Forward Heat Equation Model with Heterogeneous Computing

October 26, 2016

1 Preliminaries

Some background for the context of the problem and the math that goes into it.

1.1 FE Formulation I (PDE)

We wish to solve the heat equation in 3 dimensions with spatially depenedent material coefficients. In strong form,

$$\begin{cases} \rho C \frac{\partial T}{\partial t} = \nabla \cdot (k \nabla T(\vec{x}, t)) & \text{in domain } \Omega \times (0, t_f), \\ k \frac{\partial T(\vec{x}, t)}{\partial \vec{n}} = f(\vec{x}) & \text{on sides}, \\ T(\vec{x}, 0) = T_{ambient}, \end{cases}$$

where the relevant thermal properties are the material density, ρ, specific heat, C, and thermal conductivity, k, all of which are assumed to be constant with temperature. Discretizing in time with a θ-scheme[1], the spatio-temporal temperature profile $T(\vec{x}, t)$ is reduced to a finite set of temperatures at regularly spaced time increments, $\{T^i(\vec{x})\}_{i \in \mathcal{I}}$, $\mathcal{I} = \{0, 1, \ldots, t_f/\Delta t\}$.

$$\left(\text{in general, } \frac{dT}{dt} = Q \text{ discretizes to } \frac{T^i - T^{i-1}}{\Delta t} = \theta Q^i + (1-\theta) Q^{i-1}, \text{ where superscript denotes each time step} \right)$$

gives

$$\begin{aligned} \rho C \frac{T^i - T^{i-1}}{\Delta t} &= \theta \nabla \cdot \left(k \nabla T^i \right) + (1-\theta) \nabla \cdot \left(k \nabla T^{i-1} \right) \\ \rho C T^i - \theta \Delta t \nabla \cdot \left(k \nabla T^i \right) &= \rho C T^{i-1} + (1-\theta) \Delta t \nabla \cdot \left(k \nabla T^{i-1} \right). \end{aligned}$$

Then converting to weak form and integrating by parts

$$\left(\int_\Omega \left(\nabla^2 u \right) v dx = \int_{\partial \Omega} \frac{\partial u}{\partial \vec{n}} v ds - \int_\Omega \nabla u \cdot \nabla v dx \right)$$

with $\nabla u = k \nabla T^i$, gives the operators

$$\begin{aligned} a(T^i(\vec{x}), \phi(\vec{x})) &= \int_\Omega \left(\rho C T^i \phi + \theta \Delta t k \nabla T^i \cdot \nabla \phi \right) d\vec{x} \\ L(\phi(\vec{x})) &= \int_\Omega \left(\rho C T^{i-1} \phi - (1-\theta) \Delta t k \nabla T^{i-1} \cdot \nabla \phi \right) d\vec{x} + \int_{\partial \Omega} \Delta t f \phi ds. \end{aligned}$$

We presume T^{i-1} to be known, and find T^i so that $a(T^i, \phi) = L(\phi)$ for all ϕ in some family of functions. If this family is composed of a finite set of basis functions $\{\phi_m\}_{m \in \{1, \ldots, N\}}$, the finite element method can be used to solve for T^i as a linear combination of them $T^i = \sum_m U^i_m \phi_m$, where $\vec{U}$ is a vector of

[1] see, e.g., https://fenicsproject.org/book

coefficients. The function f is similarly represented by the coefficient vector $\vec{F}$. To improve runtime, we adjust this form further so that assembly, the computation of pariwise inner products of all basis functions into a matrix, is not repeated at every time step.

$$\begin{aligned}
a &= \int_\Omega \left(\rho C T^i v + \theta \Delta t k \nabla T^i \cdot \nabla v\right) dx \\
&= \int_\Omega \left(\left(\sum_m \rho C U_m^i \phi_m \right) \hat{\phi}_n + \theta \Delta t \nabla \left(\sum_m k U_m^i \phi_m \right) \cdot \nabla \hat{\phi}_n \right) dx \\
&= \sum_m \left(\int_\Omega \rho C \phi_m \hat{\phi}_n dx \right) U_m^i + \theta \Delta t \sum_m \left(\int_\Omega k \nabla \phi_m \cdot \nabla \hat{\phi}_n dx \right) U_m^i, \\
L &= \int_\Omega \left(\rho C T^{i-1} v - (1-\theta)\Delta t k \nabla T^{i-1} \cdot \nabla v + \Delta t f v\right) dx \\
&= \sum_m \left(\int_\Omega \rho C \phi_m \hat{\phi}_n dx \right) U_m^{i-1} - (1-\theta)\Delta t \sum_m \left(\int_\Omega k \nabla \phi_m \cdot \nabla \hat{\phi}_n dx \right) U_m^{i-1} + \Delta t \sum_m \left(\int_\Omega \phi_m \hat{\phi}_n dx \right) F_m.
\end{aligned}$$

The finite element method involves the matrices of pairwise integrals of basis functions

$$\mathbf{M} = \left[\int_\Omega \rho C \phi_m \hat{\phi}_n dx \right]_{m,n \in \{1,\ldots,N\}} \text{ and } \mathbf{K} = \left[\int_\Omega k \nabla \phi_m \cdot \nabla \hat{\phi}_n dx \right]_{m,n \in \{1,\ldots,N\}}.$$

Computation of these matrices is the process of *assembly*. With $\mathbf{M}$ and $\mathbf{K}$ available, solving the weak form of the heat equation for all $\phi \in \{\phi_m\}_{m=1,\ldots,N}$ is equivalent to solving the matrix equation at each time step

$$[\mathbf{M} + \theta \Delta t \mathbf{K}] \vec{U}^i = [\mathbf{M} - (1-\theta)\Delta t \mathbf{K}] \vec{U}^{i-1} + \Delta t \mathbf{M} \vec{F}$$

for $\vec{U}^i$.

