Derivative Discretization on GPUs

Paulius Micikevičius
NVIDIA

Manycore and Accelerator-based High-performance Scientific Computing
UC Berkeley, 2011

What this talk is about

- **Derivative discretization for FD methods**
  - Time domain
  - Explicit (derivatives approximated with stencils)
  - Examples assume second derivatives
    - Though other orders would be implemented exactly the same way
- **Goal: provide sufficient background so that a scientist can choose the right approach for the problem at hand**
  - Review implementation approaches and their tradeoffs
  - Some performance analysis
  - Experimental results showing throughputs
    - Reasonably optimized (as opposed to highly optimized)

Outline

- **Assumptions and definitions**
- **Relevant GPU details**
- **PDEs with derivatives in one dimension**
- **PDEs with derivatives in two dimensions**

Assumptions and definitions

- **Experimental setup:**
  - Fermi C2050, ECC off, 64-bit Linux, CUDA 3.2
- **3D data used in all experiments**
  - 512x512x512 (excluding the padding)
  - Results can be extrapolated for 1D and 2D data with the same number of elements
- **Dimensions: x, y, z**
  - x is the fastest varying, z is the slowest
- **Derivative discretization:**
  - Symmetric stencil with radius=R
    - Assumes isotropic medium and non-stretched grid
  - Number of stencil points:
    - 1D: 2R+1
    - 2D: 4R+1
    - 3D: 6R+1

Relevant GPU details

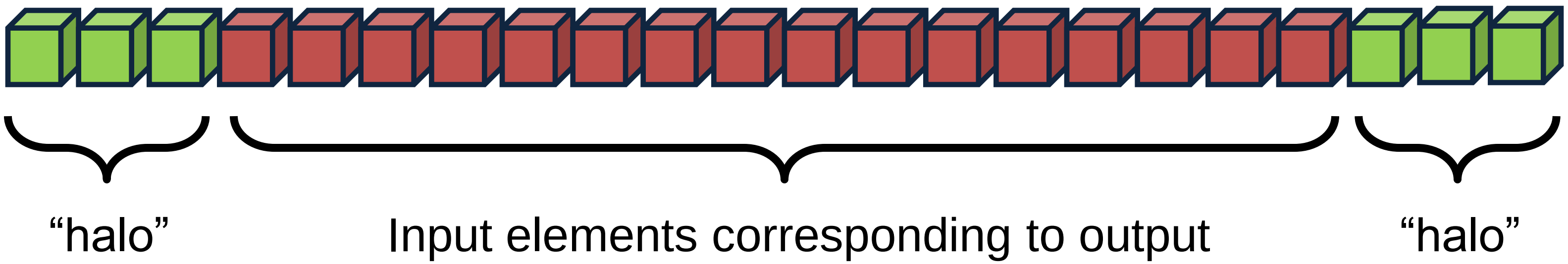
- **Memory accesses are per warp**
  - Warp = 32 threads
  - 32 addresses are converted into line requests
  - For max perf: an access by a warp should be within a line (or small number of lines)
- **GPUs need sufficient number of threads to saturate memory and instruction bandwidth**
  - ILP helps to an extent (Vasily Volkov's talk at GTC2010)
- **If there are barriers, it's often better to have a few smaller threadblocks concurrent per SM**
  - As opposed to one large one

PDEs with derivatives in 1 dimension

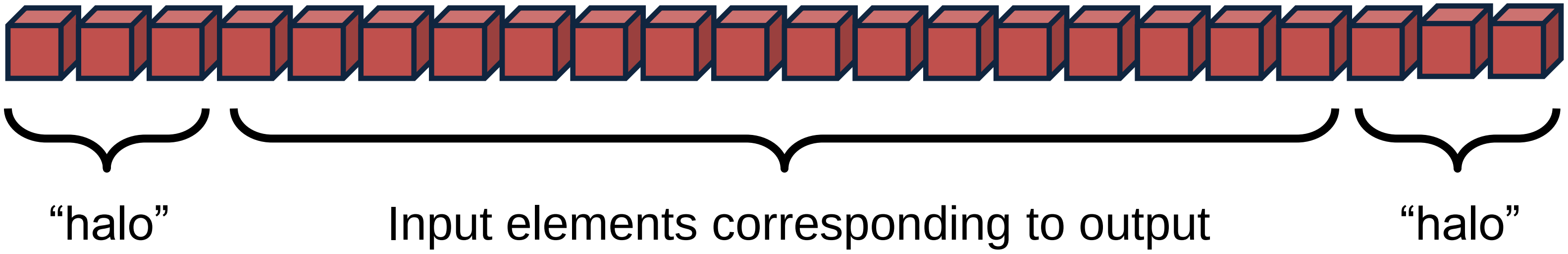
- **Two types of kernels**
  - Determined by stencil memory access pattern
- **Stencils along the fastest-varying dimension**
  - A thread needs a contiguous region of elements
  - Adjacent threads’ regions overlap
  - Staged through shared memory
- **Stencils along other dimensions**
  - Adjacent threads access adjacent elements
  - No region overlap
  - Straightforward “marching” along the dimension

Two approaches for x-stencils

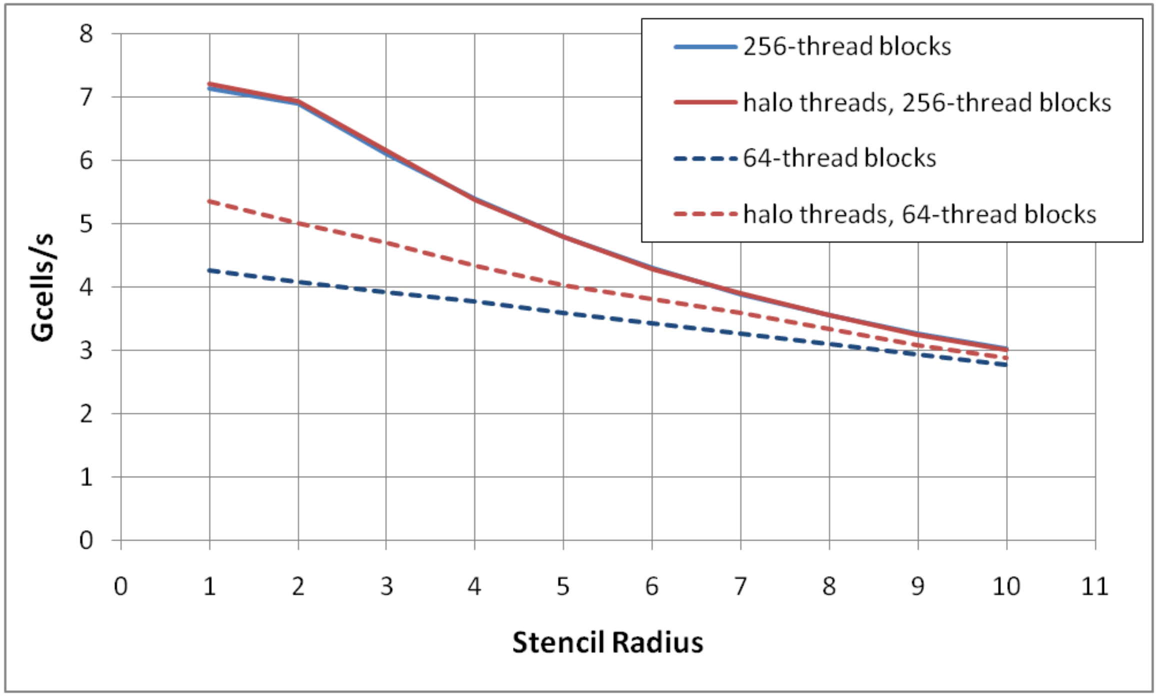
- **One thread per output element**
  - Some threads also fetch halos



