Ref. Ares(2023)6624037 - 30/09/2023

Fully functioning code generator

Deliverable No: D2.4
Deliverable Title: Fully functioning code generator
Deliverable Publish Date: 30 September 2023

Project Title: SparCity: An Optimization and Co-design Framework for Sparse Computation
Call ID: H2020-JTI-EuroHPC-2019-1
Project No: 956213
Project Duration: 36 months
Project Start Date: 1 April 2021
Contact: sparcity-project-group@ku.edu.tr

List of partners:

Participant no.	Participant organisation name	Short name	Country
1 (Coordinator)	Koç University	KU	Turkey
2	Sabancı University	SU	Turkey
3	Simula Research Laboratory AS	Simula	Norway
4	Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa	INESC-ID	Portugal
5	Ludwig-Maximilians-Universität München	LMU	Germany
6	Graphcore AS*	Graphcore	Norway

*Until M21

CONTENTS

1 Introduction 1
1.1 Objectives of This Deliverable 1
1.2 Work Performed 1
1.3 Deviations and Counter Measures 2
1.4 Resources 2

2 Kernel Template Generator for Finite Element Assembly 3
2.1 Overview of finite element assembly 3
2.2 FEniCS Framework 4
2.3 Auto-Generating CUDA Kernels in FEniCS 4
2.4 Performance of assembling finite element systems on GPUs 6

3 Concurrent Graph Processing (CGP) 10
3.0.1 Design of a New CGP Framework for GPUs 10
3.1 Performance Results 12
3.2 Future Work 13

4 Conclusions 14

5 History of Changes 16

1 INTRODUCTION

The SparCity project is funded by EuroHPC JU (the European High Performance Computing Joint Undertaking) under the 2019 call of Extreme Scale Computing and Data Driven Technologies for research and innovation actions. SparCity aims to create a supercomputing framework that will provide efficient algorithms and coherent tools specifically designed for maximizing the performance and energy efficiency of sparse computations on emerging High Performance Computing (HPC) systems, while also opening up new usage areas for sparse computations in data analytics and deep learning.

Sparse computations are commonly found at the heart of many important applications, but at the same time it is challenging to achieve high performance. SparCity delivers a coherent collection of innovative algorithms and tools for enabling high efficiency of sparse computations on emerging hardware platforms. More specifically, the objectives of the project are:

- to develop a comprehensive application and data characterization mechanism for sparse computation based on the state-of-the-art analytical and machine-learning-based performance and energy models,
- to develop advanced node-level static and dynamic code optimizations designed for massive and heterogeneous parallel architectures with complex memory hierarchy for sparse computation,
- to devise topology-aware partitioning algorithms and communication optimizations to boost the efficiency of system-level parallelism,
- to create digital SuperTwins of supercomputers to evaluate and simulate what-if hardware scenarios,
- to demonstrate the effectiveness and usability of the SparCity framework by enhancing the computing scale and energy efficiency of challenging real-life applications,
- to deliver a robust, well-supported and documented SparCity framework into the hands of computational scientists, data analysts, and deep learning end-users from industry and academia.

1.1 OBJECTIVES OF THIS DELIVERABLE

The primary goal of this deliverable is to give updates on the compiler technology that produces optimized GPU kernels tailored for sparse matrices and graphs in the context of the SparCity project. This is achieved through the utilization of kernel templates and a kernel analyzer. The objective is to demonstrate the efficient code generation for sparse matrices using finite element assembly as a compelling use-case. Furthermore, the research aims to develop an automated GPU-based concurrent graph processing system capable of simultaneously executing multiple graph algorithms. This system leverages shared processing patterns and data to minimize redundant computations and data accesses, thereby enhancing overall efficiency. We will present some performance data for both the technologies.

1.2 WORK PERFORMED

Efforts in code generation for sparse computations have advanced along two distinct avenues.

First, we've introduced an automated code generator tailored to streamline finite element computations, enabling the offloading of the finite element assembly process to GPUs. Integrated

within the FEniCS framework, this generator simplifies expressing finite element computations in a high-level, mathematically intuitive manner. This initiative serves a dual purpose. It significantly enriches the repository of sparse computation kernels within the SparCity framework by accommodating a wide array of assembly kernel requirements arising from various partial differential equations and finite element approximations. To achieve this, we've introduced a kernel template generator. Additionally, by leveraging various methodologies and tools provided by SparCity for sparse computation analysis and enhancement, we can boost the performance of finite element assembly kernels on GPUs. In this reprot, we will present some of the performance results.

The second line of work centers on graph algorithms. We have deleveloped an automated algorithm fusion technique for GPUs, combining compatible graph algorithms to reduce computational overhead. This effort aims to enhance memory access patterns and execution efficiency, particularly through modifications to push and pull engines for frontier-based graph algorithms like Breadth-First Search (BFS), Page Rank, and Single Source Shortest Path. Our code generator decouples graph structure, algorithmic computation patterns, and associated properties, enabling optimization across diverse graph processing tasks. In this report, we will present its design and performance data for BFS and SSSP.

