CSE 4373/5373 - General Purpose GPU Programming

GPU Pattern: Stencils

Alex Dillhoff

University of Texas at Arlington

Differential Equations

Any computational problem requires discretization of data or equations so that they can be solved numerically.

This is fundamental in numerical analysis, where differential equations need to be approximated.

Differential Equations

- Structured grids are used in finite-difference methods for solving partial differential equations (PDEs).
- Approximate derivatives can be computed point-wise by considering the neighbors on the grid.
- A grid representation is a natural way to think about data parallelism.

Differential Equations

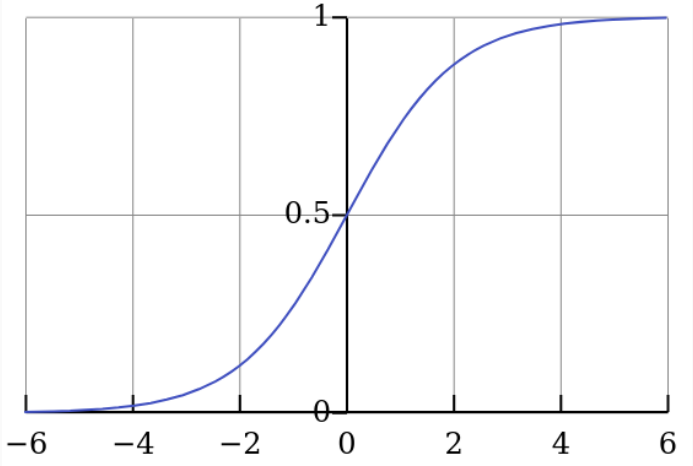
- Depending on the function and level of discretization, interpolation will be more or less accurate.
- Consider the logistic sigmoid function sampled at 4 points.
- In the middle, linear interpolation would work just fine.
- Near the *bends* of the function, a linear approximation would introduce error.

Differential Equations

The closer the spacing, the more accurate a linear approximation becomes.

The downside is that more memory is required to store the points.

Differential Equations



Logistic sigmoid function (Wikipedia).

Differential Equations

The precision of the data type also plays an important role.

Higher precision data types like `double` require more bandwidth to transfer and will typically require more cycles when computing arithmetic operations.

Stencils

Stencils

A **stencil** is a geometric pattern of weights applied at each point of a structured grid.

The points on the grid will derive their values from neighboring points using some numerical approximation.

Stencils

Consider a 1D grid of discretized values from a function $f(x)$.

The finite difference approximation can be used to find $f'(x)$:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

Stencils

In code, this would look like:

```
__global__ void f(float *f, float *df, float h) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    df[i] = (f[i+1] - f[i-1]) / (2 * h);
}
```

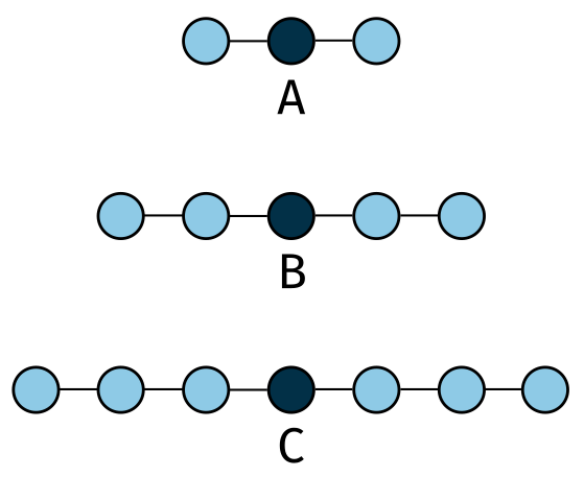
Stencils

A PDE of two variables can be solved using a 2D stencil.

Likewise, a PDE of three variables can be solved using a 3D stencil.

They typically have an odd number of points so that there is a center point.

Stencils

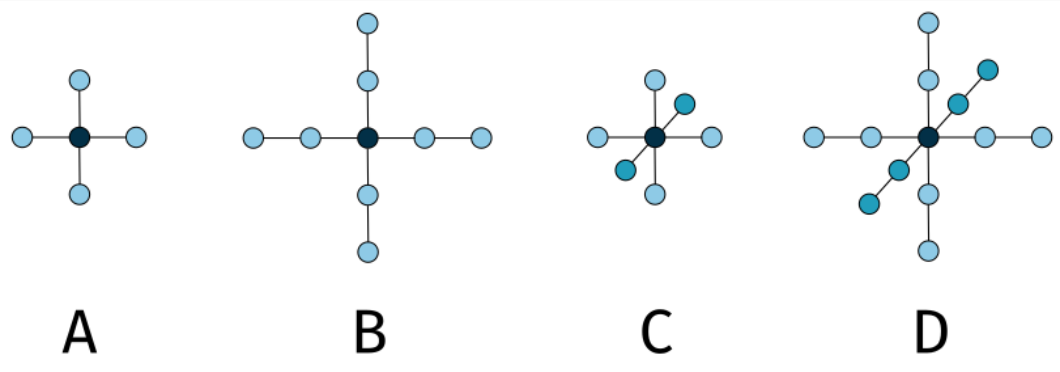


1D stencils.