- **One thread per input element**
  - Threads for halos as well (but don't compute or write)



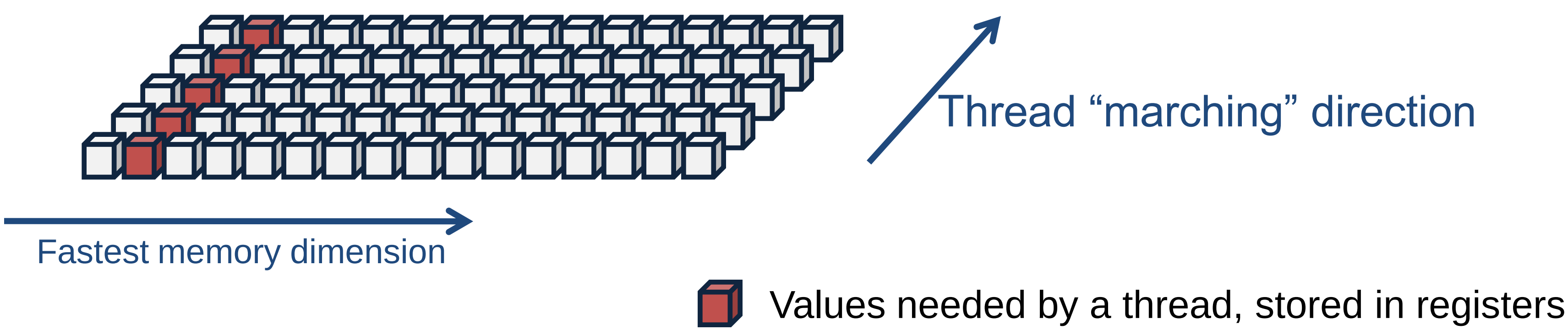
X-stencil performance



- **256- vs 64-thread blocks:**
  - Halos are a larger percentage of accesses for 64-thread blocks
    - Accesses are in 32B lines, so in increments of 4 fp64 values
    - R = 1:
      - 64-thread block: reads 72 values to produce 64
      - 256-thread block: reads 264 values to produce 256
  - Easier to saturate arithmetic pipelines with more threads
  - Perf converges for larger orders:
    - Code becomes arithmetic rather than bandwidth bound

Stencils along "slow" dimensions

- **Each thread is responsible for a "pencil" of output**
  - "Marches" along the dimension
  - Keeps the necessary number of elements in registers
- **Per output element:**
  - Read one input element, do all the arithmetic
    - Arithmetic intensity increases with stencil size
    - Memory pressure doesn't
  - Manage values in registers ("advance" the queue)



```
template <int radius, int diameter>
__global__ void dy( TYPE* g_dy, const TYPE* g_input,
                    const int nx, const int ny, const int nz,
                    const int dimx, const int dimy, const int dimz )
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iz = blockIdx.y * blockDim.y + threadIdx.y;

    int stride = dimx;
    int idx_out = iz*dimx*dimy + ix;
    int idx_in  = idx_out - radius*stride;

    TYPE buffer[diameter];

    #pragma unroll
    for( int i=1; i<diameter; i++)
    {
        buffer[i] = g_input[idx_in];
        idx_in += stride;
    }

//  #pragma unroll X
    for( int iy=0; iy<ny; iy++)
    {
        #pragma unroll
        for( int i=0; i<diameter-1; i++)
            buffer[i] = buffer[i+1];
        buffer[diameter-1] = g_input[idx_in];

        TYPE derivative = c_coeff[0] * buffer[radius];
        #pragma unroll
        for( int i=1; i<=radius; i++)
            derivative += c_coeff[i] * ( buffer[radius-i] + buffer[radius+i] );

        g_dy[idx_out] = derivative

        idx_in  += stride;
        idx_out += stride;
    }
}
```

Compute indices for access

```
template <int radius, int diameter>
__global__ void dy( TYPE* g_dy, const TYPE* g_input,
                    const int nx, const int ny, const int nz,
                    const int dimx, const int dimy, const int dimz )
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iz = blockIdx.y * blockDim.y + threadIdx.y;

    int stride = dimx;
    int idx_out = iz*dimx*dimy + ix;
    int idx_in  = idx_out - radius*stride;

    TYPE buffer[diameter];

    #pragma unroll
    for( int i=1; i<diameter; i++)
    {
        buffer[i] = g_input[idx_in];
        idx_in += stride;
    }

//  #pragma unroll X
    for( int iy=0; iy<ny; iy++)
    {
        #pragma unroll
        for( int i=0; i<diameter-1; i++)
            buffer[i] = buffer[i+1];
        buffer[diameter-1] = g_input[idx_in];

        TYPE derivative = c_coeff[0] * buffer[radius];
        #pragma unroll
        for( int i=1; i<=radius; i++)
            derivative += c_coeff[i] * ( buffer[radius-i] + buffer[radius+i] );

        g_dy[idx_out] = derivative

        idx_in  += stride;
        idx_out += stride;
    }
}
```

Compute indices for access

Declare the local (register) buffer for values
Fill it up to start the computation

```
template <int radius, int diameter>
__global__ void dy( TYPE* g_dy, const TYPE* g_input,
                    const int nx, const int ny, const int nz,
                    const int dimx, const int dimy, const int dimz )
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iz = blockIdx.y * blockDim.y + threadIdx.y;

    int stride = dimx;
    int idx_out = iz*dimx*dimy + ix;
    int idx_in  = idx_out - radius*stride;

    TYPE buffer[diameter];

    #pragma unroll
    for( int i=1; i<diameter; i++)
    {
        buffer[i] = g_input[idx_in];
        idx_in += stride;
    }

    #pragma unroll 5
    for( int iy=0; iy<ny; iy++)
    {
        #pragma unroll
        for( int i=0; i<diameter-1; i++)
            buffer[i] = buffer[i+1];
        buffer[diameter-1] = g_input[idx_in];

        TYPE derivative = c_coeff[0] * buffer[radius];
        #pragma unroll
        for( int i=1; i<=radius; i++)
            derivative += c_coeff[i] * ( buffer[radius-i] + buffer[radius+i] );

        g_dy[idx_out] = derivative

        idx_in  += stride;
        idx_out += stride;
    }
}
```

Compute indices for access

Declare the local (register) buffer for values
Fill it up to start the computation

