# Finite Element Integration on GPUs

Matthew G. Knepley Andy R. Terrel

June 19, 2012

This work was sponsered by NSF through awards OCI-0850680 and OCI-0850750. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

**Abstract**

We present a novel finite element integration method for low order elements on GPUs. We achieve more than 100GF for element integration on first order discretizations of both the Laplacian and Elasticity operators on an NVIDIA GTX285, which has a nominal single precision peak flop rate of 1 TF/s and bandwidth of 159 GB/s, corresponding to a bandwidth limited peak of 40 GF/s.

## 1 Introduction

Graphical Processing Units (GPUs) present a promising platform for scientific simulation, offering high performance with excellent power and cost efficiency. However, despite advances in programmability for these devices [17], few numerical libraries have made use of them. The challenge of rewriting a CPU code to make use of a GPU's architectural differences is a major barrier, which can lead to slower code. As a result, high level simulation algorithms, finite elements methods (FEM) in particular, are still not widely available.

In this paper, we summarize our experience with porting general FEM integration routines from the popular FEniCS project [13] to a GPU. By adjusting the code generation tools available from FEniCS, a user is able to reuse their high level weak form definition in both a CPU or GPU code. Using our decomposition of global and local portions of the FEM integration routines, our port is able to reach up to 100 GFlops on a single machine where highly-optimized CPU codes, including hand-coded assembly routines, only reach the 12 GFlop range [3]. By creating tools that allow researchers to leverage a GPU's power throughout their code, the GPU becomes an enabler of scientific discovery rather than a limited tool for only a few codes.

We give an overview of available GPU codes for scientific computing in section 2 discussing general tactics for speeding up a code with a GPU version. For completeness, we review the tensor decomposition of FEM integration and

the available form languages available from the FEniCS project in section 3. Our GPU port is described in section 4 with the numerical tests and results in section 5.

## 2 Scientific GPU Codes

Several community packages are available for basic linear algebra, such as CUBLAS [18] for the dense case and Thrust [**?**], CUSP [**?**], and CUDASparse [19] for the sparse case. While there has been excellent work bringing high order methods to the GPU, discontinuous Galerkin [11] and spectral elements [12], very little has focused on the low-order methods which make up the majority of finite element codes. Initial work in this area comes from [14], but in this paper we focus on optimizing the integration stage. [22] implemented assembly of a linear element for elasticity in three dimensions, but was not able to eliminate branches from the kernel, and used graphics primitives rather than CUDA as the implementation language. [4] also implements finite element assembly, but they consider the element subroutine as a black-box to be executed in its entirety by each thread. Solvers for FEM problem have been examined as well, e.g. [23], and could potentially benefit from this work. Tools for runtime code generation and optimization are detailed in [10], which we will make use of in our study.

There are many excellent descriptions of the NVIDIA GPU architecture in the literature [15, 5, 17], so we will focus on the aspects salient to our problem. GPUs can be characterized as a collection of small vector units which run in single-instruction multiple-thread (SIMT) mode. In the GTX285 model from NVIDIA on which we run our tests, the vector length is 8 and there are 30 of these Streaming MultiProcessors (SM), as the vector units are called, clocked at 1476 MHz. These allow for a memory bandwidth of 159 GB/s (STREAMS benchmark performance of 130 GB/s) and peak flop rate of 692 GF/s. In our integration implementation, we must allow enough concurrency to feed these vector units, while minimizing thread divergence and synchronization, which have large penalties on this SIMT processor. Moreover, in all GPU architectures there is a very large latency to global memory (400-600 cycles on the GTX285), as opposed to the shared and register memory co-located with the SM which cost just a few cycles. Therefore, we also minimize traffic to global memory by loading input into shared memory and storing intermediate results for aggregation.

## 3 FEM Integration

In [7], it is shown that for any given multilinear weak form of arity $r$, we may express the element tensor as a tensor contraction,

$$E^{i_0,\ldots,i_r} = \sum_{\mu_0,\ldots,\mu_g} G^{\mu_0,\ldots,\mu_g} K^{i_0,\ldots,i_r}_{\mu_0,\ldots,\mu_g}. \tag{1}$$

We note that the element matrix is called $A$ in [7]. The tensor $K$ only depends on the form itself and the reference element $\mathcal{T}_{\text{ref}}$, whereas the $G$ tensor depends on the mesh geometry and physical coefficients. Such a decomposition provides an advantage over the standard quadrature since $K$ can be precomputed and reused by all of a GPU's SMs. The arity $g$ of $G$ depends on the transformation needed to map the form back onto the reference element, as well as any coefficients in the form.

