Solving 2D the Poisson Equation Using SBP-SAT Finite Difference Operators in Parallel

Alex Chern, Toby Harvey

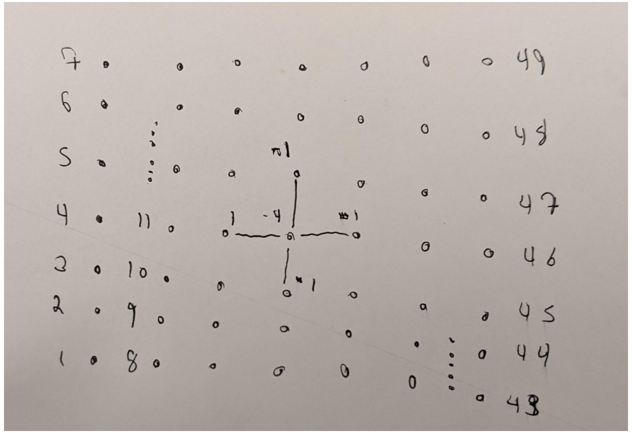
PDE Background

- Earthquake modeling with Poisson's equation
- Hooke's law and Conservation of Momentum
- Neglecting acceleration taking second time derivative to be zero results in Poisson's Equation:

$$c_1 \frac{\partial^2 u}{\partial x^2} + c_2 \frac{\partial^2 u}{\partial y^2} + F(x, y, t) = 0$$

- Goal of solving the equation is to find and equation u that satisfies the equation.
- Method of Manufactured Solutions (MMS): solution -> source functions + boundary conditions (used in solving equations)

Stencil Computations in 2D and Method of Lines



$$\begin{bmatrix} \begin{bmatrix} \vdots \\ u^1 \\ \vdots \end{bmatrix} \\ \begin{bmatrix} \vdots \\ u^2 \\ \vdots \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \vdots \\ u^n \\ \vdots \end{bmatrix} \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ \vdots \\ \vdots \\ u_{n^2-2} \\ u_{n^2-1} \\ u_n \end{bmatrix}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta x^2}$$

SBP-SAT

- Summation by Parts (SBP):
  - Would like to ensure that spatial discretization results in a “stable” system.
  - Take energy estimate of differential equation (advection example): $\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0$

$$\frac{d}{dt}\int u^2 dx = \int \frac{\partial u^2}{\partial t} dx = \int 2u\frac{\partial u}{\partial t} dx = 2\int u - a\frac{\partial u}{\partial t} = -a(U_r^2 - u_l^2)$$

Have discrete operators mimic continuous energy.
Need numerical quadrature, and boundary operators.

- Simultaneous approximation terms (SAT)
  - Injection destroys SBP
  - SATs minimize dependence between blocks of discretizations (good for parallelizing).
  - Can help model discontinuous physical properties.

Time and Space Complexity of Implementations

- Must solve Ax = b
- LU decomposition: Does not necessarily maintain sparsity. For big problems there could be storage constraints. Faster than CG a lot of the time. O(n^3) for factorization and O(n^2) for solve but not parallelizable (efficiently).
- CG with sparse matrices (CPU and GPU): Only need to store sparse A matrix and b resulting in better storage. Potentially slower than LU.
- CG matrix-free (CPU and GPU): Instead of storing A, just write functions that act on x, the way A would. Only storage constraint is b and intermediate arrays.

Implementations

CPU (Toby Harvey): Sparse Matrix Vector Product (SpMV, CSR), Matrix-Free

GPU (Alexandre Chen): Matrix-Free with Shared_memory, SpMV (using Toby's CPU CSR data and SpMV_GPU() function)

Performance Analysis (Both):

CPU implementation

- Construct all operators (embarrassingly parallel for matrix verison)
- Use parallel spmv within CG kernel, otherwise call matrix free operator which takes in the vector pk produces how A acts on in Ap.

```
spmv(csr_row_ptr, csr_col_ind, csr_vals, m, n, nnz, pk, Ap);
Amv (n, pk, Ap);
```

GPU kernels

Naive kernel

1. 1D kernel
2. No shared memory
3. Using global index to determine boundary and interior points

GPU kernels

```
template <class T>
__global__ void
D2x_kernel(T* idata, T* odata, int Nx, int Ny, double h)
{
   unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int N;
   N = Nx * Ny;
   if (idx < Ny)
   {
       odata[idx] = (idata[idx] - 2*idata[idx + Ny] + idata[idx + 2*Ny]) / (h*h);
   }
   if ((Ny <= idx) && (idx < N - Ny))
   {
       odata[idx] = (idata[idx - Ny] - 2*idata[idx] + idata[idx + Ny]) / (h*h);
   }
   if ((idx >= N - Ny) && (idx < N))
   {
       odata[idx] = (idata[idx - 2*Ny] - 2*idata[idx - Ny] + idata[idx]) / (h*h);
   }

   __syncthreads();
}
```

GPU kernels

2D shared memory kernels

1. Mapping: 2D real domain to 2D block and tiles.
2. Reading global data to static shared memory and doing calculation using shared memory
3. Global index with local thread index for determining boundary and interior points