1.2 FE Formulation II (assembly-free methods)

The matrices $\mathbf{M}$ and $\mathbf{K}$ are typically contructed by summing local contributions from each element in the assembly process. A local assembly matrix for element e with D degrees-of-freedom contains the pairwise inner products of all basis functions with support in element e,

$$\mathbf{M}_e = \left[\int_{\Omega_e} \phi_m \hat{\phi}_n dx \right]_{m,n \in \{1,\ldots,D\}}.$$

Material property coefficients are taken to be constant over each element, so that the elemental assembly matrices depend only on the geometry of the domain. If all of the finite elements are the same size and shape, a single elemental assembly matrix can be reused and the mesh is said to have a *fixed grid* (FG). The *assembly operator* A over the index set of elements $\mathcal{E}$ denotes the process of constructing a full system matrix from its local contributions. For example,

$$\mathop{\mathsf{A}}_{e \in \mathcal{E}} \rho C \mathbf{M}_e = \mathbf{M}.$$

The assembly operator can also be applied to contributions to a single vector over each element to give the full vector. It is only a notational convenience to describe the mapping from local degrees of freedom to sums over global degrees of freedom. As such, the following are valid notation for $\mathbf{M}\vec{x} = \vec{y}$

$$\mathop{\mathsf{A}}_{e \in \mathcal{E}} \rho C \mathbf{M}_e \vec{x}_e = \mathop{\mathsf{A}}_{n \in \mathcal{N}} \left(\sum_{e \in \mathcal{E}^{(n)}} \rho C \mathbf{M}_e^n \vec{x}_e \right) = \vec{y}, \tag{1}$$

meaning that assembly is computed in terms of the degrees-of-freedom in the "outer loop", with each of their elemental contributions computed separately. The first method is an EbE approach, similar to the standard method of assembling $\mathbf{M}$. The second is a DbD approach, in which the necessary vector-vector products are viewed with finer granularity, introduced by Martínez-Frutos et al. The freedom of interpretation of the assembly operator gives rise to the different strategies for parallel matrix-vector multiplication.

To simplify notation, let $\mathbf{A} = [\mathbf{M} + \theta\Delta t\mathbf{K}]$, $\mathbf{L} = [\mathbf{M} - (1-\theta)\Delta t\mathbf{K}]$, and $\vec{N} = \Delta t\mathbf{M}\vec{F}$. Set $\vec{b} = \mathbf{L}\vec{U}^{i-1} + \vec{N}$ and $\mathbf{A}_e = \rho C\mathbf{M}_e + \theta\Delta tk\mathbf{K}_e$ for $e \in \mathcal{E}$. Then the problem of finding $\vec{U}^i$ at each time step is reduced to solving

$$\mathbf{A}\vec{U}^i = \mathop{\mathsf{A}}_{e\in\mathcal{E}} \mathbf{A}_e\vec{U}_e^i = \mathop{\mathsf{A}}_{n\in\mathcal{N}} \left(\sum_{e\in\mathcal{E}^{(n)}} \mathbf{A}_e^n\vec{U}_e^i \right) = b.$$

We note once again that the explicit computation and storage of $\mathbf{A}$ and $\mathbf{L}$ is not necessary if $\{\mathbf{M}_e\}_{e\in\mathcal{E}}$ and $\{\mathbf{K}_e\}_{e\in\mathcal{E}}$ are available.

Assembly-free methods are especially useful if the spatially dependend material properties are not known in advance, or if many simulations are to be done over the same domain with varying coefficients. The generation of mesh geometry and computation of elemental assembly matrices can be done in advance and reused so that all remaining computations required for a matrix-vector product are parallelizable. Figure 1 illustrates a 3D FG mesh with tetrahedral elements and three sets of material properties parameterized by shading. Each of these spatially varying functions for the material properties have a simple functional form, and can be quickly composed with elementary assembly matrices that are computed and stored beforehand.

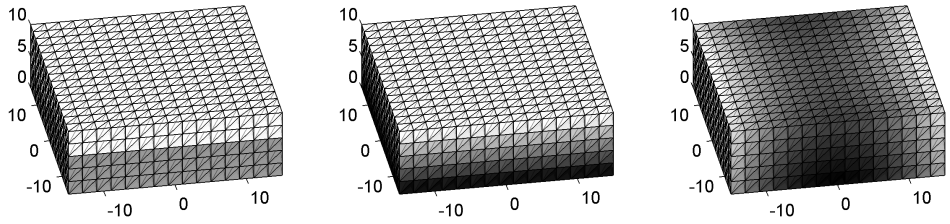

Figure 1: 30×30×10 mm domain with tetrahedral meshing and three functions describing changes in material properties. A boundary between two materials can be abrupt (left) or smoothed over many elements (middle). Heat conduction can also be simulated on a domain with more complex dependence between material properties and spatial location (right) with negligible computational burden, provided that the properties have a closed functional form (here, the shading is computed as $x^2 - 0.2y^2 + 10z$).

1.3 Preconditioned Conjugate Gradient

The system matrix $\mathbf{A} = [\mathbf{M} + \theta\Delta t\mathbf{K}]$ is large, sparse, symmetric, and positive definite. The system above is thus solvable with the PCG algorithm. At each time step, let $x = \vec{U}^i$ (to free superscripts and the index i), initialized at a starting guess. We also need $\mathbf{A}, b$, an error tolerance, and a preconditioner matrix $\mathbf{P}$ for which $P^{-1}\vec{x}$ is easily computable and $\mathbf{P}^{-1}\mathbf{A}$ is realatively well conditioned, such as the Jacobi preconditioner.
The algorithm is:[2]

```
i = 0
1) r = b - A*x                                  (MVP)
2) d = Pinv*r                                   (DIMVP)
3) delta_new = r'*d                             (VVP)
While (i < i_max) and (delta_new > tol):
   4) q = A*d                                   (MVP)
   5) alpha = delta_new/(d'*q)                  (VVP)
   6) x = x + alpha*d                           (VAVSP)
   If i is divisible by 50
      7) r = b - A*x                            (MVP)
   else
      7) r = r - alpha*q                        (VAVSP)
   8) s = Pinv*r                                (DIMVP)
   9) delta_new = r'*s                          (VVP)
   10) beta = delta_new/delta_old
   11) d = s + beta*d                           (VAVSP)
   i = i + 1
```

The annotations correspond to the computations matrix-mector product (MVP), diagonal inverse matrix-vector product (DIMVP), vector-vector product (VVP), and vector add to vector-scalar product (VAVSP). The dominating computation is the MVP in step 4 each iteration. All of the other computations are easily parallelizable. The parallization and implementation of the MVP on GPU is the focus of this work.

