- [24] Bálint Joó, Thorsten Kurth, M. A. Clark, Jeongnim Kim, Christian Robert Trott, Dan Ibanez, Daniel Sunderland, and Jack Deslippe. 2019. Performance Portability of a Wilson Dslash Stencil Operator Mini-App Using Kokkos and SYCL. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 14–25. <https://doi.org/10.1109/P3HPC49587.2019.00007>
- [25] Aditya Kashi, Pratik Nayak, Dhruva Kulkarni, Aaron Scheinberg, Paul Lin, and Hartwig Anzt. 2022. Batched Sparse Iterative Solvers on GPU for the Collision Operator for Fusion Plasma Simulations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 157–167. <https://doi.org/10.1109/IPDPS53621.2022.00024>
- [26] Kim Liegeois, Sivasankaran Rajamanickam, and Luc Berger-Vergiat. 2023. Performance Portable Batched Sparse Linear Solvers. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (2023), 1524–1535. <https://doi.org/10.1109/TPDS.2023.3249110>
- [27] Kim Liegeois, Sivasankaran Rajamanickam, and Luc Berger-Vergiat. 2023. Performance Portable Batched Sparse Linear Solvers. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (May 2023), 1524–1535. <https://doi.org/10.1109/TPDS.2023.3249110>
- [28] Phuong Nguyen, Pratik Nayak, and Hartwig Anzt. 2023. Reproducibility Artifact for Ginkgo's Batched Iterative Solvers for GPUs with CUDA, HIP and SYCL Programming Models. Zenodo. <https://doi.org/10.5281/ZENODO.8247538>
- [29] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. In *ACM SIGGRAPH 2008 Classes* (Los Angeles, California) (*SIGGRAPH '08*). Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. <https://doi.org/10.1145/1401132.1401152>
- [30] NVIDIA. 2021. cuBLAS - Basic linear algebra on NVIDIA GPUs. <https://developer.nvidia.com/cublas>. Accessed: 2021-08-24.
- [31] Satoshi Ohshima, Ichitaro Yamazaki, Akihiro Ida, and Rio Yokota. 2019. Optimization of Numerous Small Dense-Matrix-Vector Multiplications in H-Matrix Arithmetic on GPU. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*. 9–16. <https://doi.org/10.1109/MCSoc.2019.00009>
- [32] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2021. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. <https://doi.org/10.1007/978-1-4842-5574-2>
- [33] Ruyman Reyes, Gordon Brown, Rod Burns, and Michael Wong. 2020. SYCL 2020: More than Meets the Eye. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) (*IWOCL '20*). Association for Computing Machinery, New York, NY, USA, Article 4, 1 pages. <https://doi.org/10.1145/3388333.3388649>
- [34] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. <https://doi.org/10.1137/1.9780898718003>
- [35] Dennis C. Smolarski, F. Douglas Swesty, and Alan C. Calder. 2022. Performance of an Astrophysical Radiation Hydrodynamics Code under Scalable Vector Extension Optimization. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 545–548. <https://doi.org/10.1109/CLUSTER51413.2022.00071>
- [36] Pedro Valero-Lara, I. Martínez-Pérez, Raúl Sirvent, X. Martorell, and Antonio J. Peña. 2018. cuThomasBatch and cuThomasVBatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 30 (2018).