Stencils

The 1D stencils shown above are used to approximate the first-order (A), second-order (B), and third-order (C) derivatives of a function $f(x)$.

Stencils



2D and 3D stencils.

Stencils

The 2D stencils shown above are used to approximate first-order (A) and second-order (B) derivatives of a function $f(x,)$. Likewise, the 3D stencils are used to approximate first-order (C) and second-order (D) derivatives of a function $f(x, y, z)$.

Stencils

A *stencil sweep* is the process of applying the stencil to all points on the grid, similar to how a convolution is applied.

There are many similarities between the two, but the subtle differences will require us to think differently about how to optimize them.

3D seven-point stencil

```
__global__ void stencil_kernel(float *in, float *out, unsigned int N) {
    unsigned int i = blockIdx.z * blockDim.z + threadIdx.z;
    unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int k = blockIdx.x * blockDim.x + threadIdx.x;

    if (i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
        out[i*N*N + j*N + k] = c0 * in[i*N*N + j*N + k]
                             + c1 * in[i*N*N + j*N + (k - 1)]
                             + c2 * in[i*N*N + j*N + (k + 1)]
                             + c3 * in[i*N*N + (j - 1)*N + k]
                             + c4 * in[i*N*N + (j + 1)*N + k]
                             + c5 * in[(i - 1)*N*N + j*N + k]
                             + c6 * in[(i + 1)*N*N + j*N + k];
    }
}
```

3D seven-point stencil

We have two major concerns:

- Memory access patterns
- Arithmetic intensity

For memory patterns, we can immediately review a tiled version of the kernel.

3D seven-point stencil

We start with a shared array and new indexing scheme.

```
__shared__ float tile[IN_TILE_DIM][IN_TILE_DIM][IN_TILE_DIM];
int i = blockIdx.z * OUT_TILE_DIM + threadIdx.z - 1;
int j = blockIdx.y * OUT_TILE_DIM + threadIdx.y - 1;
int k = blockIdx.x * OUT_TILE_DIM + threadIdx.x - 1;
```

3D seven-point stencil

Then, the data is loaded into the shared array.

```
if (i >= 0 && i < N && j >= 0 && j < N && k >= 0 && k < N) {
    tile[threadIdx.z][threadIdx.y][threadIdx.x] = in[i*N*N + j*N + k]
}
```

3D seven-point stencil

The boundary check for a sparse grid is a bit more complex.

```
if (i >=1 && i < N-1 && j >= 1 && j < N-1 && k >= 1 && k < N-1) {
    if (threadIdx.z >= 1 && threadIdx.z < IN_TILE_DIM-1 &&
        threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM-1 &&
        threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM-1) {
```

3D seven-point stencil

Finally, the result is computed.

```
out[i*N*N + j*N + k] = c0 * tile[threadIdx.z][threadIdx.y][threadIdx.x]
                     + c1 * tile[threadIdx.z][threadIdx.y][threadIdx.x - 1]
                     + c2 * tile[threadIdx.z][threadIdx.y][threadIdx.x + 1]
                     + c3 * tile[threadIdx.z][threadIdx.y - 1][threadIdx.x]
                     + c4 * tile[threadIdx.z][threadIdx.y + 1][threadIdx.x]
                     + c5 * tile[threadIdx.z - 1][threadIdx.y][threadIdx.x]
                     + c6 * tile[threadIdx.z + 1][threadIdx.y][threadIdx.x];
```

3D seven-point stencil

Stencil patterns are more sparse than convolutional filters and require less data to be loaded into shared memory.

This will be the central detail in our analysis of stencil patterns.

Stencil size vs. Convolution size

- Convolutions are more efficient as the size increases since more values are accessed in shared memory.
- For a 3×3 convolution, the upper bound on compute to memory ratio is 4.5 OP/B.
- A 5 point 2D stencil has a ratio of 2.5 OP/B due to the sparsity of the pattern.
- The threads in the block would load the diagonal values from global memory, but each thread would only use the 5 points defined by the kernel.