In order to illustrate this decomposition, we will give a small example, found in Section 2 of [7]. The negative Laplacian with homogeneous Dirichlet boundary conditions can be expressed in Galerkin weak form as

$$\begin{aligned}
\langle v_i, -\Delta u \rangle &= \langle \nabla v_i, \nabla u \rangle && (2)\\
&= \sum_e \int_{\mathcal{T}_e} \nabla v_i(\mathbf{x}) \cdot \nabla u(\mathbf{x}) d\mathbf{x} && (3)\\
&= \sum_e \sum_{j,\alpha} u_j \int_{\mathcal{T}_e} \frac{\partial v_i}{\partial x_\alpha} \frac{\partial v_j}{\partial x_\alpha} d\mathbf{x} && (4)\\
&= \sum_e \sum_{j,\alpha,\mu,\nu} u_j \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\mu}{\partial x_\alpha} \frac{\partial v_i}{\partial \xi_\mu} \frac{\partial \xi_\nu}{\partial x_\alpha} \frac{\partial v_j}{\partial \xi_\nu} |J| d\xi. && (5)
\end{aligned}$$

where $v_i$ is any test function. Thus, the element matrix is given by

$$E_{ij} = \sum_{\mu,\nu} G^{\mu\nu} K^{ij}_{\mu\nu}, \tag{6}$$

where the analytic tensor is

$$K^{ij}_{\mu\nu} = \int_{\mathcal{T}_{\text{ref}}} \frac{\partial v_i}{\partial \xi_\mu} \frac{\partial v_j}{\partial \xi_\nu} d\xi, \tag{7}$$

and the geometric tensor is

$$G^{\mu\nu} = \sum_\alpha \frac{\partial \xi_\mu}{\partial x_\alpha} \frac{\partial \xi_\nu}{\partial x_\alpha} |J| = \sum_\alpha J^{-1}_{\mu\alpha} J^{-1}_{\nu\alpha} |J|. \tag{8}$$

We have used Roman indices to indicate summation over basis functions, and Greek indices for summation over spatial dimensions.

As a second example, we express the linear elasticity operator in the same

form

$$\frac{1}{4}\left\langle \nabla \mathbf{v}_i + \nabla^T \mathbf{v}_i, \nabla \mathbf{u} + \nabla^T \mathbf{u} \right\rangle \tag{9}$$

$$= \sum_e \int_{\mathcal{T}_e} \frac{1}{4}\left(\nabla \mathbf{v}_i + \nabla^T \mathbf{v}_i\right) : \left(\nabla \mathbf{u} + \nabla^T \mathbf{u}\right) d\mathbf{x} \tag{10}$$

$$= \sum_e \sum_{j,\alpha,\beta} \frac{u_j}{4} \int_{\mathcal{T}_e} \left(\frac{\partial v_{i,\beta}}{\partial x_\alpha} + \frac{\partial v_{i,\alpha}}{\partial x_\beta}\right)\left(\frac{\partial v_{j,\beta}}{\partial x_\alpha} + \frac{\partial v_{j,\alpha}}{\partial x_\beta}\right) d\mathbf{x} \tag{11}$$

$$= \sum_{e,j,\alpha,\beta,\mu,\nu} \frac{u_j}{4} \int_{\mathcal{T}_{\text{ref}}} \left(\frac{\partial \xi_\mu}{\partial x_\alpha}\frac{\partial v_{i,\beta}}{\partial \xi_\mu} + \frac{\partial \xi_\mu}{\partial x_\beta}\frac{\partial v_{i,\alpha}}{\partial \xi_\mu}\right)\left(\frac{\partial \xi_\nu}{\partial x_\alpha}\frac{\partial v_{j,\beta}}{\partial \xi_\nu} + \frac{\partial \xi_\nu}{\partial x_\beta}\frac{\partial v_{j,\alpha}}{\partial \xi_\nu}\right)|J| d\xi \tag{12}$$

$$\tag{13}$$

Using symmetries of this form, the FEniCS Form Compiler, discussed below, is able to decompose this into an analytic tensor $K$

$$K^{ij}_{\mu\nu} = \sum_\alpha \frac{1}{4} \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \mathbf{v}_i[\alpha]}{\partial \xi_\mu} \frac{\partial \mathbf{v}_j[\alpha]}{\partial \xi_\nu} d\xi \tag{14}$$

where $i$ and $j$ are multiindices, running over a vector valued element, and $\alpha$ is a component of this vector. The geometric tensor is identical to that for the Laplacian,

$$G^{\mu\nu} = \sum_\alpha J^{-1}_{\mu\alpha} J^{-1}_{\nu\alpha} |J|. \tag{15}$$

### 3.1 More general forms

Our formalism can accomodate any multilinear operator. As a further illustration, we present the Laplace equation incorporating an inhomogeneous coefficient $w$,

$$\int_{\mathcal{T}_e} \nabla \phi_i(\mathbf{x}) \cdot w(\mathbf{x}) \nabla u(\mathbf{x}) d\mathbf{x} \tag{16}$$

$$= \sum_{j,k,\alpha} u_j w_k \int_{\mathcal{T}_e} \frac{\partial \phi_i}{\partial x_\alpha} \phi_k \frac{\partial \phi_j}{\partial x_\alpha} d\mathbf{x} \tag{17}$$

$$= \sum_{j,k,\alpha,\mu,\nu} u_j w_k \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\mu}{\partial x_\alpha}\frac{\partial \phi_i}{\partial \xi_\mu}\frac{\partial \xi_\nu}{\partial x_\alpha}\frac{\partial \phi_j}{\partial \xi_\nu} |J| d\xi \tag{18}$$

$$= \sum_{j,k,\mu,\nu} u_j w_k G^{\mu\nu} K^{ijk}_{\mu\nu}. \tag{19}$$

The full algebra for weak forms is detailed in [8].