1.4 OpenCL Heterogeneous Computing Framework

1.4.1 Computation heirarchy

With OpenCL, GPU are programmed with *kernels*, small bits of C code that are sent in parallel to the individual cores. At any given time, each of the hundreds of cores is acting as a *work item*, which is the most granular operating unit in the hierarchy. Work items are structured in *work groups*, with as few as one work item per work group. You are responsible for defining the sizes and dimensions (1, 2, or 3) of the hierarchical structure so that at the time of execution, each work item is provided with unique identifying information and the generic kernel code. The identifying information is:

- *global_id* ranging from 0 to the total number of work items in each dimension,
- *local_id* ranging from 0 to the number of work items in a work group, in each dimension,
- *group_id* ranging from 0 to the total number of work groups in each dimension.

The total number of work items, total number of work groups, and size of each work group is also available. The kernel explicitly tells each work item how to use its ID to contextualize itself within the larger problem. When it all works together, work items gather the data they need, execute in parallel, and put everything back together.

[2] http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf

1.4.2 Memory heirarchy

The use of memory on a GPU dictates programming strategies and the success or failure of an algorithm. At fully efficient throughput, a single core can execute several floating point operations per clock cycle. However, accessing data from the "slow" global memory location can take 400-600 clock cycles. This alone warrants special attention to the OpenCL memory heirarchy[3].
Data that is loaded from the CPU or stored as the output of work items must live in *global* memory, which has GB of space. During execution, all work items have their own small amount of *private* memory, which is on-chip and not visible to any other work item. This is fast to access, and used for variables that can take different values across every work item. In between, there is *local* memory, which is also on-chip, and shared among a single work group. There are usually tens of KB reserved for local memory for each work item. Access to local memory is roughly 100 times faster than accessing global memory, provided that work items within the work group aren't making conflicting calls ("bank conflict"). Local memory is allocated outside of the kernel, and cannot be initialized. Finally, kernels can consider certain data as *constant* memory. This data is physically still in global memory, but a kernel cannot write to it. When a kernel reads data from constant memory, it is cached, so that subsequent reads are fast. Global memory is not cached.
There are two strategies for gracefully managing reads from global memory when it is necessary. First, the latency can effectively be hidden if there is enough independent computation to keep a work item busy between when the data is called and when it is used. This is preferable, though not always possible. Second, a kernel can take advantage of the way that hardware loads data from global memory to private memory. A single transaction with global memory yields 32 words (such as 4 byte floats) of data, whether it is all called for or not. If work items that are indexed sequentially by global_id,request data from global memory that is organized in the same sequential way, the calls are automatically bundled and processed as one transaction. This process is the simplest form of *coalesced* memory access[4].

1.4.3 Programming strategy

The OpenCL programming structure is as follows:

1. Investigate your machine hardware to define a *host* (CPU/hard drive etc., the conventional computing environment), its *devices(s)* (GPU with its onboard memory), and define your *context* the overall comuting playground.

2. Define initial variables on the host

3. Load data onto a device. This includes reserving space for data that a kernel will write later, anything that is meant to last from one kernel to another. The neat thing about defining all of the memory space is that you are allowed to specify whether each buffer is read or write only (or both) for both the host and device, or even if you know that the host will never try to read it. Then the space that is allocated will be optimal for however the data will be treated.

4. *Build* the compute kernels. This includes compiling the C code and specifying pointers to memory buffers where its arguments can be found. A single kernel program can be built multiple times with different arguments, as is the case with the handful of vector inner products in the PCG algorithm. Each of these are treated individually.

5. Define a *queue* in the context. The host can enqueue kernels, memory transfer operations, or wait fences. OpenCL turns these into individual tasks that are performed as cores on the GPU become available. You write your code as if you have infinitely many cores to run in parallel, and

[3]Which has critital differences from the CUDA memory heirarchy, though they share some language

[4]There have been advances in hardware and in OpenCL standards to provide more flexibility, such as allowing permutations of the 32 sequential words to 32 work items that are blocked together but not necessarily in the same order

the queue handles the rest. In some cases, the host can enqueue commands faster than they are flushed out, which isn't a real problem, it just makes profiling code more challenging. In fact, it's best not to wait for a queue to be emptied before adding more commands just to keep things always busy. Flags can be added to ensure that all tasks from one kernel are finished before any tasks from the next kernel start, in case memory is being written and then read in a dependent way.

6. Enqueue commands to copy memory from a device to the host. This can be the final result of computations, or in my case, the PCG residual so that the host can decide if it needs to go through another iteration of enqueueing kernel commands.

I wrote a collection of kernels to cary out fundamental linear algebra tasks necessary in the PCG algorithm: MVP, VAVSP, etc. All data is transferred to the GPU, which handles almost every part of each PCG iteration. For example, in the update stage, a kernel will take pointers to global memory for the old solution vector, the update unit vector direction, and the step size, and then store the updated solution vector back in global memory. The CPU is only responsible for launching these kernels and receiving the calculated residual (scalar) to determine if another iteration is necessary.

2 GPU PCG algorithm

Here I will describe the kernels I wrote, how they use GPU memory, and how they fit into the PCG algorithm.

2.1 Kernels

2.1.1 VVP_A

First stage of a vector-vector product. Takes pairwise scalar products of vector elements and then uses a standard parallel "reduction" algorithm to sum the results in $\log(N)$ complexity. The work group size is maximized for the available hardware, and the partial sum is reduced by a factor of two at each step. The total number of elements in the partial sum can only be reduced by a factor of the maximum work group size with a single kernel call. In case the length of the vector, nVertices, is not a multiple of the maximum work group size, max_wg_size, the number of work groups is rounded up. The size of the result is then

$$\text{r1\_size} = \left\lceil \frac{\text{nVertices}}{\text{max\_workgroup\_size}} \right\rceil .$$