1.3 DEVIATIONS AND COUNTER MEASURES

There was no departure from the finite element assembly kernel generator. However, in the context of graph processing, we present results for both breadth-first search and single-source shortest path algorithms. In our forthcoming milestones, we intend to broaden our range of supported algorithms and incorporate additional graph-related algorithms like page ranking and Jaccard weights.

1.4 RESOURCES

The CUDA extension in FEniCS for finite element assembly (Section 2) has been archived and made available through Zenodo.[1]

The concurrent graph processing code is publicly available at:
https://github.com/sparcityeu/gpu-cgj

[1] James D. Trotter. *Software for "Targeting performance and user- friendliness: GPU-accelerated finite element computation with automated code generation in FEniCS"*. version 1.0.0. 2023. DOI: 10.5281/zenodo.7854931. URL: https://doi.org/10.5281/zenodo.7854931.

2 KERNEL TEMPLATE GENERATOR FOR FINITE ELEMENT ASSEMBLY

Finite element methods are widely used in academia and industry to numerically solve partial differential equations (PDEs). One of the most important computational steps in such methods is the *assembly* of a system of linear equations, $Ax = b$, where matrix A is sparse and typically with an unstructured sparsity pattern (due to the underlying unstructured computational mesh). The computing speed of the assembly step depends heavily on how the target hardware is utilized, thus a fitting subject for investigation in the SparCity project.

In this section, we will report the measured performance of assembling various finite element matrices and vectors that are obtained on NVIDIA V100 and A100 GPUs. This work is a continuation of Deliverable 2.1, which summarizes our recent work in extending the automated code generator of the FEniCS framework[2] with the capability of automatically generating CUDA code for offloading the assembly step to GPUs. This is essentially a GPU kernel template generator, because the target PDE and the specific finite element discretization can be freely chosen. Since finite element assembly is a mathematically involved subject, Sections 2.1-2.3 below are a repetition of the corresponding text from Deliverable 2.1. The newly obtained performance measurements of such auto-generated kernels, together with an analysis, are presented in Section 2.4.

2.1 OVERVIEW OF FINITE ELEMENT ASSEMBLY

As mentioned above, the goal of the assembly step in finite element computations is to compute a sparse matrix A and a dense right-hand side vector b, by discretizing a target PDE using finite elements. Without diving into the mathematical details, it suffices to say that A and b are computed by assembling the contributions from all the elements of a computational mesh. (Interested readers are referred to standard textbooks on finite element methods.[3]) Specifically, the contribution from each element consists of a dense element matrix $A_T \in \mathbb{R}^{n_T \times n_T}$ and element vector $b_T \in \mathbb{R}^{n_T}$, where n_T denotes the number of degrees of freedom belonging to element T. (For example, if piece-wise linear approximations are used over a triangular mesh to solve a scalar PDE in 2D, each element is a triangle and the number of degrees of freedom per element is $n_T = 3$.)

The computation within an assembly step is typically carried out as a loop over all the elements of a computational mesh, where the work per element consists of the following substeps:

1. *Gather* coordinates of the mesh entities of element T and its mapping information (from local to global degrees of freedom) from an existing data structure that contains the entire computational mesh.

2. *Compute* the element matrix and vector, A_T and b_T, based on a chosen finite element discretization of the target PDE. Here, the resulting computational kernel depends on the PDE and the chosen approximation scheme. The computational work itself is in the form of (numerical) integrals over element T.

[2] The FEniCSx computing platform. URL: https://fenicsproject.org/; Anders Logg, Kent-Andre Mardal, and Garth N. Wells (editors). *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. DOI: 10.1007/978-3-642-23099-8.

[3] Alexandre Ern and Jean-Luc Guermond. *Theory and Practice of Finite Elements*. Springer, 2004. DOI: 10.1007/978-1-4757-4355-5; Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, 2002.

3. *Scatter* the individual values of A_T & b_T and add them into the corresponding locations of the global sparse matrix A and dense vector b. Note that an unstructured computational mesh will inevitably lead to irregular memory accesses during this substep (and also the "gather" substep). Moreover, many entries in the global matrix A and vector b are the sum of contributions from multiple elements, because neighboring elements share mesh entities (such as vertices, edges and faces). Another complexity is that a compressed storage scheme of the global sparse matrix A, such as compressed sparse rows (CSR), will require an additional mapping from global row and column indices into the corresponding locations in the compressed data structure of A.

2.2 FENICS FRAMEWORK

The FEniCS framework[4] provides user friendliness through a unified form language (UFL)[5] as its high-level interface, plus behind-the-scene automated code generation. The official version of FEniCS currently only supports automated generation of MPI-parallelized CPU code.