Notice that the analytic $K$ tensor is an integral over products of basis functions and basis function derivatives (any member of the jet space). This means that $K$ may be calculated *a priori*, independent of the mesh or form coefficients. We will use this property to design an efficient integration method on massively parallel hardware.

### 3.2 Form Languages

Using the Unified Form Language (UFL) [1] from the FEniCS project, our system accommodates generic weak forms. We use the FEniCS Form Compiler

(FFC) [8], which is implemented in Python, to process input forms and extract parts of the intermediate representation (IR) for use in GPU kernels. We illustrate this process below using linear elasticity as an example. We begin with a standard, primitive variable formulation,

$$\int_\Omega d\mathbf{x} \left( \frac{1}{4} \left( \nabla \mathbf{v} + \nabla^T \mathbf{v} \right) \cdot \left( \nabla \mathbf{u} + \nabla^T \mathbf{u} \right) - \mathbf{v} \cdot \mathbf{f} \right) = 0 \tag{20}$$

where $\mathbf{v}$ is a test function, $\mathbf{u}$ is the solution displacement, and $\mathbf{f}$ is body force. The mathematics becomes the nearly equivalent Python

```
from ufl import interval, triangle, tetrahedron
from ufl import VectorElement, TestFunction, TrialFunction
from ufl import Coefficient, grad, inner, dx
domains = [None, interval, triangle, tetrahedron]
element = VectorElement('Lagrange', domains[dim], 1)
v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(element)

def epsilon(u):
  Du = grad(u)
  return 0.5*(Du + Du.T)

a = inner(epsilon(v), epsilon(u))*dx
L = inner(v, f)*dx
```

using the FEniCS UFL library. The FFC library can processes this form in order to extract the $G$ and $K$ tensors needed for our integration routines,

```
import ffc, numpy
parameters = ffc.default_parameters()
parameters['representation'] = 'tensor'
analysis   = ffc.analysis.analyze_forms([a, L], {}, parameters)
ir         = ffc.compiler.compute_ir(analysis, parameters)

K = ir[2][0]['AK'][0][0].A0.astype(numpy.float32)
G = ir[2][0]['AK'][0][1]
```

where the $K$ tensor is just a numeric array, whereas the $G$ object contains instructions for constructing the geometry tensor given the element Jacobian.

# 4 GPU Implementation

Given an array of geometry tensors for each element, the computation will produce an array of element matrices. Each integration kernel invocation will operate on a set of elements, which we term a *batch*, and thus the set of elements will be divided into batches, of size elementBatchSize, for processing.

Each element integration is accomplished by contracting the geometry tensor $G$ with each block of the analytic tensor $K$, one for each element $E_{ij}$ of the element matrix. We will assign one contraction to each thread in a thread block. In order to increase concurrency, we will allow a thread block to work on multiple elements simultaneously, with the size being numConcurrentElements. Thus, for a vector element with dimension numComponents and a basis of size numBasisFuncs, the thread block will have (numBasisFuncs·numComponents)$^2$· numConcurrentElements threads.

The interleaving of computation with reads and writes to global memory is a strategy for hiding the latency of memory access. When a thread block attempts to write the result of a tensor contraction to global memory, a second thread block, currently in its compute phase, can be scheduled while it is waiting. In our experiments, shown in Section 5, interleaving resulted in noticeably higher performance, presumably due to the increased flexibility afforded to the thread block scheduler. We also employ a *thread coarsening* [21] strategy to increase performance by increasing the work per thread. This was used to great effect by Volkov and collaborators (see [6]) in optimization of finite difference computations. For our problem, this means and increase in the number of geometry tensors handled by a single thread.

We will construct both a CPU and GPU kernel from the same source template, using the Mako [2] templating engine. This will allow us to both check the GPU results, and compare timings easily. Moreover, a single testing setup will verify both generated kernels. A similar capability could be achieved using OpenCL, specifying a different SIMT width for CPU and GPU, and more aggressive loop restructuring. This will be the focus of future work.