Argument	**In/Out**	**Memory space**	**Data type**	**Size**
*x	input	global	float32	nVertices
*y	input	global	float32	nVertices
N	input		uint32	1
*VVP_loc		local	float32	max_workgroup_size
*r	output	global	float32	r1_size

global size	r1_size × max_wg_size	**local size**	max_wg_size

2.1.2 VVP_reduce

Intermediate and final stages of vector-vector product. Takes an intermediate partial sum from VVP_A or itself and reduces it by a factor of max_workgroup_size. If nVertices $\leq$ max_workgroup_size, the

vector-vector product is completed. Otherwise further calls with this kernel are made. Since this can be iterative, set

$$\text{rk\_size} = \left\lceil \frac{\text{r(k-1)\_size}}{\text{max\_workgroup\_size}} \right\rceil .$$

Argument	**In/Out**	**Memory space**	**Data type**	**Size**
*rPrevious	input	global	float32	r(k-1)_size * max_workgroup_size
N	input		uint32	1
*VVP_loc		local	float32	max_workgroup_size
*r	output	global	float32	rk_size
global size	rk_size × max_wg_size	**local size**	max_wg_size	

2.1.3 VVP_C

An alternative final stage for vector-vector product for step 5 of PCG. Rather than storing the final result of the sum, the scalar delta _new is loaded and alpha = delta_new/(d'*q) is stored, along with negative alpha.

Argument	**In/Out**	**Memory space**	**Data type**	**Size**
*rPrevious	input	global	float32	r(k-1)_size * max_workgroup_size
*delta	input	global	float32	1
N	input		uint32	1
*b		local	float32	max_workgroup_size
*alpha	output	global	float32	1
*neg_alpha	output	global	float32	1
global size	rk_size	**local size**	max_wg_size	

2.1.4 VAVSP

Computes the elementwise sum of a vector and a scalar multiplied by another vector. The scalars are loaded from GPU global memory, so a negative scalar must be used if vector subtraction is desired.

Argument	**In/Out**	**Memory space**	**Data type**	**Size**
*x	input	global	float32	nVertices
*y	input	global	float32	nVertices
*a	input	global	float32	1
*x_plus_ay	output	global	float32	nVertices
global size	nVertices	**local size**	None	

2.1.5 DIMVP

Computes the matrix-vector product where the matrix is the inverse of a diagonal matrix P. Element i of the result is x_i/P_{i,i}.

2.1.6 Beta_update

Computes the coefficient beta and updates delta in storage. Used to avoid data transfer between device and host for the performance of a small calculation.

Argument	In/Out	Memory space	Data type	Size
*P	input	global	float32	nVertices
*x	input	global	float32	nVertices
*Pinvx	output	global	float32	nVertices
global size	nVertices	**local size**	None	

Argument	In/Out	Memory space	Data type	Size
*delta_new	input	global	float32	1
*delta_old	both	global	float32	1
*beta	output	global	float32	1
global size	1	**local size**	1	

2.1.7 u0_update

Computes an initial guess for the next time step, u_0$^+$, based on a linear extrapolation from the initial guess of the current time step, u_0 and the PCG solution of the current time step, u_new,

$$\text{u\_0}^+ = \text{u\_new} + (\text{u\_new - u\_0}).$$

Argument	In/Out	Memory space	Data type	Size
*u0	both	global	float32	nVertices
*u_new	input	global	float32	nVertices
global size	nVertices	**local size**	None	

2.1.8 MVP and Computing P

The implementation of matrix-vector product (including cases for $\mathbf{A}\vec{x}$ and $a\mathbf{A}\vec{x} + \vec{b}$) and the determination of P is dependent on the assembly perspective. I have three sets of kernels for this: a general DbD approach, and two fixed grid methods. The FG DbD implementation with memory coalescing is discussed in Section 3.

2.2 Memory

Some input variables to introduce:

- M and F: arrays of data for elemental assembly matrices. Either the single matrices for fixed grid methods, or all entries for every element for the general methods
- DoFMapLocal and zMapLocal: small arrays of indices based on the geometry of the mesh to quickly determine local DoF to element assignment
- C: array of the number of divisions of the domain in each dimension
- vert_scale: array of the minumum vertex position and spacing between vertices in each dimention. Along with C, this allows the determination of the absolute location of a vertex given its global index. For a non-fixed grid approach, I start with a uniform mesh and deform it in a predetermined way, so that information can be used inside a kernel to still enable the recovery of absolute vertex position.

- corr_bound: spatial information about what part of the domain is corroded (or in general, a different material). For uniform corrosion after a certain depth, this can be that distance in the z-direction. For elliptical corrosion pits, it could contain the coordinates of the center of the ellipse and its axis lengths. The kernel must be programmed to know how to interpret this.
- mat_coefs: values for ρC and k for the different materials

Memory is allocated in GPU global memory buffers with flags to specify how the host and the device will access them. The flags are self-explanatory, and are combined with logical OR. The combinations I use are

- HOST_TO_DEVICE_COPY = (READ_ONLY | HOST_WRITE_ONLY | COPY_HOST_PTR)
- HOST_TO_DEVICE_USE = (READ_ONLY | HOST_WRITE_ONLY | USE_HOST_PTR)
- HOST_READ_WRITE = (READ_WRITE | COPY_HOST_PTR)
- PINNED = (READ_WRITE | USE_HOST_PTR)
- DEVICE_READ_WRITE = (READ_WRITE | HOST_NO_ACCESS)

The memory buffers allocated for the FG DbD method are enumerated below. Other methods do not differ much at this level. The last three buffers are for intermediate values that will be covered in Section 3.

2.3 PCG Again

The kernels and memory usage is as follows, folowing the steps in Section 1.3. Again, steps related to matrix-vector multiplication will be discussed in Section 3. The syntax below is a small abbreviation of the actual PyOpenCL code, and has the form

```
kernel_instance = kernel_name(args)
```

First compute P and the right hand side vector $\vec{b} = \mathbf{L}\vec{u}_i + \vec{N}$.

```
knl_PA = Jacobi_A(M_FG_buf, K_FG_buf, DoFMapLocal2_buf,
          zMapLocal_buf, C_buf, vert_scale_buf, corr_bounds_buf, mat_coefs_buf,
          theta,  dt, Ax_split_local_buf, P_split_buf)

knl_PB = FGDbDMVP_B(P_split_buf, P_buf)

knl_RHS_A = FGDbDMVP_A(M_FG_buf, K_FG_buf, u0_buf, DoFMapLocal2_buf,
         zMapLocal_buf, C_buf, vert_scale_buf, corr_bounds_buf, mat_coefs_buf,
         (1-theta), dt, x_local_buf, Ax_split_local_buf, Ax_split_buf)
knl_RHS_B = FGDbDMVP_C(Ax_split_buf, Ndt_buf, np.float32(1), b_buf)
```