Main loop

```
#pragma unroll 5
for( int iy=0; iy<ny; iy++)
{
    #pragma unroll
    for( int i=0; i<diameter-1; i++)
        buffer[i] = buffer[i+1];
    buffer[diameter-1] = g_input[idx_in];

    TYPE derivative = c_coeff[0] * buffer[radius];
    #pragma unroll
    for( int i=1; i<=radius; i++)
        derivative += c_coeff[i] * ( buffer[radius-i] + buffer[radius+i] );

    g_dy[idx_out] = derivative;

    idx_in  += stride;
    idx_out += stride;
}
```

“Advance” the local values

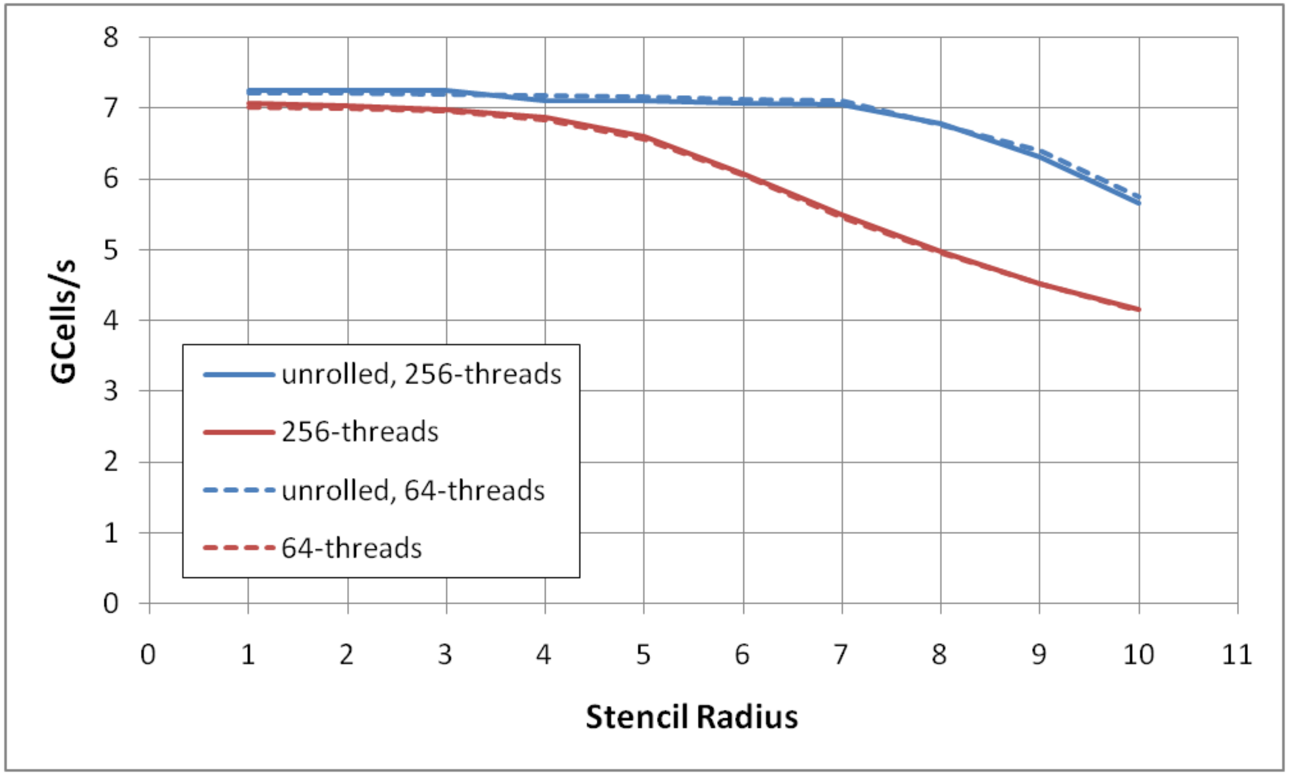
Compute the derivative

```
#pragma unroll 5
for( int iy=0; iy<ny; iy++)
{
    #pragma unroll
    for( int i=0; i<diameter-1; i++)
        buffer[i] = buffer[i+1];
    buffer[diameter-1] = g_input[idx_in];

    TYPE derivative = c_coeff[0] * buffer[radius];
    #pragma unroll
    for( int i=1; i<=radius; i++)
        derivative += c_coeff[i] * ( buffer[radius-i] + buffer[radius+i] );

    g_dy[idx_out] = derivative;

    idx_in  += stride;
    idx_out += stride;
}
```