As a very simple example, let us consider numerically solving a 2D constant-coefficient Poisson's equation $-\kappa\nabla \cdot \nabla u = f$ using piece-wise linear approximation over a triangular mesh. Algorithm 1 shows the high-level code in UFL that specifies how to discretize the target PDE (through its so-called variational form), which can be later provided to FEniCS's automated code generator to implement, among other things, the assembly step for computing the corresponding global matrix A and vector b.

```
cell = triangle
element = FiniteElement("Lagrange",cell,1)
coords = VectorElement("Lagrange",cell,1)
mesh = Mesh(coords)
V = FunctionSpace(mesh, element)
u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)
kappa = Constant(mesh)
a = kappa * inner(grad(u), grad(v))*dx
L = inner(f, v)*dx
```

Algorithm 1: Finite element discretization of Poisson's equation expressed in UFL.

The FEniCS form compiler (FFC) can automatically translate the above high-level description of a chosen finite element discretization of a target PDE into C code that implements the computation of the assembly step. For example, an auto-generated C code that is part of the computational work of substep 2 per element (see Section 2.1) is shown in Algorithm 2.

2.3 AUTO-GENERATING CUDA KERNELS IN FENICS

To enable offloading the assembly step to GPUs, we first extend the FEniCS form compiler to be able to generate CUDA-compatible code that executes the same computation as in Algorithm 2. First, the `__global__` execution space specifier is automatically annotated in front the relevant functions. Second, to make the generated code fully compatible with CUDA, various minor

[4]; Logg, Mardal, and Wells (editors), *Automated Solution of Differential Equations by the Finite Element Method*.

[5]Martin S. Alnæs et al. "Unified form language: A domain-specific language for weak formulations of partial differential equations". *ACM Trans. Math. Softw.* 40.2 (2014). ISSN: 0098-3500. DOI: 10.1145/2566630.

```
void tabulate_tensor_poisson_a(
  double* restrict A,
  const double* restrict w,
  const double* restrict c,
  const double* restrict coordinate_dofs)
{
  alignas(32) static const double weights1[1] = {0.5};
  alignas(32) static const double FE3_C0_D01_Q1[1][1][1][3]
    = {{{{-1.0,0.0,1.0}}}};
  alignas(32) static const double FE3_C0_D10_Q1[1][1][1][2]
    = {{{{-1.0,1.0}}}};
  const double J_c0 =
    coordinate_dofs[0] * FE3_C0_D10_Q1[0][0][0][0] +
    coordinate_dofs[2] * FE3_C0_D10_Q1[0][0][0][1];
  const double J_c3 =
    coordinate_dofs[1] * FE3_C0_D01_Q1[0][0][0][0] +
    coordinate_dofs[3] * FE3_C0_D01_Q1[0][0][0][1] +
    coordinate_dofs[5] * FE3_C0_D01_Q1[0][0][0][2];
  const double J_c1 =
    coordinate_dofs[0] * FE3_C0_D01_Q1[0][0][0][0] +
    coordinate_dofs[2] * FE3_C0_D01_Q1[0][0][0][1] +
    coordinate_dofs[4] * FE3_C0_D01_Q1[0][0][0][2];
  const double J_c2 =
    coordinate_dofs[1] * FE3_C0_D10_Q1[0][0][0][0] +
    coordinate_dofs[3] * FE3_C0_D10_Q1[0][0][0][1];
  alignas(32) double sp[23];
  sp[0] = J_c0 * J_c3;          sp[1] = J_c1 * J_c2;
  sp[2] = sp[0] + -1 * sp[1];   sp[3] = J_c0 / sp[2];
  sp[4] = (-1 * J_c1) / sp[2];  sp[5] = sp[3] * sp[3];
  sp[6] = sp[3] * sp[4];        sp[7] = sp[4] * sp[4];
  sp[8] = J_c3 / sp[2];         sp[9] = (-1 * J_c2) / sp[2];
  sp[10] = sp[9] * sp[9];       sp[11] = sp[8] * sp[9];
  sp[12] = sp[8] * sp[8];       sp[13] = sp[5] + sp[10];
  sp[14] = sp[6] + sp[11];      sp[15] = sp[12] + sp[7];
  sp[16] = c[0] * sp[13];       sp[17] = c[0] * sp[14];
  sp[18] = c[0] * sp[15];       sp[19] = fabs(sp[2]);
  sp[20] = sp[16] * sp[19];     sp[21] = sp[17] * sp[19];
  sp[22] = sp[18] * sp[19];
  const double fw0 = sp[22] * weights1[0];
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
      A[3*i+j] += fw0*FE3_C0_D10_Q1[0][0][0][i]*FE3_C0_D10_Q1[0][0][0][j];
  const double fw1 = sp[21] * weights1[0];
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 3; ++j)
      A[3*i+j] += fw1*FE3_C0_D10_Q1[0][0][0][i]*FE3_C0_D01_Q1[0][0][0][j];
  for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 2; ++j)
      A[3*i+j] += fw1*FE3_C0_D01_Q1[0][0][0][i]*FE3_C0_D10_Q1[0][0][0][j];
  const double fw2 = sp[20] * weights1[0];
  for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 3; ++j)
      A[3*i+j] += fw2*FE3_C0_D01_Q1[0][0][0][i]*FE3_C0_D01_Q1[0][0][0][j];
}
```