Setup for PCG

```
knl_1A = FGDbDMVP_A(M_FG_buf, K_FG_buf, x_buf, DoFMapLocal2_buf,
         zMapLocal_buf, C_buf, vert_scale_buf, corr_bounds_buf,
         mat_coefs_buf, theta, dt, x_local_buf, Ax_split_local_buf, Ax_split_buf)
knl_1B = FGDbDMVP_C(Ax_split_buf, b_buf, np.float32(-1), r_buf)

knl_2 = DIMVP(P_buf, r_buf, d_buf)

knl_3A = VVP_A(r_buf, d_buf, nVertices, VVP_loc_buf, r1_buf)
knl_3B = VVP_reduce(r1_buf, r1_size, VVP_loc_buf, r2_buf)
```

Name	Flag	Initialization
M_FG_buf	HOST_TO_DEVICE_USE	M_FG
K_FG_buf	HOST_TO_DEVICE_USE	K_FG
DoFMapLocal2_buf	HOST_TO_DEVICE_COPY	DoFMapLocal2
zMapLocal_buf	HOST_TO_DEVICE_COPY	zMapLocal
C_buf	HOST_TO_DEVICE_COPY	C
vert_scale_buf	HOST_TO_DEVICE_COPY	vert_scale
corr_bounds_buf	HOST_TO_DEVICE_COPY	corr_bounds
mat_coefs_buf	HOST_TO_DEVICE_COPY	mat_coefs
P_buf	DEVICE_READ_WRITE	x.nbytes
u0_buf	HOST_TO_DEVICE_COPY	u0
Ndt_buf	HOST_TO_DEVICE_COPY	Ndt
x_buf	HOST_READ_WRITE	x
b_buf	DEVICE_READ_WRITE	x.nbytes
r_buf	DEVICE_READ_WRITE	x.nbytes
d_buf	DEVICE_READ_WRITE	x.nbytes
q_buf	DEVICE_READ_WRITE	x.nbytes
s_buf	DEVICE_READ_WRITE	x.nbytes
delta_buf	PINNED	delta
alpha_buf	DEVICE_READ_WRITE	delta.nbytes
neg_alpha_buf	DEVICE_READ_WRITE	delta.nbytes
delta_new_buf	DEVICE_READ_WRITE	delta.nbytes
beta_buf	DEVICE_READ_WRITE	delta.nbytes
r1_buf	DEVICE_READ_WRITE	r1_size*4
r2_buf	DEVICE_READ_WRITE	r2_size*4
VVP_loc_buf	LocalMemory	max_wg_size*4
Ax_split_buf	DEVICE_READ_WRITE	x.nbytes*4
P_split_buf	DEVICE_READ_WRITE	x.nbytes*4
x_local_buf	LocalMemory	32*4*4

```
knl_3C = VVP_reduce(r2_buf, r2_size, VVP_loc_buf, delta_buf)

knl_4A = FGDbDMVP_A(M_FG_buf, K_FG_buf, d_buf, DoFMapLocal2_buf,
          zMapLocal_buf, C_buf, vert_scale_buf, corr_bounds_buf,
          mat_coefs_buf, theta, dt, x_local_buf, Ax_split_local_buf, Ax_split_buf)
knl_4B = FGDbDMVP_B(Ax_split_buf, q_buf)

knl_5A = VVP_A(d_buf, q_buf, nVertices, VVP_loc_buf, r1_buf)
knl_5B = VVP_reduce(r1_buf, r1_size, VVP_loc_buf, r2_buf)
knl_5C = VVP_C(r2_buf, delta_buf, r2_size, VVP_loc_buf, alpha_buf, neg_alpha_buf)

knl_6 = VAVSP(x_buf, d_buf, alpha_buf, x_buf)

knl_7A = FGDbDMVP_A(M_FG_buf, K_FG_buf, x_buf, DoFMapLocal2_buf,
          zMapLocal_buf, C_buf, vert_scale_buf, corr_bounds_buf,
          mat_coefs_buf, theta, dt, x_local_buf, Ax_split_local_buf, Ax_split_buf)
knl_7B = FGDbDMVP_C(Ax_split_buf, b_buf, np.float32(-1), r_buf)

knl_7 = VAVSP(r_buf, q_buf, neg_alpha_buf, r_buf)
```

```
knl_8 = DIMVP(P_buf, r_buf, s_buf)

knl_9A = VVP_A(r_buf, s_buf, nVertices, VVP_loc_buf, r1_buf)
knl_9B = VVP_reduce(r1_buf, r1_size, VVP_loc_buf, r2_buf)
knl_9C = VVP_reduce(r2_buf, r2_size, VVP_loc_buf, delta_new_buf)

knl_10 = Beta_update(delta_new_buf, delta_buf, beta_buf)

knl_11 = VAVSP(s_buf, d_buf, beta_buf, d_buf)

knl_u0 = u0_update(u0_buf, x_buf)
```

The CPU handles logical decisions and calls these kernels according to the algorithm until convergence is realized.

3 FG DbD with memory coalescing

Details of the fixed grid DbD approach to computing a matrix-vector product are discussed here.

3.1 Mesh Geometry

A 3D regular mesh with linear tetrahedral elements is generated based on the domain boundaries and the number of divisions in each dimension. If the number of divisions are such that each increment is the same length in every dimension, the domain is then divided into cubes according to these divisions (otherwise, rectangular prisms, but we'll assume cubes here). Each cube is then subdivided into six tetrahedra, as seen in figure 2.

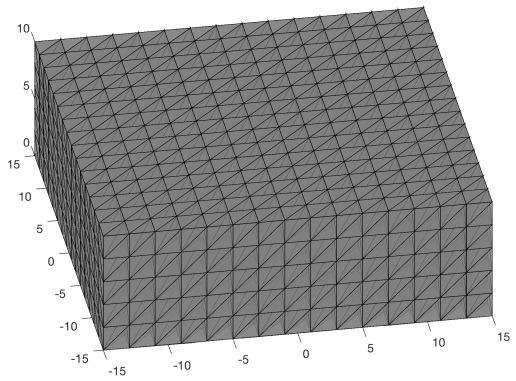

Figure 2: 30×30×10 mm domain with tetrahedral meshing.