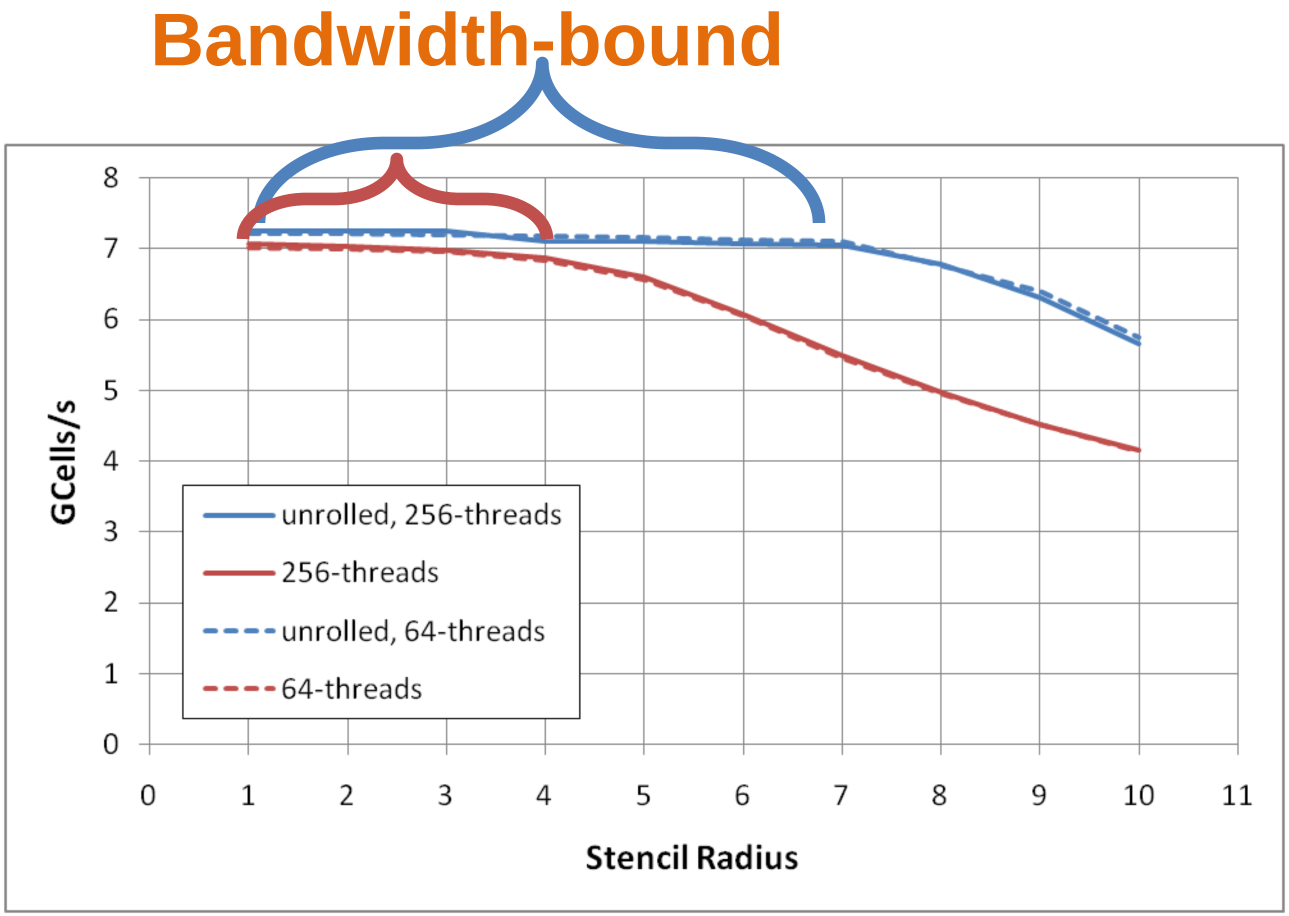
"Advance" the local values

Compute the derivative

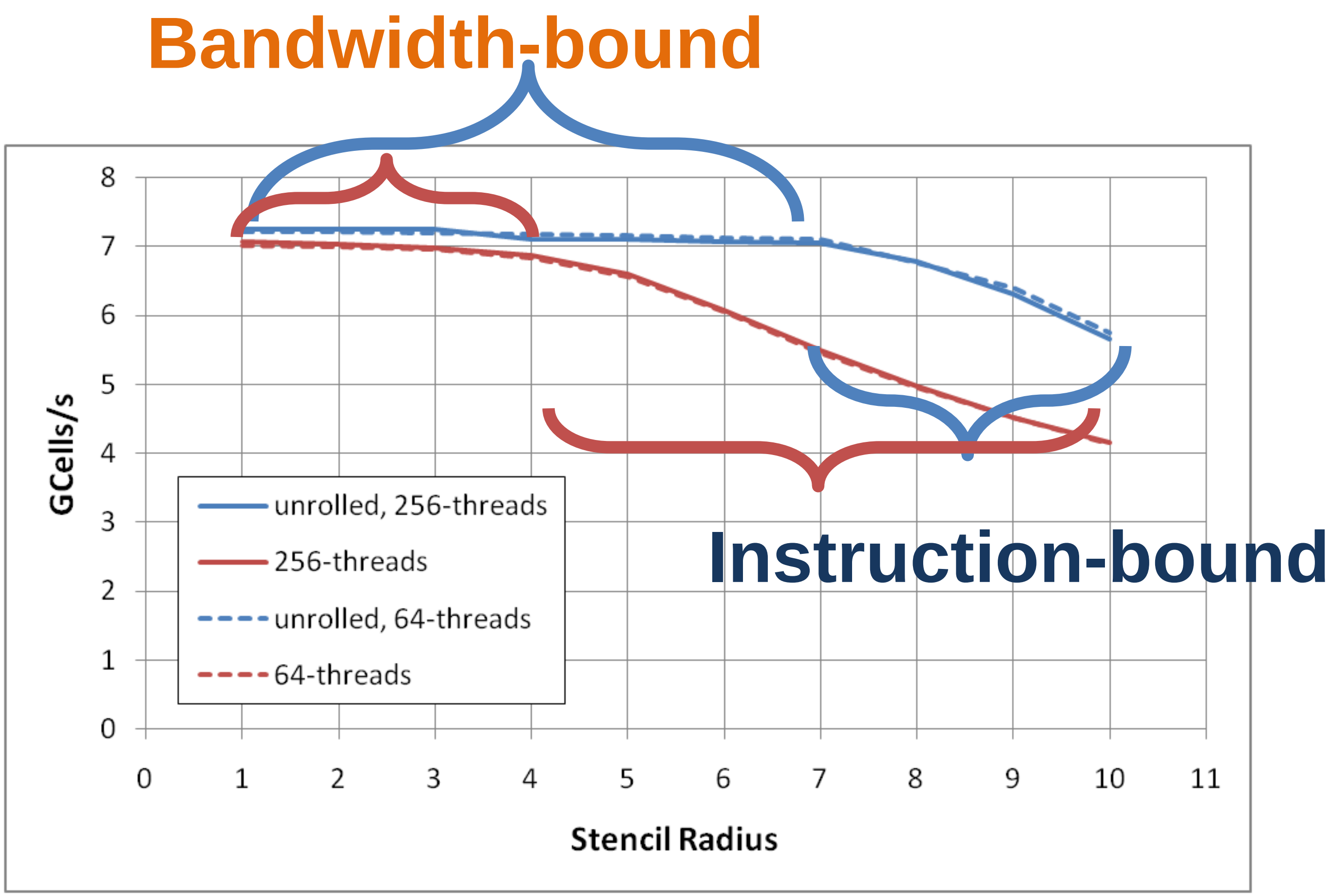
Y-stencil throughput



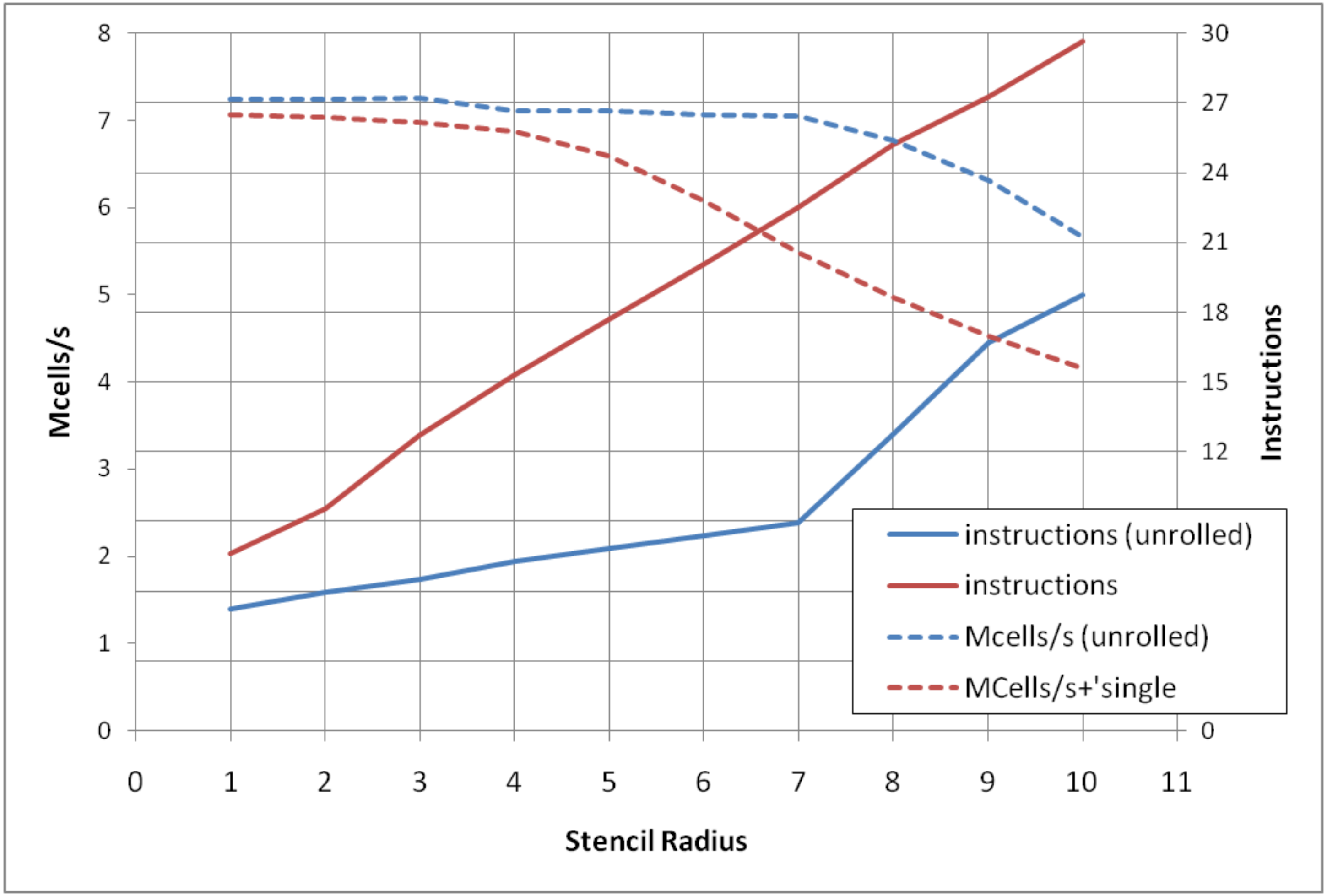
- Z-stencil is pretty much the same



Bandwidth-bound
GCells/s
Stencil Radius
unrolled, 256-threads
256-threads
unrolled, 64-threads
64-threads
0
1
2
3
4
5
6
7
8
9
10
11


Bandwidth-bound
Instruction-bound
GCells/s
Stencil Radius
0
1
2
3
4
5
6
7
8
9
10
11
unrolled, 256-threads
256-threads
unrolled, 64-threads
64-threads

Y-stencil performance vs instructions issued



Summary: PDEs with 1-dimensional derivatives

- **Derivatives along the fastest-dimension tend to be instruction-throughput limited**
  – Small threadblocks perform slower for low orders
- **Derivatives along the “slow” dimensions stay memory bandwidth limited until larger orders**
  – Perform essentially as memcopies

