Automated Code Generation for GPU-Based Finite Element Computations in FEniCS

James D. Trotter, Johannes Langguth, Xing Cai

Simula Research Laboratory, Norway

Introduction

FEniCS is a popular finite element frame-work for solving PDEs that uses automated code generation, where a domain-specific compiler, called the FEniCS Form Compiler (FFC), is used to generate low-level kernels from high-level descriptions of finite element problems. These auto-generated kernels calculate problem-dependent element vectors and matrices, which are then used to assemble linear systems that are to be solved.

In this work, we enable GPU acceleration of the linear system assembly procedure in FEniCS and realize a seamless integration with GPU-accelerated linear solvers.

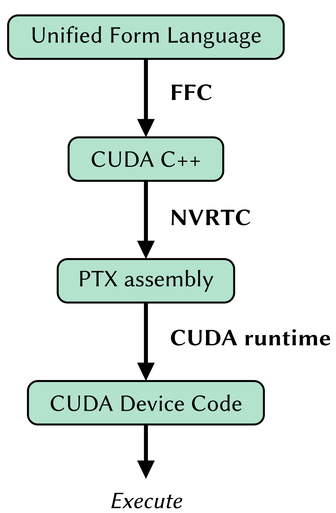

Fig. 1. Stages of translating and compiling FEniCS's Unified Form Language (UFL) to CUDA Device Code that is executed on a GPU. CUDA C++ code is first generated by FFC, then compiled to PTX assembly by the NVIDIA Runtime Compiler (NVRTC).

Methods

- We modify the FEniCS-X Form Compiler (FFCX) to generate CUDA C++ kernels for computing element matrices and vectors.
- We add CUDA C++ kernels to DOLFINX, the part of FEniCS that sits atop FFCX, to support GPU-accelerated linear system assembly.
- We compare two assembly approaches:
  a) *GPU-based local assembly*—element matrix calculation on GPU, but global assembly still done by the CPU.
  b) *GPU-based global assembly*—element matrix calculation and global assembly on GPU.
- We investigate the cost of transferring global matrix values between CPU and GPU.

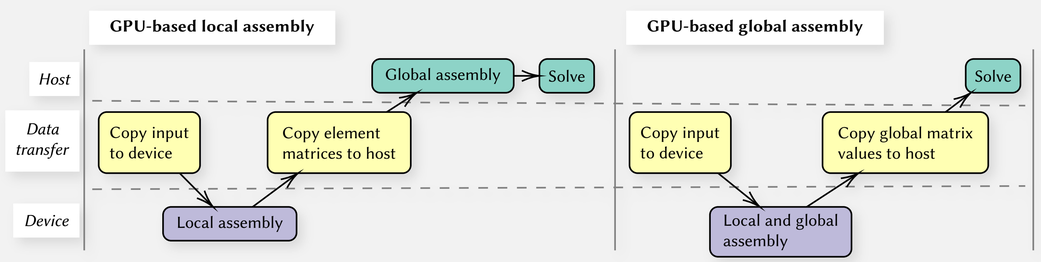

Fig. 2. Host (CPU) and device (GPU) operations, as well as memory transfers for the local and global approaches to offloading finite element assembly.

```
element = FiniteElement("Lagrange",tetrahedron ,1)
coords = VectorElement("Lagrange",tetrahedron ,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
```

Fig. 3. Unified Form Language (UFL) code for Poisson's equation.

Fig. 4. Unstructured mesh, *Cardiac mesh 20,* made up of 16.9 million tetrahedra.

Results

- GPU-based local assembly performs poorly due to large CPU-GPU data transfers, and global assembly on the CPU beomes a bottleneck.
- Offloading global assembly to GPU yields a 8–20x speedup over FEniCS's CPU-based, MPI-parallel assembly on dual-socket AMD Epyc 7601 CPUs (64 cores).
- A lookup table can be used instead of binary searches for finding matrix entries of the global, sparse matrix, further accelerating GPU assembly.
- The cost of CPU-GPU data transfers is significant, even when global assembly is done on the GPU. A GPU-based linear solver avoids transfers by keeping the linear system on the GPU.

Table 1. Performance (in Mcell/s) of assembling a matrix for the Poisson problem on NVIDIA V100.

Mesh	CPU	GPU Local	GPU Global	GPU Global w/ Lookup-table
Uniform mesh	58.4	3.7	1100.5	1336.9
Cardiac mesh 1	56.1	2.1	1232.5	1183.1
Cardiac mesh 20	58.0	2.1	582.5	996.6
Cardiac mesh 41	58.7	1.9	469.7	909.8
Cardiac mesh 44	56.1	2.0	533.7	983.6

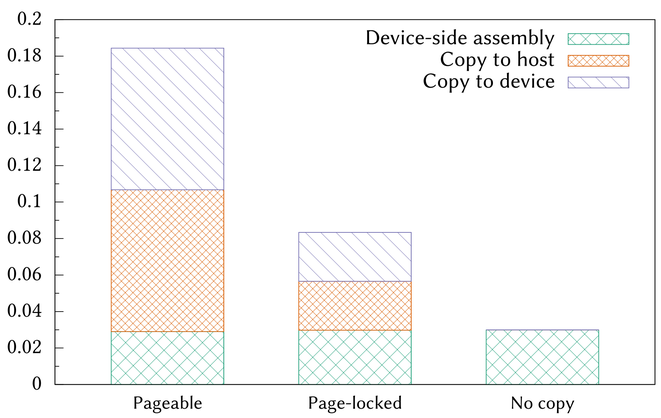

Fig. 5. Time (in seconds) spent in assembly and copying data between GPU device memory and main memory (DRAM) using pageable or page-locked memory for the Poisson problem with *Cardiac mesh 20* on NVIDIA V100.

Acknowledgements

This work is supported by the Research Council of Norway project *Meeting Exascale Computing with Source-to-Source Compilers* (251186) and *Experimental Infrastructure for Exploration of Exascale Computing (eX3)* (270053).

simula