Tiling Analysis

Let us consider the effectiveness of shared memory tiling where each thread performs 13 floating-point ops (7 multiplies and 6 adds) with each block using $(T-2)^3$ threads.

The $T-2$ factor comes from the fact that the output size is smaller than the input size by 2 in each dimension.

Tiling Analysis

Each block also performs T^3 loads of 4 bytes each.

The compute to memory ratio can be express as:

$$\frac{13(T-2)^3}{4T^3} = \frac{13}{4}\left(1-\frac{2}{T}\right)^3$$

Tiling Analysis

Due to the low limit on threads, the size of T is typically small.

This means there is a smaller amount of reuse of data in shared memory.

The ratio of floating-point ops to memory accesses will be low.

Tiling Analysis

Each warp loads values from 4 **distant** locations in global memory.

This means that the memory accesses are not coalesced: the memory bandwidth is low.

Tiling Analysis

Consider an $8 \times 8 \times 8$ block.

A warp of 32 threads will load 4 rows of 8 values each.

The values within each row contiguous, but the rows are not contiguous.

Thread Coarsening

Thread Coarsening

Stencils do not benefit from shared memory as much as convolutions due to the sparsity of the sampled points.

Most applications of the stencil patterns work with a 3D grid, resulting in relatively small tile sizes per block.

Thread Coarsening

A solution to this is to increase the amount of work each thread performs, AKA *thread coarsening*.

The price paid for parallelism in the stencil pattern is the low frequency of memory reuse.

Thread Coarsening

- Each thread performs more work in the z direction for a 3D seven-point stencil.
- All threads collaborate to load in a z layer at time from $z-1$ to $z+1$.
- There are then 3 different shared memory tiles per block.

Thread Coarsening

- After computing values in the current output tile, the shared memory is rearranged for the next layer.
- This means that there are more transfers between shared memory as opposed to global memory.

Thread Coarsening

- The threads are launched to work with a 2D tile at a time, so the size of the block is now T^2.
- This means we can use a larger value for T.
- The compute to memory ratio is almost doubled under this scheme.
- Additionally, the amount of shared memory required is $3T^2$ rather than T^3.

Thread Coarsening

We need to establish the starting point for the z dimension and state the shared memory arrays.

```
int iStart = blockIdx.z * OUT_TILE_DIM;
int j = blockIdx.y * OUT_TILE_DIM + threadIdx.y - 1;
int k = blockIdx.x * OUT_TILE_DIM + threadIdx.x - 1;
__shared__ float inPrev_s[IN_TILE_DIM][IN_TILE_DIM];
__shared__ float inCurr_s[IN_TILE_DIM][IN_TILE_DIM];
__shared__ float inNext_s[IN_TILE_DIM][IN_TILE_DIM];
```

Thread Coarsening

Begin by loading the data into shared memory.

```
if (iStart - 1 >= 0 && iStart - 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
    inPrev_s[threadIdx.y][threadIdx.x] = in[(iStart - 1)*N*N + j*N + k];
}
if (iStart >= 0 && iStart < N && j >= 0 && j < N && k >= 0 && k < N) {
    inCurr_s[threadIdx.y][threadIdx.x] = in[iStart*N*N + j*N + k];
}
```

Thread Coarsening

The coarsening loop starts and the next slice is loaded into shared memory.

```
for (int i = iStart; i < iStart + OUT_TILE_DIM; i++) {
    if (i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
        inNext_s[threadIdx.y][threadIdx.x] = in[(i + 1)*N*N + j*N + k];
    }
    __syncthreads();
```

Thread Coarsening

The boundary checks are applied.

```
if (i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
    if (threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1 &&
        threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
```

Thread Coarsening

The output value is calculated, all from shared memory.