PDEs with derivatives in 2 dimensions

- **Two "subtypes"**
  - Combination of derivatives along one dimension

$$\left(\frac{\partial^2}{\partial^2 x}+\frac{\partial^2}{\partial^2 y}\right) \quad \left(\frac{\partial^2}{\partial^2 x}+\frac{\partial^2}{\partial^2 z}\right) \quad \left(\frac{\partial^2}{\partial^2 y}+\frac{\partial^2}{\partial^2 z}\right)$$

  - Mixed derivatives

$$\frac{\partial^2}{\partial x \partial y} \quad \frac{\partial^2}{\partial x \partial z} \quad \frac{\partial^2}{\partial y \partial z}$$

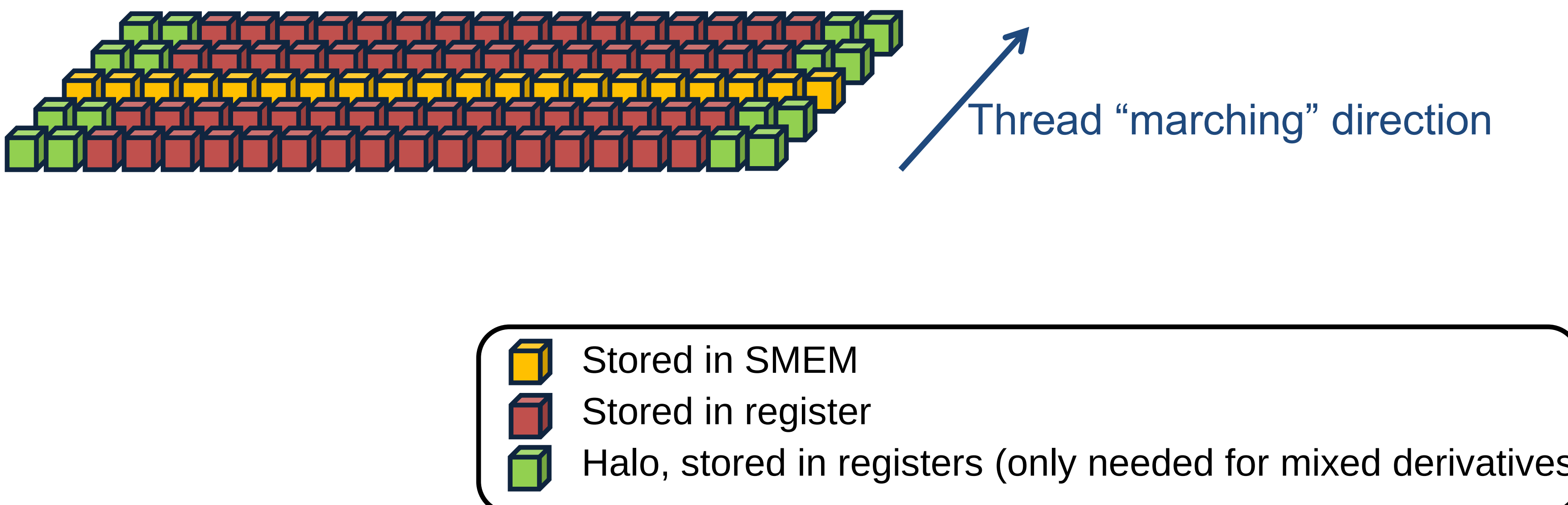
- **Implementation choices:**
  - Two-pass approach
    - 2 kernel launches, 2nd consumes the output of the 1st one
    - More accesses per output cell, but halos are a small percentage of accesses
  - Single-pass approach
    - Fewer accesses per output cell, but halos can start dominating