The six tetrahedra within each cube are indexed in a sequential way—six in the first cube, then six in the next cube in the x direction, etc. until the y dimension is incremented, and then the next slice in the z dimension begins after that. An emphasized view of the six tetrahedra is given in Figure

3. Information about which of the eight vertices of each cube (numbered in the same way) correspond to each element is kept in the array DoFMapLocal.

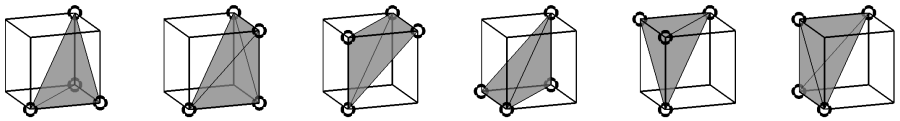

Figure 3: Emphasized view of the tetrahedral subdivisions of each small cube in the mesh.

3.2 Coalesced memory access

With this fixed grid approach to matrix-vector multiplication, the only data that must be loaded from global memory is the input vector. To do this efficiently, it must be loaded in blocks of 32 consecutive entries by blocks of 32 consecutive work items. A single memory read of this form does not give enough data to perform assembly for any element. However, if four blocks of memory are read, corresponding to the four horizontal edges of a long rectangular prism, then every tetrahedral element within that prism can be integrated over. That is, the contributions from each of these elements to the degrees of freedom that are loaded can be computed.
Using the memory access length of 32 as the guiding limit, 31 cubes of six tetrahedral elements will be integrable. Therefore, work groups of 186 work items are invoked—one work item for each tetrahedral element. Since only 128 work items are necessary to read data from global memory, some work items are assigned a dual purpose and some remain idle during the loading process. Those first 128 receive both a global elemental index for their integration responsibility as well as a global vertex index for loading data from global memory into local memory. Figure 4 depicts the way in which global data is accessed by each work item for one of the four edges of the long rectangular prism. The other edges are treated in the same way, with offset information determined in-kernel based on how many divisions are made in the domain in each dimension.

Every work group is responsible for finding elements contributions to a vertex in the $\pm x$ directions for one of the four quadrants to which it is connected in the y-z plane. Therefore only the 30 "internal" vertices from each block of 32 is contributed to in storage[5]. This partitioning pattern is also demonstrated in Figure 4.
Another consequence of coalesced memory access is that some consecutive elements of the input vector do not actually share an element in the domain. This is demonstrated in Figure 5 for a small example mesh. The problem is avoided by padding the domain with non-physical elements in the $+x$ and $+y$ directions for the sake of loading and computation, and then discarding their contributions upon storage of the product. This results in some amount of wasted computation, but the relative volume of padding elements decreases as the granularity of the mesh increases (surface area vs volume). Furthermore, effeciency gains from coalesced memory access are a much higher priority than losses from this wasted computation.

3.3 Kernels

The kernels for FG DbD MVP are discussed here.

[5]Except for the first workgroup, and possibly the last.

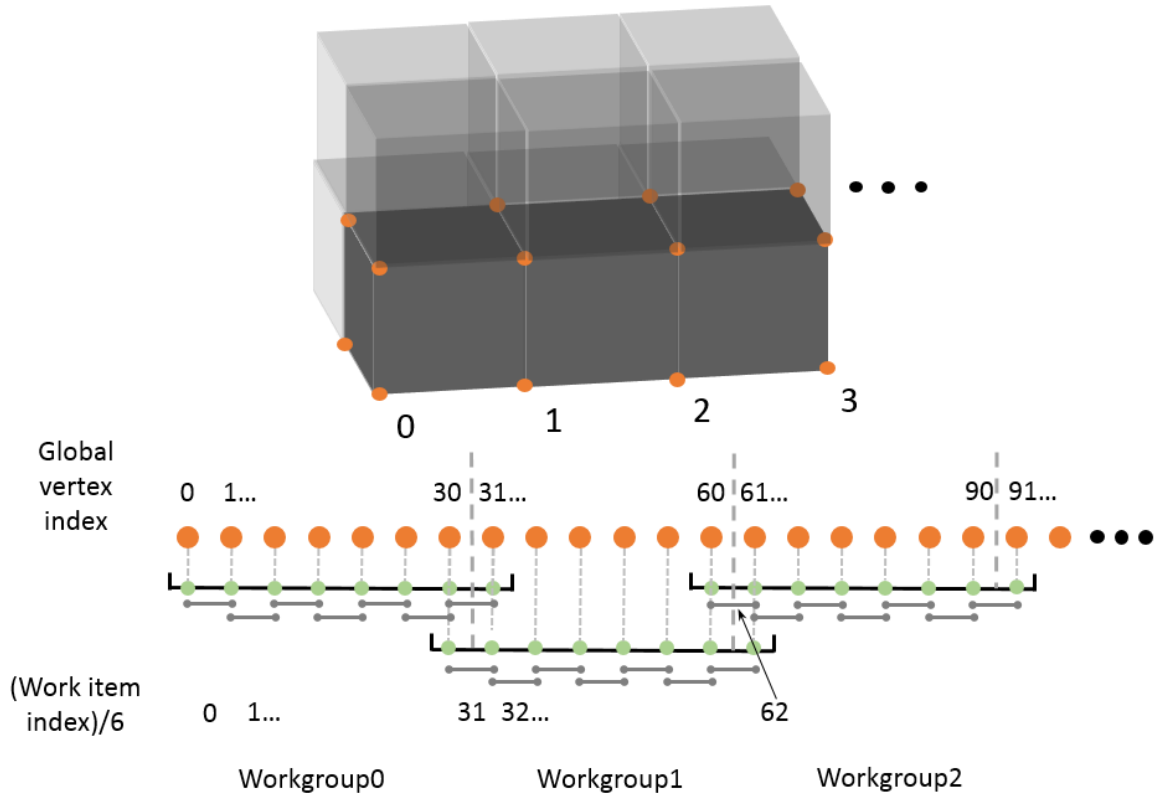

Figure 4: Memory access and computational partitioning pattern for the coalesced DbD MVP method.