## 4.1 Partitioning the Computation

The integration kernel has signature

```
__global__ void integrateJacobian(float *elemMat,
                                  float *geometry,
                                  float *analytic)
```

on the GPU, where `geometry` is an array of the $G$ tensors for elementBatchSize elements, `analytic` is the $K$ tensor, and `elemMat` is an array of the element matrix for each element. On the CPU, we have

```
void integrateJacobian(int numElements,
                       float *elemMat,
                       float *geometry,
                       float *analytic)
```

where the number of elements is passed explicitly to the CPU kernel so that it can execute a loop, whereas the GPU execution grid replaces this loop. In CUDA, we use the block decomposition of kernels to partition the elements into batches,

```
/* Indexes element batch */
const int gridIdx = blockIdx.x + blockIdx.y*gridDim.x;
```

whereas on the CPU we use a loop over batches,

```
/* Loop over element batches */
const int batches = numElements/ELEMENT_BATCH_SIZE;
for(int gridIdx = 0; gridIdx < batches; ++gridIdx) {
```

where we note that in the code itself `ELEMENT_BATCH_SIZE` is replaced by its numeric value.

Once a batch of elements is allocated to a thread block, we assign a thread to each contraction. In CUDA, we use the thread block decomposition to index into $K$ (`KROWS` = numBasisFuncs $\cdot$ numComponents),

```
/* This is (i,j) for test and basis functions */
const int Kidx = threadIdx.x + threadIdx.y*KROWS;
/* Unique thread ID (K block is for a single element) */
const int idx  = Kidx;
```

and on the CPU we have

```
/* Loop over test functions */
for(int i = 0; i < KROWS; ++i) {
  /* Loop over basis functions */
  for(int j = 0; j < KROWS; ++j) {
    /* This is (i,j) for test and basis functions */
    const int Kidx = i + j*KROWS;
    /* Unique thread ID (K block is for a single element) */
    const int idx  = Kidx;
```

This scheme must be modified slightly when we concurrently evaluate several elements in a single thread block. In CUDA, we use the third thread block dimension to index the simultaneous evaluations,

```
/* This is (i,j) for test and basis functions */
const int Kidx = threadIdx.x + threadIdx.y*KROWS;
/* Unique thread ID
   (Same K block is used by all concurrent elements) */
const int idx  = Kidx + threadIdx.z*KROWS*KROWS;
```

and on the CPU we introduce another loop

```
/* Loop over test functions */
for(int i = 0; i < KROWS; ++i) {
  /* Loop over basis functions */
  for(int j = 0; j < KROWS; ++j) {
    /* Loop over simultaneous evaluations */
    for(int k = 0; k < NUM_CONCURRENT_ELEMENTS; ++k) {
```

```
/* This is (i,j) for test and basis functions */
const int Kidx = i + j*KROWS;
/* Unique thread ID
   (Same K block is used by all concurrent elements) */
const int idx  = Kidx + k*KROWS*KROWS;
```

Hereafter we will assume that we have simultaneous evaluations, since the reduction to the single evaluation case is straightforward. We will refer to the set of contractions performed by a given thread as the *sequential* contractions, and contractions that happen simultaneously using different sets of threads in a thread block as *concurrent* contractions. The set of threads in a thread block which all perform contractions for the same element set will be termed a *contraction set*.

In order to clarify the data layout on the GPU, Fig. 4.1 shows how memory and computation is laid out across the GPU. It represents the calculation by our kernel of the FEM element matrices of the 3D $P_1$ Laplacian for eight cells, evaluated concurrently in groups of two. Looking at subfigure (a), the boxes along the sides are geometric cell data loaded when the kernel starts up, and labeled by $G_k^c$ where $k$ counts the groups of concurrent evaluations and $c \in [0, \text{numConcurrentElements})$ is the index into each group. Each block in the center matrix represents the $Kij$ blocks to be contracted with each $G$ to produce the element matrix entry $E^{ij}$. The arrows represent the contraction computation, linking the data involved, and are labeled by the thread which executes the computation. Notice we have 32 threads in this example, which is $\text{numBasisFuncs}^2 \cdot \text{numConcurrentElements}$. Each thread marches through a set of cells, generating a single element matrix entry each time.

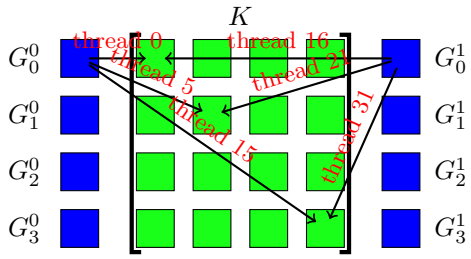
## 4.2 Marshaling Data

For each sequential contraction, all threads in the contraction set must access the set of $G$ tensors for the elements in question. Therefore, these are loaded into shared memory from the `geometry` input array using a sequence of coalesced loads followed by a remainder if necessary. We illustrate this below for the case where $G$ is $3 \times 3$, elementBatchSize is 5, and there are 16 threads.