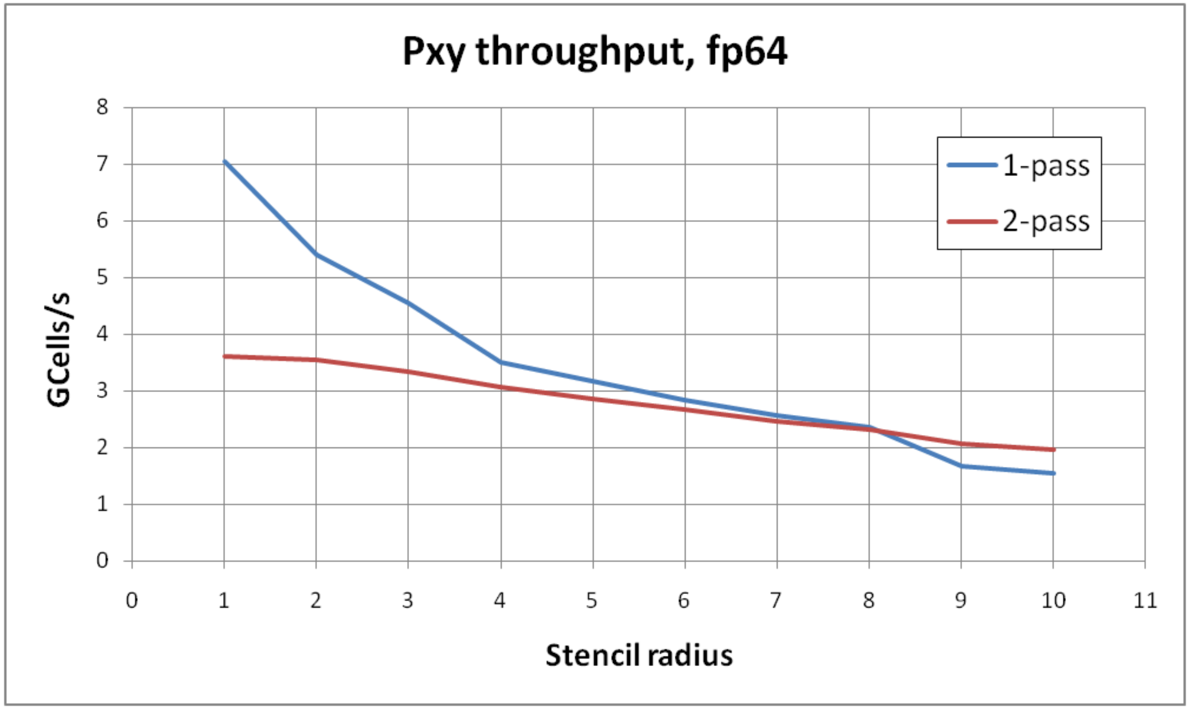
Two pass approach

- **Mixed derivatives:**
  - Straightforward: run 2 kernels in sequence
  - 4 accesses per output cell
- **Combination of "single" derivatives:**
  - 2nd kernel needs a to read both the original data and the output of the 1st kernel
  - 5 accesses per output cell

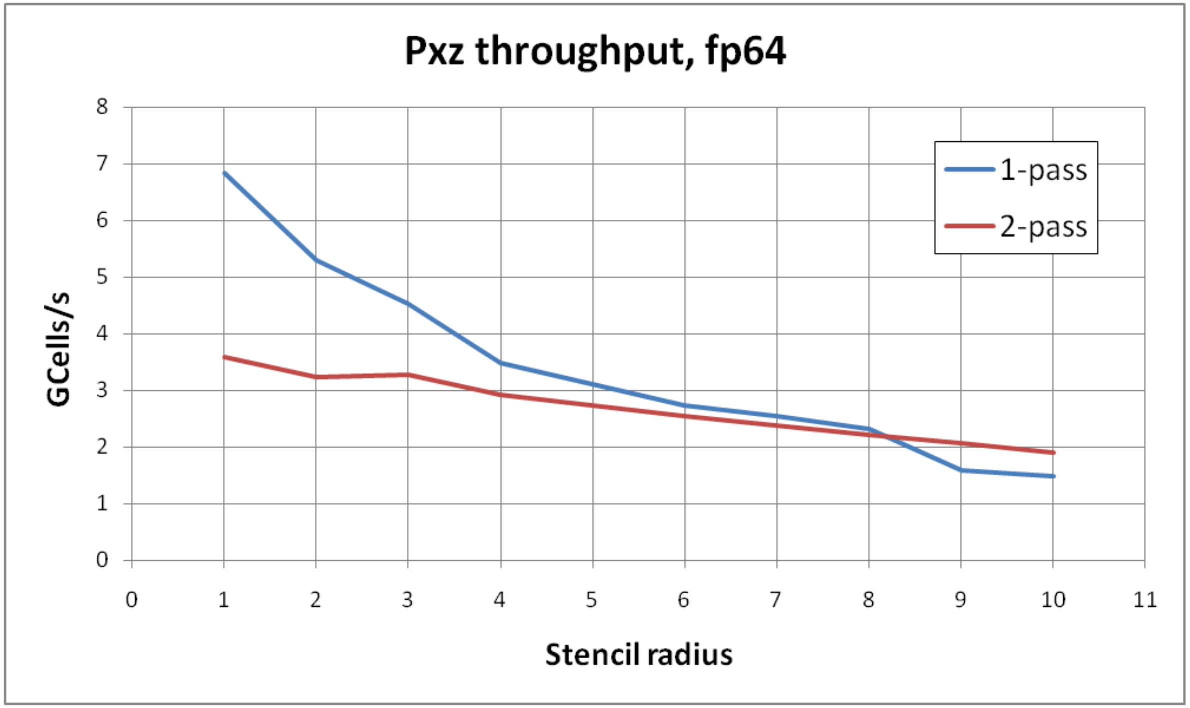
Single-pass approach

- **Derivatives including the fastest-varying dimension**
  - Compute the derivative in the "slow" dimension out of registers, store into SMEM
  - Compute the derivative in the "fast" dimension out of SMEM





Pxy throughput, fp64
GCells/s
Stencil radius
1-pass
2-pass
0
1
2
3
4
5
6
7
8
9
10
11

