Prototype of kernel template generator and kernel analyzer

Deliverable No: D2.1
Deliverable Title: Prototype of kernel template generator and kernel analyzer
Deliverable Publish Date: 31 August 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

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 main objective of this work is to develop compiler technology that generates optimized GPU kernels for sparse matrices and graphs by using kernel templates and kernel analyzer. The task aims to show finite element assembly as a use-case for efficient code generation for sparse matrices. In addition, it aims to create an automated GPU-based concurrent graph processing system capable of simultaneously handling multiple graph algorithms, leveraging shared processing patterns and data to reduce redundant computations and data accesses.

1.2 WORK PERFORMED

The efforts related to code generation for sparse computation were undertaken through two distinct lines of work.

Firstly, we have created an automated code generator tailored to handle finite element computations, allowing for the offloading of the finite element assembly step onto GPUs. This generator

is seamlessly integrated within the FEniCS framework, facilitating the expression of finite element computations in a way that is both high-level and intuitive, closely resembling mathematical formulations.

This endeavor serves a twofold purpose. Firstly, it significantly enriches the repertoire of sparse computation kernels within the SparCity framework. This achievement comes from catering to the diverse assembly kernel requirements stemming from various partial differential equations and finite element approximations. To accomplish this, we have introduced a kernel template generator. Secondly, by leveraging numerous methodologies and tools offered by SparCity for analyzing and improving sparse computations, we can enhance the performance of finite element assembly kernels on GPUs.

The second line of work focuses onto graph algorithms. We are developing an automated algorithm fusion technique for GPUs. This approach involves combining compatible graph algorithms to minimize computational overhead. This endeavor is geared towards enhancing memory access patterns and execution efficiency. We are achieving this through strategies like modifying push and pull engines for frontier-based graph algorithms such as 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 jobs.

1.3 DEVIATIONS AND COUNTER MEASURES

There was no deviation from the finite element assembly kernel generator. However, for the graph processing part, we initially selected Krill[1] - a graph processing framework designed for CPUs - to implement our code generator for GPU-based concurrent graph processing. However, we encountered compatibility issues with Krill's data structures on GPUs. Adapting it necessitated extensive code refactoring and debugging efforts. In response, we have taken the initiative to develop a new framework from the ground up, drawing inspiration from Krill's concepts.

Furthermore, because the individual who was initially working on this task is currently on maternity leave, we have enlisted another team member to take over the responsibilities. This transition took place in June 2023. While these developments have led to delays in completing the code generator, we remain optimistic. We are confident that by the time of the next deliverable, we will have a functional prototype of the kernel code generator ready.

1.4 RESOURCES

The CUDA extension in FEniCS for finite element assembly is not publicly available yet. Its related publication is currently under review, and will publish the CUDA code of the FEniCS extension when the paper is accepted.

[1] **empty citation**.

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 summarize our recent work in extending the FEniCS framework,[2] in particular its automated code generator that implements finite element computations, with the capability of automatically generating CUDA code for offloading the assembly step to GPUs. The result is that future FEniCS users can continue to express their finite element computations in a high-level, user-friendly and mathematics-resembling manner, independent of the target hardware platform, whereas the assembly step can be seamlessly executed on GPUs. On one side, this work greatly enriches SparCity's collection of sparse computation kernels, because different PDEs and finite element approximations give rise to different assembly kernels. We have thus provided a kernel template generator. On the other side, several of SparCity's methodologies and tools for analyzing/optimizing sparse computations can be used to improve the GPU performance of finite element assembly kernels.

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]

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 FUTURE WORK

In the near future, a number of assembly kernels will be auto-generated by the extended FEniCS framework. We will examine scalar and vector PDEs in 2D/3D with or without variable coefficients, together with a few approximation choices (e.g., piece-wise linear or quadratic). These auto-generated kernels will be run using a representative set of computational meshes. The purpose is to analyze and enhance the performance of the assembly kernels with the help of SparCity's methodologies and tools.