2D static shared memory

```
D2x_shared_kernel(T* idata, T* odata, int Nx, int Ny, double h)
{
    const int HALO_WIDTH = 2;
    const int TILE_DIM_1 = 4; // will be set globally
    const int TILE_DIM_2 = 4;
    __shared__ double tile[TILE_DIM_1][TILE_DIM_2 + 2 * HALO_WIDTH];  // calculation in x direction
```

Mapping

```
unsigned tidx = threadIdx.x;
unsigned tidy = threadIdx.y;

unsigned i = blockIdx.x * blockDim.x + tidx;    // global index in x direction
unsigned j = blockIdx.y * blockDim.y + tidy;    // global index in y direction

unsigned global_index = i + j * Ny;            // global index for stacked array

// for tile indexing in shared memory
unsigned int k = tidx;
unsigned int l = tidy;
```

Loading data from global memory into shared memory

```
 // for tile itself
if ((k < TILE_DIM_1) && (l < TILE_DIM_2) && (i < Ny) && (j < Nx))
{
    tile[k][l+HALO_WIDTH] = idata[global_index];    //   left halo left empty
}
// for left halo
if ( (k < TILE_DIM_1) && (l < HALO_WIDTH) && (i < Ny) && (  j >= HALO_WIDTH ) && (j < HALO_WIDTH + Nx))
{
    tile[k][l] = idata[global_index - HALO_WIDTH * Ny];   //   left-most tile doesn't need to fill left halo
}
// for right halo
if ((k < TILE_DIM_1) && (l >= TILE_DIM_2 - HALO_WIDTH) && (l < TILE_DIM_2) && (i < Ny) && (   j < Nx - HALO_WIDTH ))
{
    tile[k][l+2*HALO_WIDTH] = idata[global_index + HALO_WIDTH*Ny]; //   right-most tile doesn't need to fill right
halo
}
__syncthreads();
```

Calculating Finite Difference Using Data loaded to shared memory

```
// left boundary for second order
   if ((k < TILE_DIM_1) && (l + HALO_WIDTH < TILE_DIM_2 + 2*HALO_WIDTH - 2) && (i < Ny) && (j == 0)){
       odata[global_index] = (tile[k][l + HALO_WIDTH] - 2*tile[k][l+HALO_WIDTH+1] + tile[k][l+HALO_WIDTH+2]) / (h*h);
   }  // We need to make sure tile index and global_index satisfy certain criteria

   // center
   if ((k < TILE_DIM_1) && (l + HALO_WIDTH < TILE_DIM_2 + 2*HALO_WIDTH -1) && (i < Ny) && (j >= 1) && (j < Nx - 1)) {
       odata[global_index] = (tile[k][l + HALO_WIDTH - 1] - 2*tile[k][l+HALO_WIDTH] + tile[k][l+HALO_WIDTH + 1]) / (h*h);
   }

   // right boundary for second order
if ((k < TILE_DIM_1) && (2 <= l + HALO_WIDTH) && (l + HALO_WIDTH < TILE_DIM_2 + 2*HALO_WIDTH) && (i < Ny) && (j == Nx-1)){
       odata[global_index] = (tile[k][l+HALO_WIDTH - 2] - 2*tile[k][l + HALO_WIDTH - 1] + tile[k][l+HALO_WIDTH])/ (h*h);
   }
```

Matrix-Free Function

1. One GPU function that calls multiple kernels for different operators to replace spmv_gpu() in CG.
2. Using vec_add_gpu() to merge all outputs. Might need to write multi_vec_add_gpu().
3. Ideal design:
   a. Merging D2x and D2y kernels into one kernel with four halo regions.
   b. Merging boundary operators into four (Dx, Dy, Hxinv, …)
   c. Reducing memory allocations for intermediate results
   d. Doing boundary operations on CPU?

Current GPU Matrix-Free Function:

```
void matrix_free(int Nx, int Ny, double h, double* dx, double* db, double* ...) // more variables
{
   unsigned TILE_DIM_1 = 4;
   unsigned TILE_DIM_2 = 4; // we could explore different tile shapes

   dim3 dimBlock_2d_x(TILE_DIM_1, TILE_DIM_2, 1);
   dim3 dimGrid_2d_x(Nx/TILE_DIM_1 + 1, Ny/TILE_DIM_2 + 1,1);

   dim3 dimBlock_2d_y(TILE_DIM_2, TILE_DIM_1, 1);
   dim3 dimGrid_2d_x(Nx/TILE_DIM_1 + 1, Ny/TILE_DIM_2 + 1,1);

   D2x_shared_kernel<double><<<dimGrid_2d_x, dimBlock_2d_x>>>(dx, D2x_out, Nx, Ny, h);
   D2y_shared_kernel<double><<<dimGrid_2d, dimBlock_2d>>>(dx,D2y_out,Nx,Ny,h);
   vec_add_gpu(Nx*Ny, 1.0, D2x_out, D2y_out, D2_out); // Combine D2x_out and D2y_out;
   …

}
```

Progress So far

1. All kernel functions implemented and tested.
2. Prototype for one big Matrix-free GPU function
3. Convergence test on CPU code …

To Do

1. Finished implementation by Dec 3rd.
2. More test for performance and convergence for project paper.
3.