Algorithm 2: Excerpt of a CPU code that is auto-generated by the FEniCS form compiler, for computing the element matrix related to discretizing a 2D Poisson's equation using piece-wise linear approximations (3 degrees of freedom per element).

replacements are automatically enforced. For example, the C99 keyword `restrict` is replaced with the `__restrict__` keyword (for the purpose of mitigating pointer aliasing issues). Third, the CUDA extended code generator avoids external header files, which would otherwise complicate runtime code compilation that takes place later.[6] Finally, we add a suitable interface to the FEniCS form compiler, thus providing a point of access to the auto-generated CUDA code, so that it may be passed on to the runtime code compilation framework, NVRTC.[7]

Prior to offloading, the required input data must be copied from host memory to device memory, including degrees of freedom for each element, mapping info, vertex coordinates, coefficients, etc. Since transferring data between host and device can easily become a performance bottleneck, the extended FEniCS framework chooses to manage data transfers explicitly using `cudaMemcpy`. Consequently, we augment various classes and data structures in DOLFIN (the accompanying C++ library of FEniCS) to mirror and synchronise data that must be present in both host- and device-side memory.

Besides the data needed for assembly, the resulting linear system may also need to reside in GPU memory if a subsequent linear solver runs on the same device. Thus, the extended FEniCS framework internally creates a device-side pointer to the non-zero matrix values, performs assembly on the GPU, and passes the assembled matrix directly to a GPU-enabled linear solver without transferring any data to the host.

The extended FEniCS framework is now able to invoke CUDA kernels that loop over every element, and execute the auto-generated CUDA kernels in such a way that a single CUDA thread computes an entire element vector or matrix (i.e., not using multiple threads to divide the work per element). Algorithm 3 shows an automated CUDA kernel that offloads the entire assembly step of computing a global matrix A (stored in CSR format) to GPU. This kernel works for linear approximations over a triangular mesh (3 degrees of freedom per element). Note that another automated-generated CUDA kernel (`tabulate_tensor`) is called per element, which incorporates details that are specific for the target PDE.

2.4 PERFORMANCE OF ASSEMBLING FINITE ELEMENT SYSTEMS ON GPUS

To demonstrate the capabilities of the automated, CUDA-enabled code generator, we evaluate the performance of auto-generated CUDA code obtained from different PDEs and finite element discretisations. More specifically, we consider finite element methods with piece-wise linear, quadratic or cubic elements on a 3D tetrahedral, unstructured mesh for i) Poisson's equation and ii) a hyperelastic material model.[8] The source code used for the following experiments has been archived and made available through Zenodo.[9]

The unstructured mesh used in the following experiments consists of 1 255 775 vertices and 6 735 654 tetrahedra. Furthermore, for Poisson's equation, we employ elements from 1st to 3rd order, such that the number of degrees of freedom varies from about 1.2 million up to 31.5 million. The hyperelasticity model is a vector PDE with three unknowns, such that the number of degrees of freedom is 3.8 million for 1st-order elements and 24.4 million for 2nd-order elements. (Due to

[6]Ben Barsdell and Kate Clark. *Jitify: CUDA C++ Runtime Compilation Made Easy*. GPU Technology Conference 2017, San Jose, CA. NVIDIA Corporation, 2017.

[7]NVIDIA Corporation. *NVRTC – CUDA runtime compilation user guide*. NVIDIA Corporation, 2019. URL: https://docs.nvidia.com/cuda/nvrtc/.

[8]Kristian B. Ølgaard and Garth N. Wells. "Applications in solid mechanics". *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. Chap. 26, pp. 505–526. DOI: 10.1007/978-3-642-23099-8.

[9]Trotter, *Software for "Targeting performance and user- friendliness: GPU-accelerated finite element computation with automated code generation in FEniCS"*.

```
void __global__ cuda_global_assembly(
  int num_active_cells, const int * active_cells,
  const int * vertices_per_cell,
  const double * vertex_coords,
  int num_coeffs_per_cell, const double * coeffs,
  const double * constants,
  const int * dofmap0, const int * dofmap1,
  const int * rowptr,
  const int * colidx,
  double * values)
{
  for (int i=blockIdx.x*blockDim.x+threadIdx.x;
       i < num_active_cells;
       i += blockDim.x * gridDim.x) {
    // Set element matrix values to zero
    double Ae[3*3];
    for (int j = 0; j < 3*3; j++) Ae[j] = 0.0;

    // Gather cell vertex coords/coefficients
    int c = active_cells[i];
    double cell_vertex_coords[3*2];
    for (int j = 0; j < 3; j++) {
      int vertex = vertices_per_cell[c*3+j];
      for (int k = 0; k < 2; k++)
        cell_vertex_coords[j*2+k] =
          vertex_coords[vertex*2+k];
    }
    const double * cell_coeffs = &coeffs[c*num_coeffs_per_cell];

    // Compute element matrix
    tabulate_tensor(Ae, cell_coeffs, constants, cell_vertex_coords);

    // Add values to a global matrix (CSR storage)
    for (int j = 0; j < 3; j++) {
      int row = dofmap0[c*3+j];
      for (int k = 0; k < 3; k++) {
        int column = dofmap1[c*3+k];
        int r = binary_search(
          rowptr[row+1] - rowptr[row],
          &colidx[rowptr[row]], column);
        r += rowptr[row];
        atomicAdd(&values[r], Ae[j*3+k]);
      }
    }
  }
}
```