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

```
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.

3 CONCURRENT GRAPH PROCESSING (CGP)

In recent years, graph-based models with vast numbers of vertices have been employed to address a variety of real-world challenges, spanning scientific computations, social networks, machine learning, and biology. Enhancing the performance of graph algorithms is vital for optimizing the efficiency of these applications. However, creating high-performance graph algorithms is complex. Consequently, various graph processing applications have been proposed and extensively explored in recent times [10-19].

Existing graph processing systems can be categorized into several groups. These include CPU-based single graph processing systems, GPU-based single graph processing systems, CPU- based concurrent graph processing systems. Each category focuses on distinct aspects of graph processing, aiming to optimize performance for a specific type of scenario. However, there is a need for a GPU-based concurrent graph processing system capable of managing multiple graph algorithms simultaneously, leveraging shared underlying patterns and data to minimize redundant computations and data accesses. To achieve these goals, SparCity set up crucial GPU data structures to support these efforts. We have started manually implementing data and computation integration for concurrent processing of a number of graph algorithms. These include breadth-first-search (BFS), Jaccard (JC), PageRank (PR), and BellmanFord (SSSP). BFS is bandwidth-conserving algorithms and the last three are memory bandwidth-consuming algorithms. We have designed six job sets. The experiment results of our manual integration of these algorithms show that the concurrent graph processing can greatly reduce the execution time of these algorithms. These findings are reported in the previous deliverables. As we transition into the subsequent phase, it's essential to clarify that in this specific step of our work, our efforts have been directed towards the actual implementation of kernel code generation and automation of this process on the GPU architecture.

In the next section, we first give background on concurrent graph processing and then discuss our failed attempts to extend Krill to leverage GPU acceleration along with challenges encountered. Finally we discuss our new design for GPU code generation. In the next period, we will focus on the performance evaluation and improvements, results on a large set of graphs.

3.1 BACKGROUND ON CGP

In recent times, Concurrent Graph Processing (CGP) has emerged as a response to the demand for effectively managing multiple graph algorithms running concurrently. This need arises in various scenarios, including cloud-based graph systems catering to user queries and large-scale platforms executing multiple jobs on the same graph. Unlike traditional approaches that emphasize vertex-level parallelism, CGP introduces job-level parallelism, where numerous graph jobs are executed concurrently. This shift presents both challenges and opportunities for optimization.

Concurrent graph processing systems have been developed to address the complexities of running multiple graph algorithms concurrently. One basic approach involves using traditional single graph processing systems to manage multiple jobs, leading to redundant memory consumption due to graph structure copying for each job. However, more sophisticated CGP systems aim to efficiently utilize shared graph structures. Seraph,[7] for instance, introduces a Graph-Exchange-State model that separates the graph structure from graph jobs, enabling the reuse of

[7] Jilong Xue et al. "Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing". *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 227–238. DOI: 10.1145/2600212.2600222. URL: https://doi.org/10.1145/2600212.2600222.

the underlying graph for various concurrent jobs. While this approach improves resource sharing, it introduces data access overheads due to divergent access paths.

CGraph[8] and GraphM take more refined approaches by partitioning the graph at the cache level, prioritizing spatial and temporal locality for different jobs. GraphM further enhances performance by using a scheduler to manage loading orders for graph chunks, reducing cache thrashing overheads by synchronizing job progress.

Krill[9] presents an innovative SAP (Structure-Algorithm-Property) model that disentangles the graph's structure, algorithms, and property data. It comprises a high-level compiler and a graph runtime. The compiler generates dynamic libraries for runtime execution based on user-defined algorithm and property description files. This design empowers users to focus on algorithm implementation rather than low-level details. For algorithm description, users define the "cond" and "update" functions, determining vertex visitation and property value calculation in each iteration. Batch job submission is possible by declaring multiple jobs in the algorithm description file. The property description is facilitated through a lightweight programming interface called the "property buffer." By writing a few lines of code in this file, users can specify property values for each job.