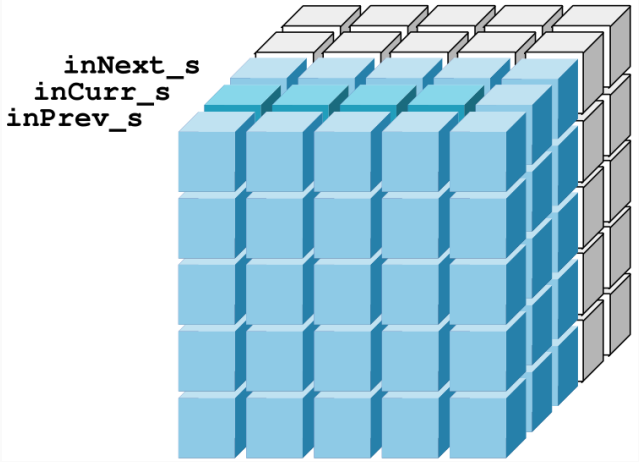
```
out[i*N*N + j*N + k] = c0 * inCurr_s[threadIdx.y][threadIdx.x]
+ c1 * inCurr_s[threadIdx.y][threadIdx.x - 1]
+ c2 * inCurr_s[threadIdx.y][threadIdx.x + 1]
+ c3 * inCurr_s[threadIdx.y - 1][threadIdx.x]
+ c4 * inCurr_s[threadIdx.y + 1][threadIdx.x]
+ c5 * inPrev_s[threadIdx.y][threadIdx.x]
+ c6 * inNext_s[threadIdx.y][threadIdx.x];
```

Thread Coarsening

The threads are synchronized before moving the data around in shared memory.

```
__syncthreads();
inPrev_s[threadIdx.y][threadIdx.x] = inCurr_s[threadIdx.y][threadIdx.x];
inCurr_s[threadIdx.y][threadIdx.x] = inNext_s[threadIdx.y][threadIdx.x];
```

Thread Coarsening



Thread coarsening tiles.

Thread Coarsening

- All of the blue blocks are loaded from global memory.
- The dark blue blocks represent the *active* plane that is used to compute the corresponding output values.
- After the current plane is completed, the block synchronizes before moving the current values to the previous plane and loads the next plane's values into the current plane `inCurr_s`.

Register Tiling

Register Tiling

- Each thread works with a single element in the previous and next shared memory tiles.
- There are 4 elements in that example that really need to be loaded from shared memory.
- For the 3 elements that are only required by the current thread, they can be loaded into registers.
- Since only the values in the $x - y$ direction are required for shared memory, the amount of memory used is reduced by $\frac{1}{3}$.

Register Tiling

Only a single shared memory array is required.

```
int iStart = blockIdx.z * OUT_TILE_DIM;
int j = blockIdx.y * OUT_TILE_DIM + threadIdx.y - 1;
int k = blockIdx.x * OUT_TILE_DIM + threadIdx.x - 1;
float inPrev;
float inCurr;
float inNext;

__shared__ float inCurr_s[IN_TILE_DIM][IN_TILE_DIM];
```

Register Tiling

The data is loaded into shared memory; `inCurr` is also loaded into shared memory.

```
if (iStart - 1 >= 0 && iStart - 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
    inPrev = in[(iStart - 1)*N*N + j*N + k];
}
if (iStart >= 0 && iStart < N && j >= 0 && j < N && k >= 0 && k < N) {
    inCurr = in[iStart*N*N + j*N + k];
    inCurr_s[threadIdx.y][threadIdx.x] = inCurr;
}
```

Register Tiling

The single z-value is loaded into a register before syncing.

```
for (int i = iStart; i < iStart + OUT_TILE_DIM; i++) {
    if (i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
        inNext = in[(i + 1)*N*N + j*N + k];
    }
    __syncthreads();
```

Register Tiling

The single z-value is loaded into a register before syncing.

```
for (int i = iStart; i < iStart + OUT_TILE_DIM; i++) {
    if (i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
        inNext = in[(i + 1)*N*N + j*N + k];
    }
    __syncthreads();
```

Register Tiling

The same boundary checks are applied before computing the output.

```
out[i*N*N + j*N + k] = c0 * inCurr
    + c1 * inCurr_s[threadIdx.y][threadIdx.x - 1]
    + c2 * inCurr_s[threadIdx.y][threadIdx.x + 1]
    + c3 * inCurr_s[threadIdx.y - 1][threadIdx.x]
    + c4 * inCurr_s[threadIdx.y + 1][threadIdx.x]
    + c5 * inPrev
    + c6 * inNext;
```

Register Tiling

The data is moved for the next iteration.

```
__syncthreads();
inPrev = inCurr;
inCurr = inNext;
inCurr_s[threadIdx.y][threadIdx.x] = inNext_s[threadIdx.y][threadIdx.x];
```

Summary

Summary

- Stencil patterns are used to approximate derivatives of functions on a structured grid.
- They are similar to convolutional filters, but are more sparse.
- The low memory reuse and sparsity of the pattern means that shared memory is not as effective as it is for convolutional filters.
- Thread coarsening and register tiling are two techniques that can be used to improve the performance of stencil patterns.

Summary

Applications

- Finite-difference methods for solving PDEs
- Image processing
- Fluid dynamics
- Seismic imaging
- Weather modeling