Algorithm 3: Auto-generated CUDA code for offloading the assembly step to GPU.

Variational form	matrix	DOFs	NVIDIA V100 time [s]	NVIDIA V100 performance [MDOF/s]	NVIDIA A100 time [s]	NVIDIA A100 performance [MDOF/s]
Poisson, 1st-order	4×4	1 255 775	0.0227	55.3	0.0053	238.3
Poisson, 2nd-order	10×10	9 505 341	0.1853	51.3	0.0692	137.2
Poisson, 3rd-order	20×20	31 484 353	7.7322	4.1	3.5107	9.0
Hyperelasticity, 1st-order	12×12	3 767 325	0.1931	19.5	0.0686	54.9
Hyperelasticity, 2nd-order	30×30	24 361 803	–	–	9.9460	2.45

Table 1 *Dimensions of per-element matrices, degrees of freedom (DOFs), execution time and performance of assembling matrices for different variational forms on NVIDIA V100 and A100 GPUs.*

Variational form	vector	DOFs	NVIDIA V100 time [s]	NVIDIA V100 performance [MDOF/s]	NVIDIA A100 time [s]	NVIDIA A100 performance [MDOF/s]
Poisson, 1st-order	4	1 255 775	0.00168	747.0	0.00124	1016.8
Poisson, 2nd-order	10	9 505 341	0.00409	2326.3	0.00246	3862.4
Poisson, 3rd-order	20	31 484 353	0.00820	3839.1	0.00516	6099.3
Hyperelasticity, 1st-order	12	3 767 325	0.00594	633.5	0.00292	1291.2
Hyperelasticity, 2nd-order	30	24 361 803	–	–	0.03301	738.0

Table 2 *Dimension of per-element vectors, degrees of freedom (DOFs), execution time and performance of assembling right-hand side vectors for different variational forms on NVIDIA V100 and A100 GPUs.*

memory constraints, the case of 2nd-order elements uses a slightly smaller mesh, consisting of 1 030 301 vertices and 6 000 000 tetrahedra, and was only run on NVIDIA A100.)

The extended form compiler is used to automatically generate a total of 10 kernels, where five of them are for assembling small, dense element matrices into a global sparse matrix, and the other five kernels are for assembling small, dense element vectors into a global dense vector. These kernels vary considerably in the size of element matrices/vectors that are computed, as well as the number of floating point operations and memory accesses performed. As a result, a wide range of arithmetic intensities and performance characteristics are observed. For Poisson's equation, the per-element matrices are of dimensions 4×4, 10×10 and 20×20 for 1st-, 2nd- and 3rd-order elements, respectively. The per-element matrices for the hyperelasticity model are of dimensions 12×12 and 30×30 for 1st- and 2nd-order elements, respectively.

Table 1 shows the execution time and performance of matrix assembly using the auto-generated CUDA kernels when running on NVIDIA V100 and A100 GPUs. The performance, which is reported as the number of million degrees of freedom assembled per second (MDOF/s), varies from 4.1 to 55.3 MDOF/s on NVIDIA V100 and from 2.45 to 238.3 MDOF/s on NVIDIA A100. We observe that NVIDIA A100 is about 2.2–4.3× faster than NVIDIA V100 for this type of kernels.

Table 2 shows corresponding results for assembling right-hand side vectors. In this case, performance varies from about 600 to 3800 MDOF/s on NVIDIA V100 and from 740 to 6100 MDOF/s on NVIDIA A100. Thus, for vector assembly, NVIDIA A100 is aobut 1.3–2.0× faster than NVIDIA V100. Performance differences between the two GPUs are partly explained by the NVIDIA A100's 2039 GB/s memory bandwidth, which is twice as much as the NVIDIA V100's 981 GB/s.