The property buffer compiler frontend processes the property description, transforming data layout, while the backend generates a property package. The generated code, in the form of a C++ header file, contains classes for automatic property data management and communication with the graph server. APIs like "get" and "set" are provided for efficient property data access. Krill's runtime system introduces "graph kernel fusion," which combines different jobs to address memory access issues. The server runtime includes a graph store, fetcher, scheduler, and executor as shown in Figure 1. By fusing ongoing and new jobs, Krill achieves concurrent execution while maintaining job boundaries for privacy and fault tolerance. Importantly, Krill streamlines property declaration at compile-time, enhancing performance without server-side recompilation.

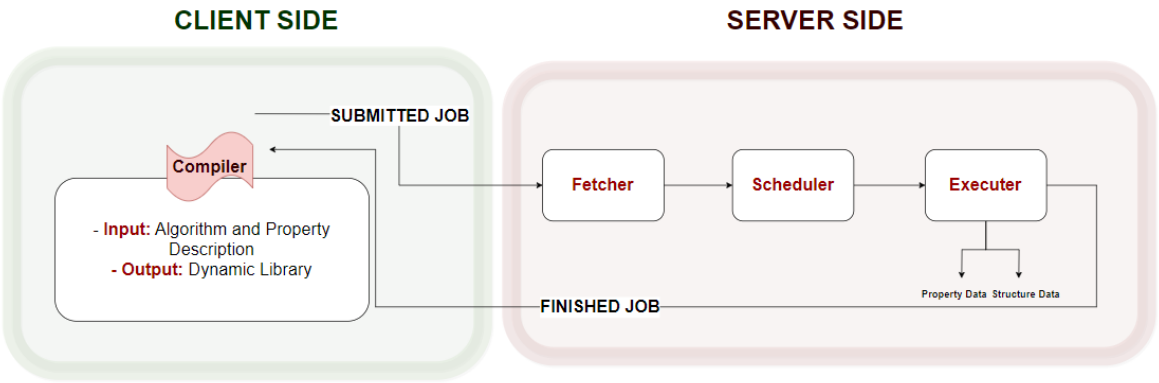

Figure 1 *Krill Overview*

[8]Yu Zhang et al. "CGraph: A Distributed Storage and Processing System for Concurrent Iterative Graph Analysis Jobs". *ACM Trans. Storage* 15.2 (2019). ISSN: 1553-3077. DOI: 10.1145/3319406. URL: https://doi.org/10.1145/3319406.

[9]Hongzheng Chen et al. "Krill: A Compiler and Runtime System for Concurrent Graph Processing". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis, Missouri: Association for Computing Machinery, 2021. DOI: 10.1145/3458817.3476159. URL: https://doi.org/10.1145/3458817.3476159.

3.2 AUTOMATING CGP ON GPU

Krill demonstrates its potential to efficiently manage concurrent graph jobs on multicore. However, with the exponential data growth, there is a demand for faster computation and processing on GPUs. The transition to GPU acceleration promises to unlock a new level of performance, enabling concurrent graph jobs to be executed concurrently on the GPU's multitude of cores. Our initial plan was to extend Krill to leverage GPU acceleration. To fully harness the potential of GPU acceleration, a crucial step was to migrate the core data essential for graph jobs to the GPU's memory. This data includes graph, vertex, property, and frontier data. By moving this data to the GPU, we could modify Krill to accelerate operations on the GPU.

We planed to transform the functions responsible for computations into device functions in Krill. By making these functions compatible with GPU execution, we could enable them to run directly on the GPU, effectively offloading a significant portion of the computation from the CPU. However, we have encountered several issues in adapting Krill to GPUs, which are discussed next.

3.2.1 CHALLENGES IN ADAPTING KRILL FOR GPUS

Adapting the Krill codebase to run on the GPU demands careful consideration due to its intricately nested data structures and intertwined functions. This exploration delves into the complexities of this task, highlighting the trade-offs between modifying the existing codebase and starting a new one.

The developers of Krill originally designed it to excel in a CPU-centric environment. This design poses multiple challenges for migrating the code to run on the GPU:

- Data Structure Complexity: The deeply nested data structures that facilitate efficient CPU execution can pose a considerable challenge when moving to the GPU. The design choices optimized for CPUs might not align seamlessly with GPU architecture, necessitating intricate modifications.
- Interwoven Functions: The interdependencies between functions that contribute to Krill's effectiveness on the CPU introduce complexities when aiming to parallelize computation. We must decouple those functions to enable effective parallelism, which might entail a significant restructuring effort.
- Memory Hierarchies: GPUs have different memory hierarchies compared to CPUs. Adapting Krill's memory management to suit GPU memory access patterns is a non-trivial task that requires careful consideration of data movement and access patterns.
- Algorithmic Adaptation: Certain algorithms that work well on CPUs might require substantial modification to exploit GPU parallelism.

Given the complexities of migrating Krill to GPU processing, starting from a fresh codebase warrants exploration. Thus we have departed from Krill as our baseline and decided to develop our own framework.

3.2.2 DESIGN OF A NEW CGP FRAMEWORK FOR GPUS

Creating a new codebase designed with GPU parallelism in mind could offer several advantages:

- Optimized Structure: A new codebase can be structured to inherently exploit GPU parallelism and memory hierarchies, leading to more efficient execution.

- Reduced Complexity: Starting afresh enables shedding the legacy intricacies of CPU-oriented designs, potentially leading to cleaner, more maintainable code.
- Learning from Experience: We can leverage lessons from the original Krill design to make informed decisions and avoid pitfalls.