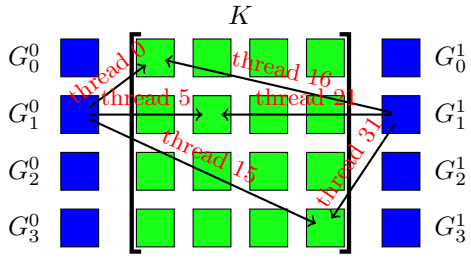
```
const int        Goffset = gridIdx*DIM*DIM*ELEMENT_BATCH_SIZE;
__shared__ float G[DIM*DIM*ELEMENT_BATCH_SIZE];

G[idx+0]  = geometry[Goffset+idx+0];
G[idx+16] = geometry[Goffset+idx+16];
if (idx < 13) G[idx+32] = geometry[Goffset+idx+32];
```

In the CPU version, we merely load G from memory on the first iteration. Each thread uses a single block of $K$ for every contraction it performs. In 2D, we have, after unrolling the loop,

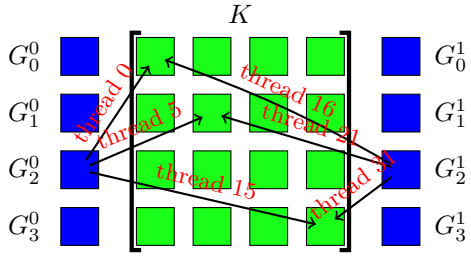
```
const int Koffset = Kidx*DIM*DIM;
float     K[DIM*DIM];
```



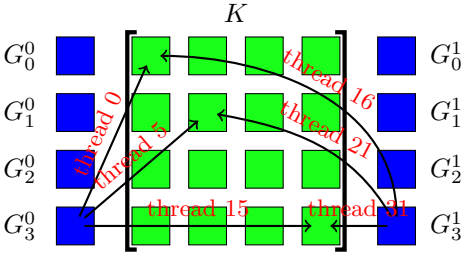

(a) Thread assignments for the calculation of two concurrent contractions for the first set of cells.




(b) Thread assignments for the calculation of two concurrent contractions for the second set of cells.




(c) Thread assignments for the calculation of two concurrent contractions for the third set of cells.




(d) Thread assignments for the calculation of two concurrent contractions for the fourth set of cells.

Figure 1: Tensor Contraction kernel $G^{\beta\gamma}(\mathcal{T})K^{ij}_{\beta\gamma}$ for 3D $P_1$ Laplacian assembly calculated for a batch of eight cells, with two cells evaluated concurrently. Subfigure (a) shows the contraction at the first step, where geometric data for the first two cells ($G_0^0$ and $G_0^1$) is contracted with the subblocks of the $K$ tensor. Each thread performs a single contraction, resulting in a single element of the cell FEM element matrix. Subfigures (b), (c) and (d) show subsequent contractions for the othe pairs of cell input.

```
K[0] = analytic[Koffset+0];
K[1] = analytic[Koffset+1];
K[2] = analytic[Koffset+2];
K[3] = analytic[Koffset+3];
```

This load is performed after the $G$ load, but before the call to `__syncthreads()` needed to make the $G$ data available, in order to try and cover the latency of this uncoalesced read. Finally, we allocate space to hold the element matrix entry produced by each thread,

```
const int Eoffset = gridIdx*KROWS*KROWS*ELEMENT_BATCH_SIZE;
float     E[ELEMENT_BATCH_SIZE/NUM_CONCURRENT_ELEMENTS];
```

however we can replace `E[]` with a single scalar if we interleave calculation with writes to global storage, as shown below.

For each thread block, we load elementBatchSize $G$ tensors, one $K$ tensor, and write elementBatchSize $E$ matrices. We can parametrize the size of these tensors using two variables, the total number of entries in $G$, sizeG, and the number of basis functions per element, numBasisFuncs. Thus, the memory traffic per thread thread block is given by

$$MT_{TB} = \text{elementBatchSize}\left(\text{sizeG} + \text{numBasisFuncs}^2\right) + \text{numBasisFuncs}^2\text{sizeG} \tag{21}$$

floating point numbers, and the shared memory required is

$$MS_{TB} = \text{elementBatchSize}\left(\text{sizeG} + \frac{\text{numBasisFuncs}^2}{\text{numConcurrentElements}}\right), \tag{22}$$

with `sizeG` in thread local memory for each block of $K$. For example, in our 3D $P_1$ Laplacian evaluation, we have sizeG $= 9$ and numBasisFuncs $= 4$, so that

$$MS_{TB} = \text{elementBatchSize}\left(9 + \frac{16}{\text{numConcurrentElements}}\right). \tag{23}$$

The best performance was realized using 128 elements per batch and 2 concurrent evaluations, so that

$$MS_{TB} = 128\left(9 + \frac{16}{2}\right)4\frac{\text{bytes}}{\text{scalar}} = 8.5KB. \tag{24}$$

Since the GTX 285 has 1GB of main memory, it could contain all the data for almost 16M elements.