Variational form	matrix assembly	vector assembly
Poisson, 1st-order	4.36×	1.89×
Poisson, 2nd-order	2.11×	2.47×
Poisson, 3rd-order	1.07×	2.44×

Table 3 *Speedup after applying RCM reordering to the unstructured mesh when assembling matrices and right-hand side vectors for Poisson's equation on an NVIDIA A100 GPU.*

Like other sparse kernels considered in the SparCity framework, our auto-generated kernels for assembling finite element linear systems are vulnerable to irregular memory access patterns. However, reordering the underlying mesh entities can mitigate such issues by improving cache reuse and prefetching for the irregular memory accesses. One approach is therefore to reorder the nodes and elements of the mesh based on the bandwidth-reducing Reverse Cuthill-McKee (RCM) ordering.[10] This is one of the optimisations that were applied to obtain the results in Tables 1 and 2.

To illustrate the importance of mesh ordering, Table 3 shows the speedup on NVIDIA A100 that results from applying the RCM-based mesh reordering scheme with respect to the auto-generated assembly kernels for Poisson's equation. Due to reordering, matrix assembly with linear (1st-order) and quadratic (2nd-order) elements is 4.36× and 2.11× faster, respectively. The benefit of reordering diminishes considerably for cubic (3rd-order) elements. This is likely due to an increased arithmetic intensity or higher register pressure, meaning that memory bandwidth is no longer the primary bottleneck of the computation. For vector assembly, on the other hand, reordering maintains a speedup of more than 2.4×, even in the case of cubic elements. Both arithmetic intensity and register pressure are lower compared to matrix assembly, as per-element vectors require considerably less storage than per-element matrices. For example, a single element vector is 160 bytes for Poisson's equation with cubic elements, whereas the corresponding element matrix is 3200 bytes.

[10] James D. Trotter, Xing Cai, and Simon W. Funke. "On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core CPUs". *ACM Transactions on Mathematical Software* 48.2 (2022). ISSN: 0098-3500. DOI: 10.1145/3503925. URL: https://doi.org/10.1145/3503925.

3 CONCURRENT GRAPH PROCESSING (CGP)

In recent years, large-scale graph-based models have found applications across various domains, including scientific computing, social networks, machine learning, and biology. Optimizing graph algorithm performance is crucial for the efficiency of these applications. While many graph processing systems have been proposed and explored, there remains a need for a GPU-based concurrent graph processing system capable of efficiently handling multiple graph algorithms simultaneously.

To address these needs, SparCity has established essential GPU data structures and implemented the data and computation integration for various graph algorithms, including breadth-first search (BFS) and BellmanFord (SSSP). Our experiments have shown that concurrent graph processing reduces execution times for these algorithms.In this phase of our work, our primary focus on the implementation details of kernel code generation and its performance results on BFS and SSP.

3.0.1 DESIGN OF A NEW CGP FRAMEWORK FOR GPUS

A key feature in our framework design was decoupling computation and scheduling data structures. This design choice made it easy to decouple host (CPU) functions and device (GPU) ones. For example, the function for job initialization runs on the CPU, while that for update runs on the GPU.

We developed a Concurrent Graph Job (CGJ) system for the GPU inspired by Krill's scheduling approach. Despite migrating the heavy computations to the GPU, the CPU could retain its role in scheduling jobs. This hybrid approach leverages the strengths of both the CPU and GPU. While the GPU excels at parallel computations, the CPU's role in managing tasks and scheduling ensures optimal resource allocation and orchestration of the computational workflow.

Domain-specific languages (DSLs) and the concept of algorithm decoupling have emerged as notable trends in the research landscape. Halide,[11] TVM,[12] GraphIt,[13] Taichi,[14] and HeteroCL[15] exemplify the concept of isolating computation from scheduling, resulting in heightened performance and adaptability across diverse domains, including image processing and deep learning. Inspired by the principles of domain-specific languages (DSLs) and algorithm decoupling, we also adopt this approach to enhance both performance and flexibility. The notion of kernel fusion, employed by our framework, optimizes diverse concurrent graph jobs on GPUs, setting it apart from prior efforts.

We employed CUDA in our GPU CGJ implementation. CUDA stands out from other GPU platforms due to its reputation of being robust and mature, underpinned by years of development

[11] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". *SIGPLAN Not.* 48.6 (2013), pp. 519–530. ISSN: 0362-1340. DOI: 10.1145/2499370.2462176. URL: https://doi.org/10.1145/2499370.2462176.

[12] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594.

[13] Yunming Zhang et al. "GraphIt: A High-Performance Graph DSL". *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276491. URL: https://doi.org/10.1145/3276491.

[14] Yuanming Hu et al. "Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures". *ACM Trans. Graph.* 38.6 (2019). ISSN: 0730-0301. DOI: 10.1145/3355089.3356506. URL: https://doi.org/10.1145/3355089.3356506.

[15] Yi-Hsiang Lai et al. "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing". *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 242–251. DOI: 10.1145/3289602.3293910. URL: https://doi.org/10.1145/3289602.3293910.

and refinement. Additionally, the widespread adoption of CUDA in research circles underscores its credibility and reliability. This broad utilization signifies its effectiveness in addressing intricate computational challenges. The presence of a large and vibrant community dedicated to CUDA provides a wealth of resources, support, and expertise, ensuring we have the necessary tools to navigate challenges and optimize our implementation.