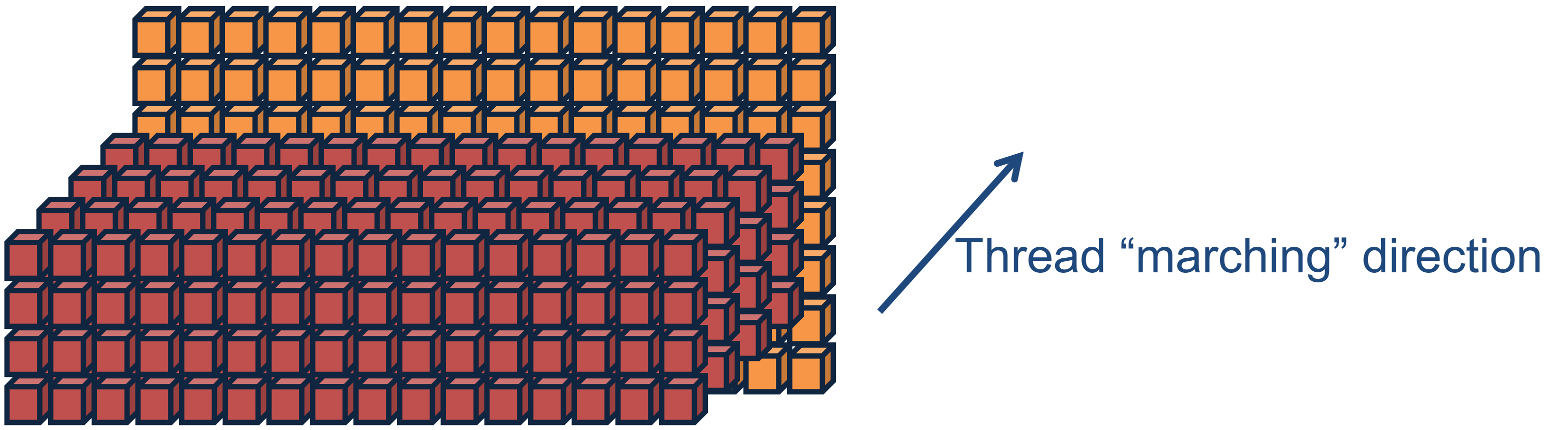


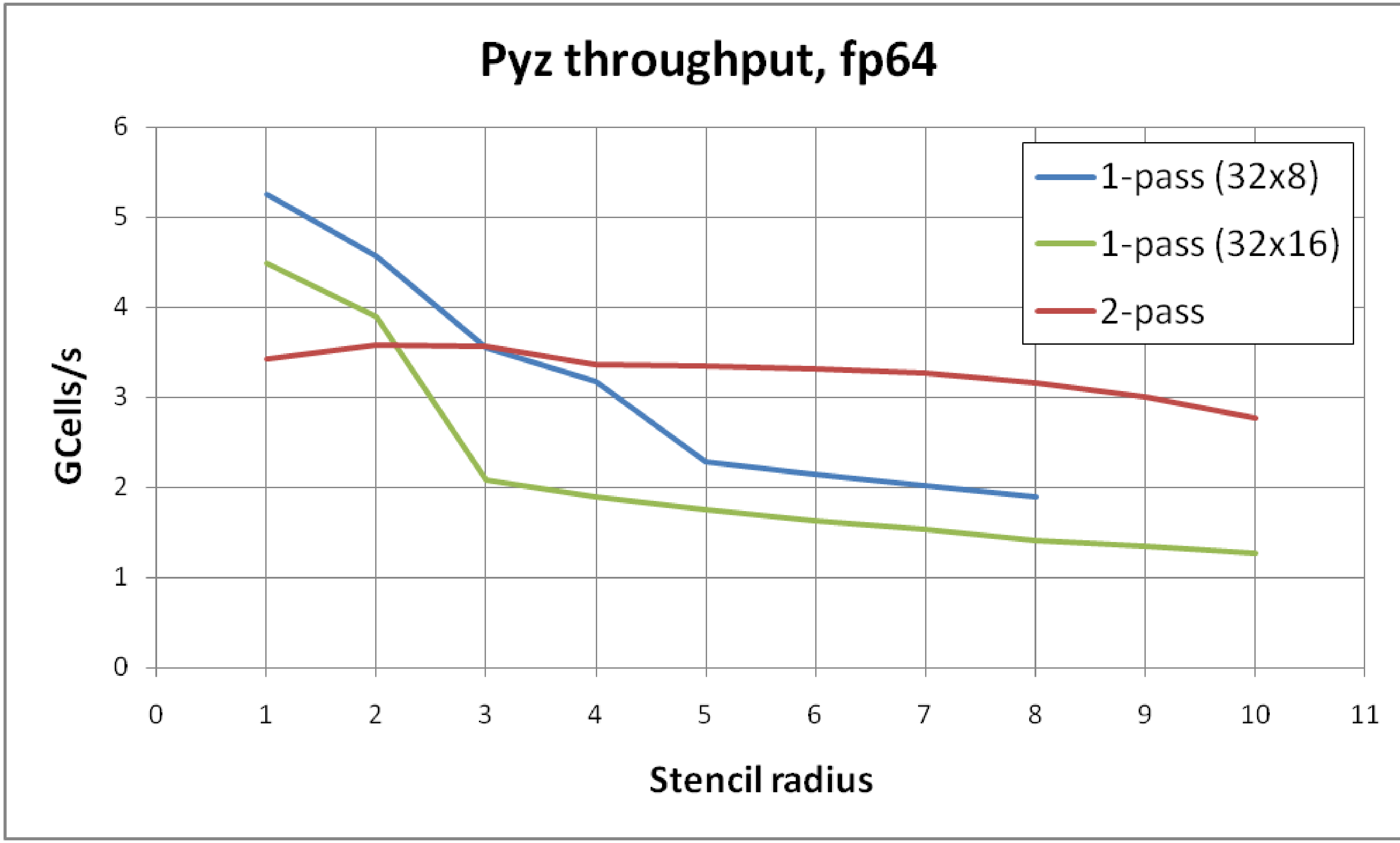
Pxz throughput, fp64
GCells/s
Stencil radius
1-pass
2-pass
0
1
2
3
4
5
6
7
8
9
10
11

Single-pass approach

$$\frac{\partial^2}{\partial y \partial z}$$

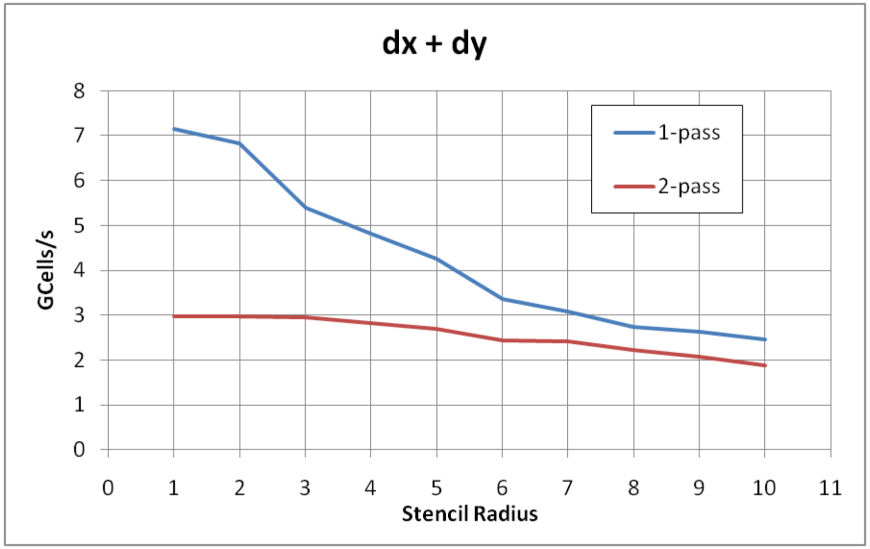
- **Mixed Derivatives not including the fastest-varying dimension**
  - Successive threads still need to access along the fastest-varying dimension
    - To get GMEM coalescing
  - Use 2D threadblocks
    - Tile the xy-plane with threadblocks
    - Each threadblock "marches" along z dimension
    - Load data and halos above/below at the front into **SMEM**, compute y-deriv
    - Propagate y-derivs through **registers**, compute z deriv