A key feature in our framework design will be decoupling computation and scheduling data structures. This design choice will make it easy to decouple host (CPU) functions and device (GPU) ones. We will develop a Concurrent Graph Job (CGJ) system for the GPU. However, we will take inspiration from 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,[10] TVM,[11] GraphIt,[12] Taichi,[13] and HeteroCL[14] 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 decided to employ 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 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 potential challenges and optimize our implementation.

Like Krill, our implementation will have components responsible for graph loading, scheduling, executing, and property data. However, we aim to have much more decoupled components than Krill.

The graph loading component will load graphs stored in CSR format (and possibly other ones) to the device memory. This component will allocate the needed buffers on the host and on the device. Then, it will move the graph data to the GPU memory. The GPU memory hierarchy

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

is suitable for accelerating operations that fit into the streaming programming model rather than general serial computation. Therefore, we will use array-like data structures as much as possible.

We will adapt the scheduling component from Krill. The scheduler will fuse kernels of the different graph jobs and send them to the executor. We will initially support a predefined set of job kernels. However, we might add the ability to define arbitrary kernels using code generation (compiler) techniques later.

Our executor component will be 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 will use GPU-based data structures for the graph, vertex, and frontier data.

Finally, we will have a component to manage property data. We will initially have a predefined set of property data for the supported kernel types. However, we will support arbitrary property data via code-generation techniques.

We expect that our framework will significantly reduce memory access requirements for Concurrent Graph Jobs running on GPUs, achieving substantial memory reduction and improved performance compared to existing systems.

3.3 FUTURE WORK

The initial implementation will be a prototype supporting a fixed set of kernels and property data. It might also have some performance issues. And it will 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

Efforts in this task were pursued along two distinct avenues to generate kernel codes. Firstly, an automated code generator was designed specifically for finite element computations, enabling the transfer of the finite element assembly process to GPUs. This integration seamlessly fits within the FEniCS framework, allowing for an intuitive high-level expression of finite element computations, closely resembling mathematical formulations. Future work's focus will encompass scalar and vector PDEs in both 2D and 3D settings, 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 intention is to optimize and boost the performance of these assembly kernels utilizing the methodologies and tools provided by SparCity.

The second avenue of work centers around graph algorithms. An automated algorithm fusion technique for GPUs is under development. This approach involves the amalgamation of compatible graph algorithms to minimize computational overhead. The objective is to enhance memory access patterns and execution efficiency. The code generator decouples graph structure, algorithmic computation patterns, and associated properties, enabling optimization across various graph processing tasks. In the next report, we will present our final design, implementation details and performance of the newly designed framework.

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, Hongzheng et al. "Krill: A Compiler and Runtime System for Concurrent Graph Processing". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis, Missouri: Association for Computing Machinery, 2021. DOI: 10.1145/3458817.3476159. URL: https://doi.org/10.1145/3458817.3476159.

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.

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.

Xue, Jilong et al. "Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing". *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 227–238. DOI: 10.1145/2600212.2600222. URL: https://doi.org/10.1145/2600212.2600222.

Zhang, Yu et al. "CGraph: A Distributed Storage and Processing System for Concurrent Iterative Graph Analysis Jobs". *ACM Trans. Storage* 15.2 (2019). ISSN: 1553-3077. DOI: 10.1145/3319406. URL: https://doi.org/10.1145/3319406.

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	01.08.2023	Initial draft
0.2	Xing Cai	25.08.2023	Improved draft for Section 2
0.3	Beyza Cavusoglu	28.08.2023	Improved draft for Section 3
0.4	Ameer Taweel	29.08.2023	Improved draft for Section 3
0.5	Didem Unat	31.08.2023	Finalizing the content for submission

Table 1 *Document History of Changes*