Like Krill, our implementation has components responsible for graph loading, scheduling, and executing jobs.

- **Graph Loader:** The graph loading component loads graphs stored in CSR format (we might support others in the future) to the device memory. It starts by allocating the needed buffers on the host and on the device. Then, it loads the graph data to the host memory first. Finally, it copies the graph data from the host buffer to the device buffer. The GPU memory hierarchy is suitable for accelerating operations that fit into the streaming programming model rather than general serial computation. Therefore, we used array-like data structures for graph data.
- **Scheduler:** Our Krill-inspired scheduler component takes a set of jobs and sends them to the GPU for execution. It monitors which jobs done and which are not and keeps running iterations on the GPU until all jobs finish. However, due to GPU restrictions, our scheduler only supports homogenous jobs at this point.

```
while (true) {
        copyJobCompletionStatusFromGPU();
        if (allJobsDone) break;
        gpuExecute<<<gridSize, blockSize>>>(graph_data, job_data);
        cudaDeviceSynchronize();
}
```

Algorithm 4: Simplified Scheduler Logic

- **Executor:**

  Our executor component is fundamentally different from Krill's. Krill performs well on CPUs, which are serial processors. However, GPUs are streaming processors with more restrictions on kernels and memory access patterns. We use GPU-based data structures for the graph, vertex, and frontier data. For example, we use arrays as dense frontiers because GPUs are not good with dynamic sparse queues.

```
__global__ void gpuExecute(graph, jobs) {
    // Get our global thread ID
    auto id = getThreadId();

    if (id >= graph.size) return;

    for (auto i = 0; i < jobs.length; i++) {
            // Skip finished jobs
            if (jobs[i].done) continue;
            // Skip vertices not in the frontier
            if (!jobs[i].frontier[id]) continue;
            jobs[i].kernel(graph, id);
    }
}
```

Algorithm 5: Simplified Executor Logic

Our executor can only execute homogeneous jobs due to GPU abstraction limitations. We've implemented BFS and Bellman-Ford SSSP and used them for performance testing.

3.1 PERFORMANCE RESULTS

Figure 1 and 2 depict the execution times of a single graph algorithm, multiple executions of the same algorithm, and concurrent executions of the same algorithm for BFS and SSSP, respectively. As indicated in the figures, our auto-generated code closely resembles that of multiple runs. In future work, our focus will be on enhancing the performance of the generated code.

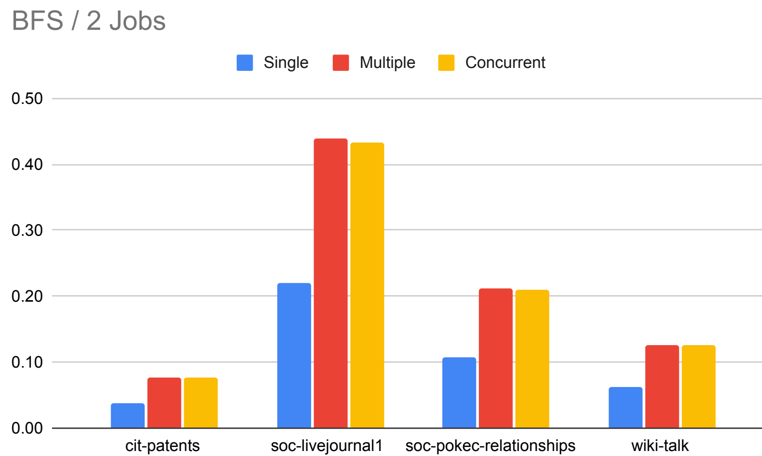

Figure 1 *Execution times of single BFS, multiple BFS(s) and concurrent BFS(s) using our generator for four different grafts*

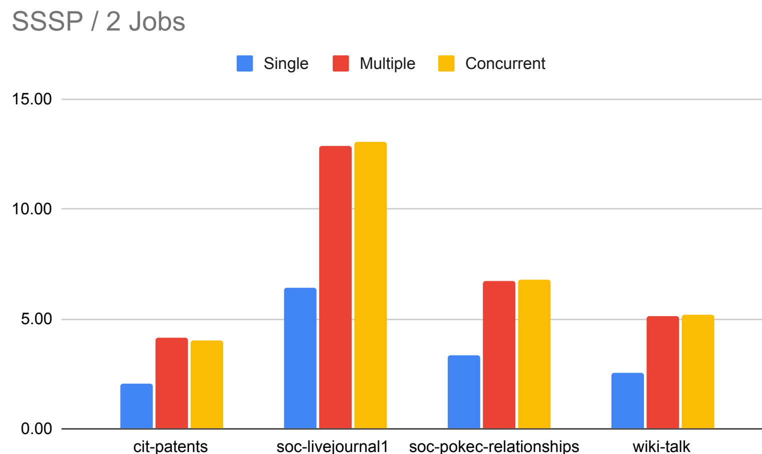

Figure 2 *Execution times of single SSSP, multiple SSSP(s) and concurrent SSSP(s) using our generator for four different grafts*

3.2 FUTURE WORK

This initial implementation is a prototype that supports homogeneous BFS and SSSP kernels due to GPU restrictions on abstraction. It also does not improve the performance considerably compared to separate graph jobs. And it does not allow real-time job arrival. Users should specify all jobs before running the program.