## 4.3 Computation

When computing the contraction of a set of $G$ tensors with a block of $K$, we can choose to update global memory after the entire set of contractions has been processed, or after each contraction in turn. The `interleaveStores` flag

determines which strategy we pursue in the generated code. Interleaving computation with writes to global memory may allow the latency of a write to be covered by computation from another warp in the thread block, or another thread block scheduled on the SM.

Our generation engine allows each loop to be either generated, or unrolled to produce straight-line code [16]. In our examples, we will only display the loop code due to its brevity, but unrolled versions are presented in the results (see Section 5).

```
const int serialBatchSize =
  ELEMENT_BATCH_SIZE/NUM_CONCURRENT_ELEMENTS;
for(int b = 0; b < serialBatchSize; ++b) {
  const int n = b*numConcurrentElements;
  contractBlock('n', dim, 'E', 'G', "Goffloc", 'K', loopUnroll)
}
```

Here `contractBlock()` generates the proper contraction code using the names provided for input and output arrays and offsets, the sizes, and the flag for loop unrolling.

We then write each element matrix into memory contiguously with a fully coalesced write,

```
/* Store contraction results */
const int outputSize = NUM_CONCURRENT_ELEMENTS*KROWS*KROWS;
for(int n = 0; n < serialBatchSize; ++n) {
  elemMat[Eoffset+idx+n*outputSize] = E[n];
}
```

where we note that this loop is fully unrolled in the generated code.

When interleaving stores, we do a single contraction and then immediately write the result to global memory. The latency for this write can be covered by scheduling contractions in other warps on this SM. This strategy has produced consistently better results than fully calculating the contractions before writing the resulting element matrices to global memory. We show the code below, where as before the contraction is fully inlined in the generated code.

```
for(int b = 0; b < serialBatchSize; ++b) {
  const int n = b*numConcurrentElements;
  E = 0.0;
  contractBlock('n', dim, 'E', 'G', "Goffloc", 'K', loopUnroll)
  /* Store contraction result */
  elemMat[Eoffset+idx+b*outputSize] = E;
}
```

Each contraction consumes $2 \cdot \text{sizeG}$ flops, and there are $\text{numBasisFunc}^2$ contractions per element matrix. Thus, the total flops executed per thread block is given by

$$W_{TB} = 2 \cdot \text{sizeG} \cdot \text{numBasisFunc}^2 \cdot \text{elementBatchSize}. \tag{25}$$

For our 3D $P_1$ Laplacian calculation, we have 288 flops per cell. Using (21) and (25), we can classify this algorithm according to its resource requirements. We can examine the ratio of flops to bytes required, which we will call $\beta$,

$$\beta = \frac{W_{TB}}{4MT_{TB}} = \frac{1}{2}\frac{1}{\frac{1}{\text{numBasisFunc}^2} + \frac{1}{\text{sizeG}} + \frac{1}{\text{elementBatchSize}}}. \tag{26}$$

For our 3D $P_1$ Laplacian calculation with batches of 128 cells, we have $36,864$ flops and $13,376$ bytes transferred, giving

$$\beta = 2.75\frac{flop}{byte}. \tag{27}$$

For a very large batch size, we could approach $\beta = 2.88$, whereas for a single cell $\beta = 0.43$. We can make use of this ratio by calculating the bandwidth necessary to run at the peak flop rate for the GTX 285,

$$B_{req} = \frac{F_{peak}}{\beta} = \frac{692}{2.75}GB/s = 252GB/s, \tag{28}$$

which is beyond the device capability. At the achievable bandwidth, we could obtain

$$F_{max} = \beta B_{peak} = (2.75)(130)GF/s = 358GF/s. \tag{29}$$

We do not achieve this peak, but something above the single cell threshold of $56GF/s$, which suggests that we have been unable to completely cover the non-coalesced load of $K$ for each thread block. A future enhancement might put $K$ into constant memory to better utilize bandwidth.

# 5 Results

We demonstrate the performance of our integration method using the common Laplacian and linear elasticity operators, as shown in Fig. 2, with all computations being done in single precision. We achieve nearly 100GF for the Laplacian, and even a little more for the elasticity operator. Note that we achieved the highest performance using interleaved stores and having each thread block operate on two elements simultaneously. The batch sizes are somewhat different, but performance is not very sensitive to this variable, as shown in Fig. 3.

To demonstrate the benefit of interleaving stores, we examine integration of the 3D $P_1$ Laplacian. The best performance was realized for an element batch size of 128 using 2 concurrent element evaluations. In Fig. 4 we show the results for these choices for both fully unrolled loops and the no unrolling case. Clearly, interleaving produces better performance, even in the fully unrolled case where discrepancies appear only for large runs. The disparity between the loop unrolling cases indicates that the compiler may not be applying this transformation optimally. We have performed over 3000 test runs with various combinations of the input parameters, which are archived along with the source code, so that they may be mined in a similar fashion by other researchers.