3.3.1 FGDbDMVP_A

Computes contributions to a matrix-vector product from each element according to coalesced memory access limitations. Data is loaded from the input vector as described in Section 3.2 and stored in local memory. Then each work item determines its global element index by first finding the cube in which it belongs: global_id/6 - group_id[6], and then the tetrahedron within the cube: local_id mod 6.
The cube index also corresponds to the global vertex index for its first corner. This is used with the arrays C and vert_scale to determine the vertex's absolute position. Based on this, the array zMapLocal helps give the z positions of the other vertices of the tetrahedron, so that they can be compared with corr_bounds to determine for each vertex whether it is in the corroded region of the domain or not. Material coefficients are taken from mat_coefs and averaged over the element for both $\mathbf{M}_e$ and $\mathbf{K}_e$.
Next, each element reads the data it needs about the input vector from local memory, referring to DoFMapLocal2 for the proper indices. This has been delayed as long as possible to hide the latency from the global memory load. Dot products are taken with the local assembly matrices, and the results are summed with the proper coefficients in another local memory array, Ax_split_local, which has 12 entries for every DoF, corresponding to the 12 possible contributing elments in the $\pm x$ directions. Finally, the work items that are responsible for loading and storing data sum over Ax_split_local for their DoF and write to global memory. The resulting memory buffer has 4 entries for every global DoF, one for every quadrant in the y-z plane. These are all filled out in turn by further work groups. The total number of work items needed for this kernel is found by determining the total number of elements that need to be considered including the padding in $+x$ and $+y$ directions, dividing by 30 since every work group yields the elemental contributions for blocks of 30 vertices, and multiplying by

[6]global_id is an unsigned int, so integer divisionautomatically rounds down.

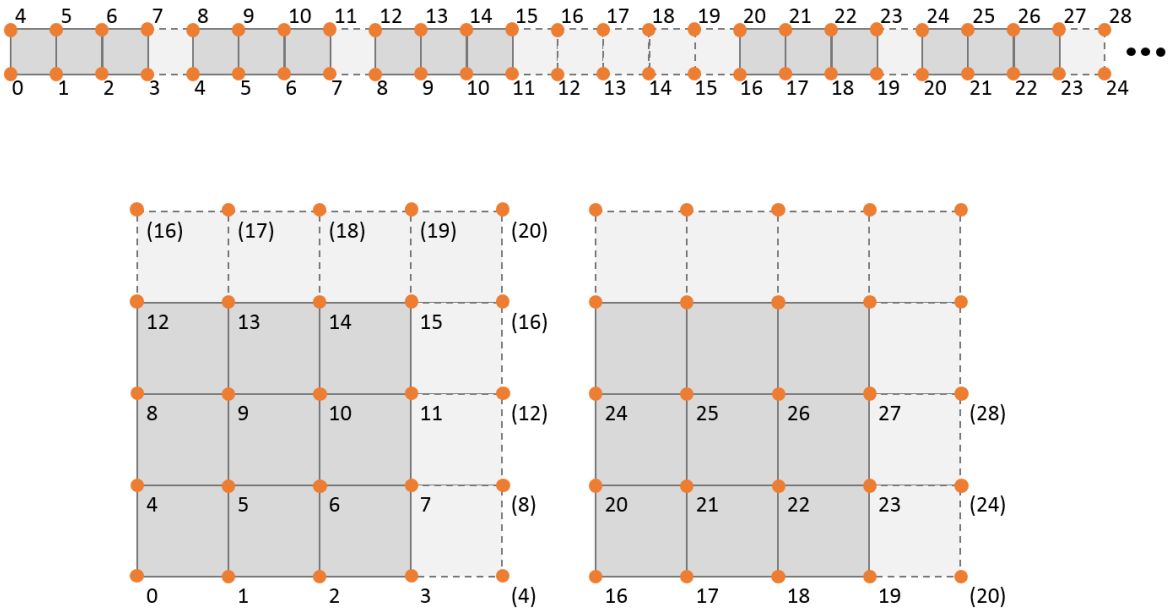

Figure 5: Element padding for a 3×3×n mesh as seen from a top-down perspective. All of the data is loaded, but only contributions from solid elements are stored.

the work group size. As with vector-vector products, we round up the integer division

$$\text{MVP\_global\_size} = 186 \left\lceil \frac{6(C[0]+1)(C[1]+1)C[2]}{180} \right\rceil .$$

Argument	In/Out	Memory space	Data type	Size
*M_FG	input	constant	float32	4
*K_FG	input	constant	float32	4
*x	input	global	float32	nVertices
*DoFMapLocal2	input	constnat	uint32	12
*zMapLocal	input	constant	float32	12
*C	input	constant	uint32	3
*vert_scale	input	constant	float32	6
*corr_bounds	input	constant	float32	
*mat_coefs	input	constant	float32	4
theta	input		float32	1
dt	input		float32	1
*x_local		local	float32	4*32
*Ax_split_local		local	float32	4*32*12
*Ax_split	output	global	float32	nVertices*4
global size	MVP_global_size	**local size**	186	

3.3.2 FGDbDMVP_B

Finishes the matrix-vector multiplication started by FGDbDMVP_A. Work items sum the four contributions to each DoF from Ax_split and store them in the final result array.

Argument	In/Out	Memory space	Data type	Size
*Ax_split	input	global	float32	nVertices*4
*Ax	output	global	float32	nVertices
global size	MVP_global_size	**local size**	186	

3.3.3 FGDbDMVP_C

Finishes the matrix-vector multiplication started by FGDbDMVP_A with an extra SAXPY operation so that a separate call to VAVSP is not necessary. Work items sum the four contributions to each DoF from Ax_split, multiply them by a scalar, add them to an element from another vector, and store the result.

Argument	In/Out	Memory space	Data type	Size
*Ax_split	jiput	global	float32	nVertices*4
*b	input	global	float32	nVertices
c	input		float32	1
*cAx_plus_b	output	global	float32	nVertices
global size	MVP_global_size	**local size**	186	

3.3.4 Jacobi_A

Determines contributions to the Jacobi preconditioner P. The algorithm is the same as FGDbDMVP_A, except instead of taking dot products with elemental assembly matrices and an input vector, the diagonal elements of the elemental assembly matices are scaled according to material coefficients, combined, and stored. Calling FGDbDMVP_B on the result produces the diagonal of P.