However, we will expand our work in the future to allow arbitrary kernels and property data using code generation techniques. Moreover, we will perform various performance optimizations by profiling the code and spotting the bottlenecks. We will support real-time job arrival by adopting a client-server architecture. The server will be a long-running process that accepts jobs from different clients at runtime.

4 CONCLUSIONS

Firstly, we designed an automated code generator tailored for finite element computations, enabling the transition of the finite element assembly process to GPUs. This seamless integration within the FEniCS framework allows for an intuitive high-level expression of finite element computations, closely mirroring mathematical formulations. Our future focus includes handling scalar and vector PDEs in both 2D and 3D contexts, accommodating variable coefficients, and diverse approximation options like piece-wise linear or quadratic. These automatically generated kernels will be executed on various computational meshes for analysis. The goal is to optimize and enhance the performance of these assembly kernels using methodologies and tools provided by SparCity.

The second approach revolves around graph algorithms. We are actively developing an automated algorithm fusion technique for GPUs, aiming to combine compatible graph algorithms to minimize computational overhead. This effort seeks to improve memory access patterns and execution efficiency. Our code generator decouples graph structure, algorithmic computation patterns, and associated properties, facilitating optimization across various graph processing tasks. By the end of the project, we will present the final design of the generator, optimization details, and performance of both homogeneous and heterogeneous job schedulers.

REFERENCES

Alnæs, Martin S. et al. "Unified form language: A domain-specific language for weak formulations of partial differential equations". *ACM Trans. Math. Softw.* 40.2 (2014). ISSN: 0098-3500. DOI: 10.1145/2566630.

Barsdell, Ben and Kate Clark. *Jitify: CUDA C++ Runtime Compilation Made Easy*. GPU Technology Conference 2017, San Jose, CA. NVIDIA Corporation, 2017.

Chen, Tianqi et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594.

Ciarlet, Philippe G. *The Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, 2002.

Ern, Alexandre and Jean-Luc Guermond. *Theory and Practice of Finite Elements*. Springer, 2004. DOI: 10.1007/978-1-4757-4355-5.

The FEniCSx computing platform. URL: https://fenicsproject.org/.

Hu, Yuanming et al. "Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures". *ACM Trans. Graph.* 38.6 (2019). ISSN: 0730-0301. DOI: 10.1145/3355089.3356506. URL: https://doi.org/10.1145/3355089.3356506.

Lai, Yi-Hsiang et al. "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing". *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 242–251. DOI: 10.1145/3289602.3293910. URL: https://doi.org/10.1145/3289602.3293910.

Logg, Anders, Kent-Andre Mardal, and Garth N. Wells (editors). *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. DOI: 10.1007/978-3-642-23099-8.

NVIDIA Corporation. *NVRTC – CUDA runtime compilation user guide*. NVIDIA Corporation, 2019. URL: https://docs.nvidia.com/cuda/nvrtc/.

Ølgaard, Kristian B. and Garth N. Wells. "Applications in solid mechanics". *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. Chap. 26, pp. 505–526. DOI: 10.1007/978-3-642-23099-8.

Ragan-Kelley, Jonathan et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". *SIGPLAN Not.* 48.6 (2013), pp. 519–530. ISSN: 0362-1340. DOI: 10.1145/2499370.2462176. URL: https://doi.org/10.1145/2499370.2462176.

Trotter, James D. *Software for "Targeting performance and user- friendliness: GPU-accelerated finite element computation with automated code generation in FEniCS"*. Version 1.0.0. 2023. DOI: 10.5281/zenodo.7854931. URL: https://doi.org/10.5281/zenodo.7854931.

Trotter, James D., Xing Cai, and Simon W. Funke. "On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core CPUs". *ACM Transactions on Mathematical Software* 48.2 (2022). ISSN: 0098-3500. DOI: 10.1145/3503925. URL: https://doi.org/10.1145/3503925.

Zhang, Yunming et al. "GraphIt: A High-Performance Graph DSL". *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276491. URL: https://doi.org/10.1145/3276491.

5 HISTORY OF CHANGES

Version	Author(s)	Date	Comment
0.1	Didem Unat	07.09.2023	Initial draft
0.2	James Trotter	25.09.2023	Improved draft for Section 2
0.3	Xing Cai	26.09.2023	Improved draft for Section 2
0.3	Ameer Taweel	27.09.2023	Improved draft for Section 3
0.5	Didem Unat	30.09.2023	Finalizing the content for submission

Table 4 *Document History of Changes*