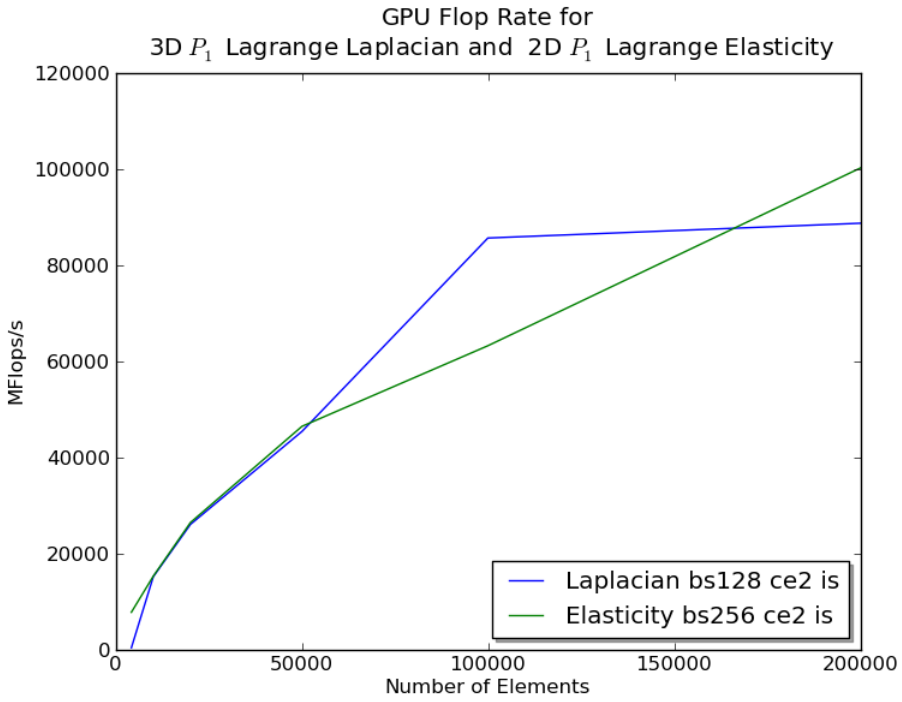



Figure 2: This graph shows the peak performance achieved for element integration of the 3D $P_1$ Laplacian and 2D $P_1$ Elasticity operators. We use *bs* to denote the element batch size, *ce* the number of concurrent element evaluations, *is* interleaved stores, and *unroll* for fully unrolled contraction loops.

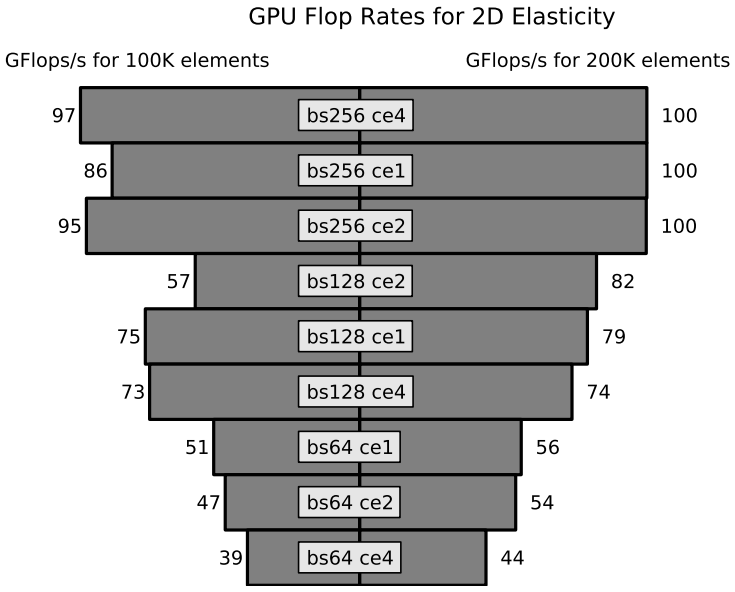



Figure 3: This graph shows the dependence of flop rate on the element batch size for the 2D $P_1$ Elasticity operator. We use *bs* to denote the element batch size and *ce* the number of concurrent element evaluations; each run used interleaved stores and fully unrolled contraction loops.

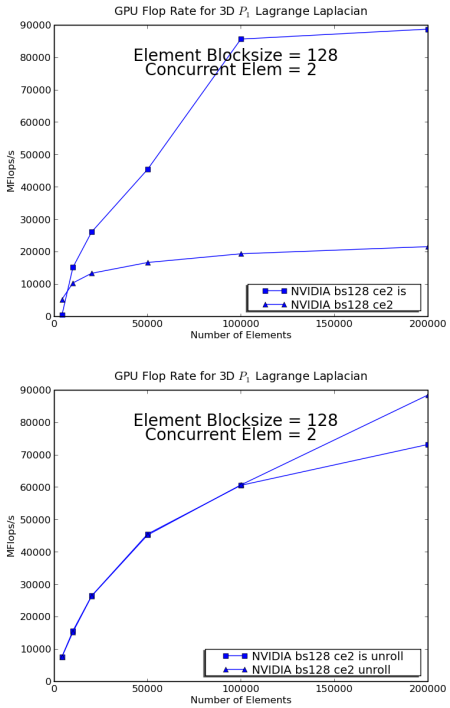



Figure 4: This graph shows the dependence of flop rate on the the interleaving of global stores for the 3D $P_1$ Laplacian operator. We use *bs* to denote the element batch size, *ce* the number of concurrent element evaluations, *is* interleaved stores, and *unroll* for fully unrolled contraction loops. The left graph shows performance with no loop unrolling, and the right for fully unrolled loops.