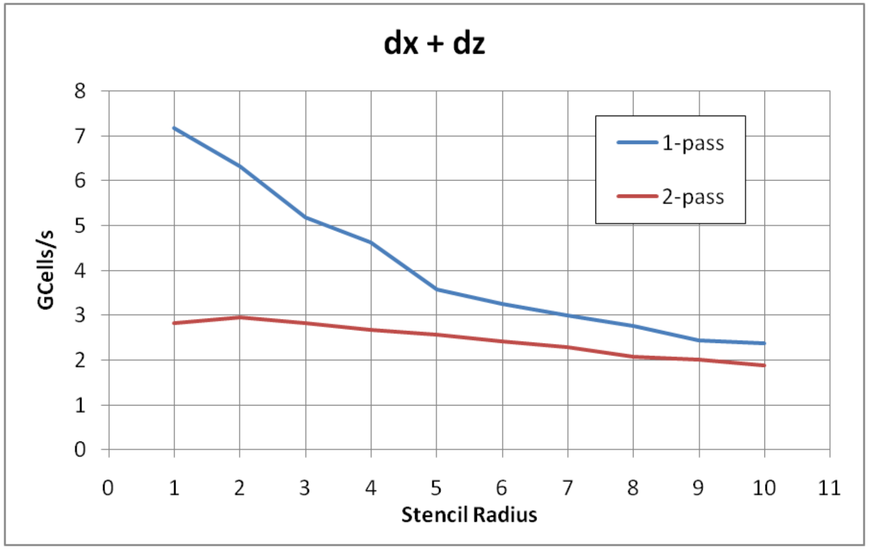


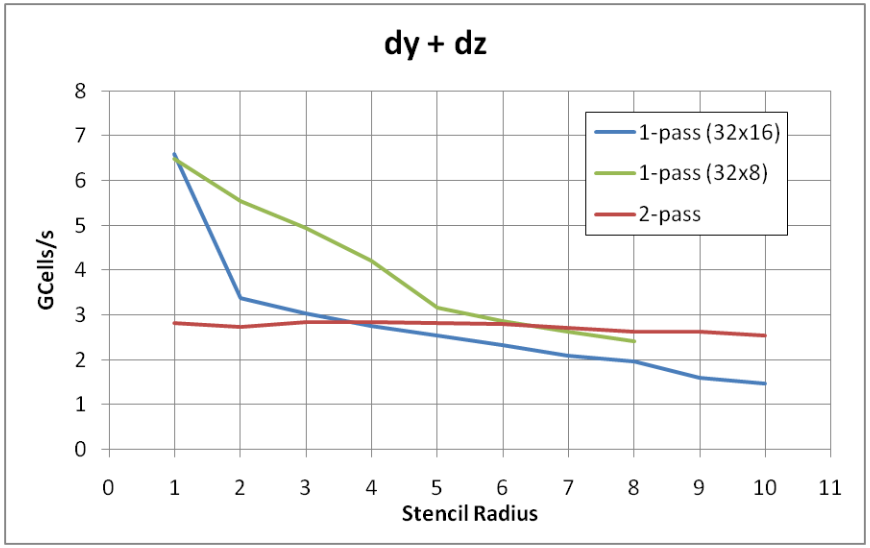


Pyz throughput, fp64
GCells/s
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10
11
Stencil radius
1-pass (32x8)
1-pass (32x16)
2-pass

Combinations of “single” derivatives

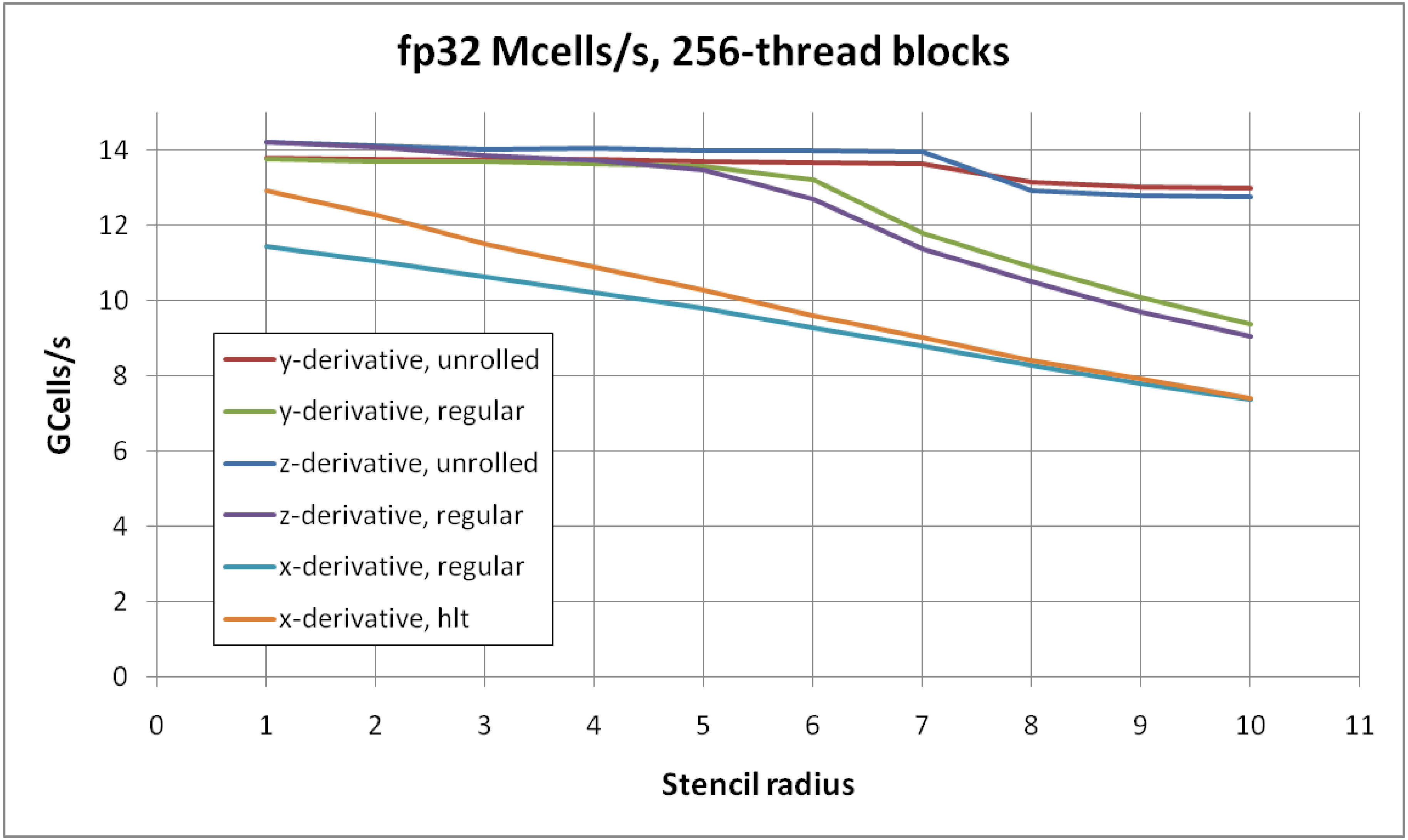






Comments and conclusions

- **Understanding basic computer-architecture concepts allows for very effective optimizations**
  - Know whether code is memory or instruction bound, optimize accordingly
    - loop-unrolling pragma for {y, z}-stencils
    - Choosing 1- or 2-pass approach for yz-stencils
  - Keep mem system in mind when parallelizing
- **Output throughput does not decrease by much when increasing spatial order from 2nd to 4th or 6th**
  - May allow working with smaller grids / longer time-steps
- **Fp64 stencil code is bandwidth-bound for smaller orders, instruction-bound for larger ones**
  - Cross-over: 8th to 14th order in space
  - Fp32 stencils are bandwidth bound for even greater orders



fp32 Mcells/s, 256-thread blocks
GCells/s
Stencil radius
0
2
4
6
8
10
12
14
0
1
2
3
4
5
6
7
8
9
10
11
y-derivative, unrolled
y-derivative, regular
z-derivative, unrolled
z-derivative, regular
x-derivative, regular
x-derivative, hlt