Arguments for this kernel are the same as for FGDbDMVP_A, except without *x and *x_local.

4 Dual GPU Adaptation

The fixed grid coalesced method described above is modified for use on dual GPU. The description below applies to two devices, but can be extended to more with little extra analysis.

The domain is split in the z direction according to the additive Schwartz method. This keeps all vertices in each subdomain in adjacent blocks. A fraction of the domain to be assigned to device 1, m, is specified, typically 0.5. This fraction of z slices, rounded up, with one extra layer of vertices is the number of vertices that device 1 is responsible for, m_1. The rest of the vertices, in addition to two overlapping layers are assigned to device 2, totaling m_2. This split is demonstrated in Figure 6.

If these vectors are initialized from a full set of global data, then one matrix-vector product can be performed on each device, and the result can be faithfully reconstructed. For more than one matrix-vector product, the shared boundary data must be updated. Only one layer of vertices needs to be transfered in each direction. In the PCG algorithm, the iterating solution vector x is initialized at the beginning, and the intermediate vector d must be transfered at each iteration.

The scalar results of dot products must also be communicated between devices, both the residual `delta` and step size `alpha`. To do this, each partial dot product is handled separately, omitting the extra boundary vertices. The partial results are stored in their own buffers, which are then transfered both directions. We thus have four buffers for each scalar quantity, one native buffers for the partial results on the device which computed it, and one target buffer on each device to store the copied value from the other. Many permutations of methods for the bidirectional communication of partial dot products were considered. This method was found to be the fastest reliable way for each device to

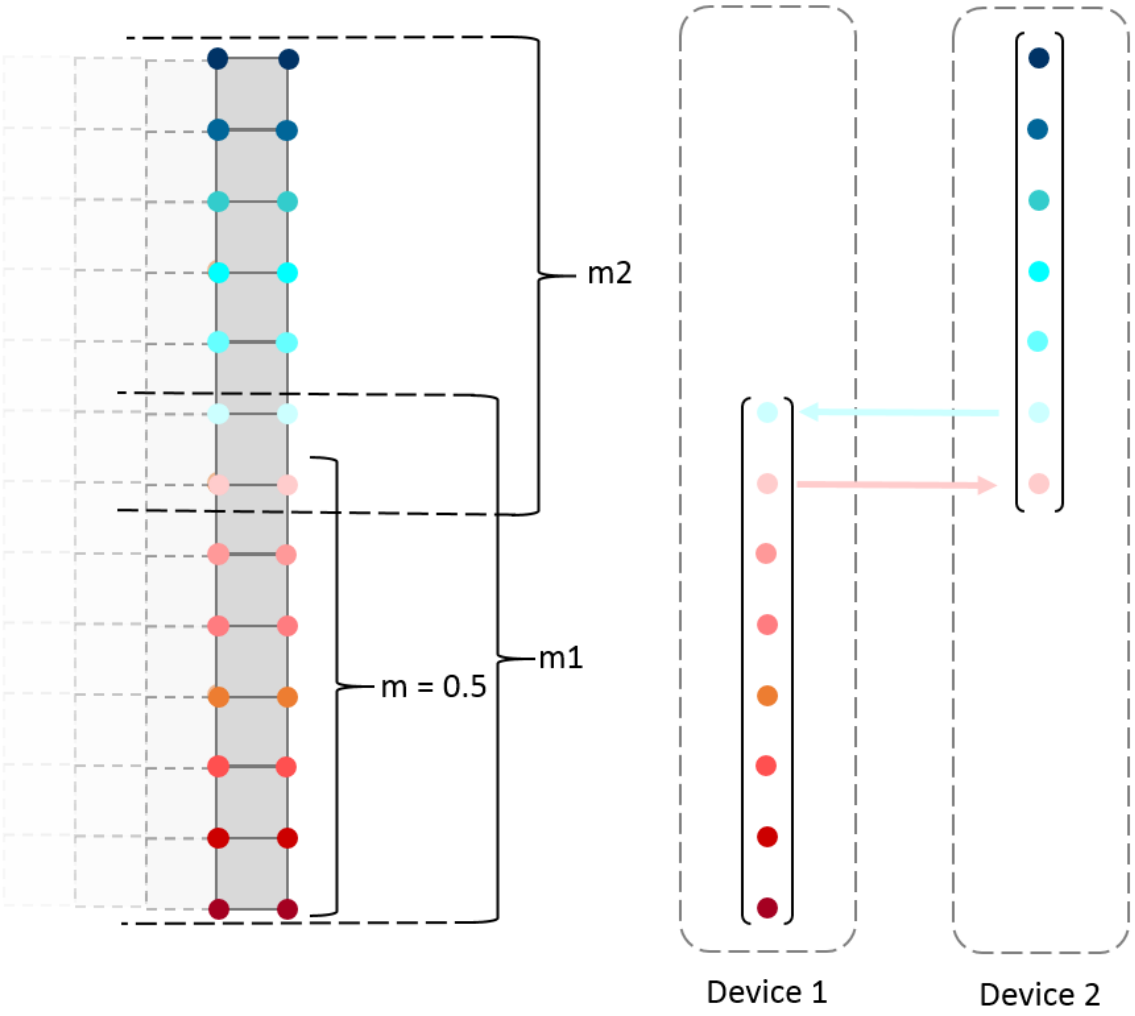

Figure 6: (left) splitting of vertices between two devices according to a user-specified fraction m. (right) Data that must be transfered between devices at each iteration.

receive the full results. No special kernel is needed to combine them. Rather, modifications to VAVSP are made to accept both buffers on a device and sum them as part of the same process.

The only remaining matter for modification is adjusting the memory objects that contain vertex-to-spatial-location information. One each is made for the two devices, with the second encoding the z shift for its first index. Then each devices runs PCG in parallel, and can contextualize the location of vertices within the total domain. The host initialized two sets of every kernel that has been described for the serial method, as well as launching one queue to transfer boundary and scalar information and properly wait.