# 6 Discussion

We have shown that by generating vectorized code, which is also able to overlap computation and memory access, we can take advantage of the large memory bandwidth and many vector units of a GPU for FEM integration, even on very low order elements. We expect this strategy to be effective for the forseeable future on manycore machines since we can easily accomodate increased vector lengths by using more concurrent element evaluations. We note that a version of the Laplace kernel was tested in which $K$ is loaded into shared memory and all threads perform the complete contraction with a given $G$ simultaneously. However, this strategy was abandoned due to lack of efficiency, mainly arising from the lower level of concurrency available.

We will extend these initial results to more complex operators with variable coefficients, as well as systems of equations which arise in multiphysics problems. This will necessitate a more systematic approach to optimization over the algorithmic variants. We plan to use the loop slicing tool Loo.py [9] and generated, optimized quadrature rules from FFC [20] in addition to exhaustive strategies. We have an integrated build and test framework, which allows us to run all variants in a single execution and record the results in HDF5 for later processing and analysis. Moreover, when processing coupled systems, we will be able to break the weak form into blocks commensurate with different preconditioning strategies and evaluate the performance. This entire package will be integrated into both PETSc and FEniCS for easy use by projects already using these frameworks.

# References


[1] M. S. Alnæs and A. Logg. *UFL Specification and User Manual*. Simula Research, 2009. `https://launchpad.net/ufl`.

[2] Mike Bayer. The Mako templating system. `http://www.makotemplates.org/`, 2010.

[3] N. Bell and M. Garland. The Cusp library. `http://code.google.com/p/cusp-library/`, 2010.

[4] N. Bell and J. Hoberock. The Thrust library. `http://code.google.com/p/thrust/`, 2010.

[5] Jed Brown, 2011. Private communication with code sample.

[6] Cris Cecka, AJ Lew, and Eric Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85:640–669, 2011.

[7] Jonathan Cohen and Michael Garland. Solving computational problems with GPU computing. *Computing in Science and Engineering*, 11(5):58–63, 2009.

[8] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[9] R. C. Kirby, M. G. Knepley, A. Logg, and L. R. Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(6):741–758, 2005.

[10] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, 2006.

[11] Andreas Klöckner. Loo.py, 2011. unpublished loop slicing tool.

[12] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA: GPU run-time code generation for high-performance computing. `http://arxiv.org/abs/0911.3456v1`, 2009.

[13] Andreas Klöckner, Tim Warburton, Jeff Bridge, and Jan S Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 228:7863–7882, 2009.

[14] Dimitri Komatitsch, David Michéa, and Gordon Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing*, 69(5):451 – 460, 2009.

[15] Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Automated scientific computing. `http://launchpad.net/fenics-book`, To appear 2011.

[16] G.R. Markall, D.A. Ham, and P.H.J. Kelly. Generating Optimised Finite Element Solvers for GPU Architectures. In *American Institute of Physics Conference Series*, volume 1281, pages 787–790, College Park, MD, 2010. AIP.

[17] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity GPUs. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, Piscataway, NJ, 2010. IEEE.

[18] G.S. Murthy, M. Ravishankar, M.M. Baskaran, and P. Sadayappan. Optimal loop unrolling for GPGPU programs. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11, Piscataway, NJ, April 2010. IEEE.

[19] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, Santa Clara, CA, 2007.

[20] NVIDIA Corporation. *NVIDIA CUBLAS User Guide*. NVIDIA Corporation, Santa Clara, CA, 2010.

[21] NVIDIA Corporation. *NVIDIA CUSPARSE User Guide*. NVIDIA Corporation, Santa Clara, CA, 2010.

[22] K. Oelgaard, A. Logg, and G. N. Wells. Automated code generation for discontinuous galerkin methods. *SIAM J. Sci. Comput.*, 31(2):849–864, 2008.

[23] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[24] Z.A. Taylor, M. Cheng, and S. Ourselin. High-speed nonlinear finite element analysis for surgical simulation using graphics processing units. *Medical Imaging, IEEE Transactions on*, 27(5):650–663, may 2008.

[25] Stefan Turek, Dominik Göddeke, Christian Becker, Sven H.M. Buijssen, and Hilmar Wobker. FEAST - realisation of hardware-oriented numerics for HPC simulations with finite elements. *Concurrency and Computation: Practice and Experience*, 22(6):2247–2265, May 2010.