# Accelerating Finite Difference Computations Using General Purpose GPU Computing

**Date: 7 November 2012**

**POC: James D. Stevens**
**559th SMXS/MXDECC**
**Phone: 405-736-4051**
**Email: james.stevens@tinker.af.mil**

## Background

- **Presenter: Jim Stevens**
  - **MS in Computer Science, 2011, School of Engineering and Applied Science at Washington University in St. Louis**
  - **76th Software Maintenance Group at Tinker AFB**
- **Mission: maintain software on weapon systems**
  - **Support the warfighter!**
- **Supercomputing resources 660core/3.5 TFLOPS**
  - **Recycled computer systems ($75 initial budget)**
- **GPGPU programming with CUDA**

# Outline

- Goal: accelerate weather prediction model
  - History of optimization efforts
- A look at the GPU
  - Can it help reduce real time computing requirements?
- GPU programming approach
  - Basic overview
- Our case study and results
  - Techniques for optimization
  - Results/evaluation
    - Weather calculations
    - EM wave calculations
- Road map for future work

# Weather Model

- **U,V,W represent winds**
- **Theta $\theta$ represents temperature**
- **$\pi$ represents pressure**
- **T – Time**
- **X – east west direction**
- **Y – north south direction**
- **Z – vertical direction**
- **Turb – turbulence terms (what can't be measured/predicted)**
- **S – Source terms, condensation, evaporation, heating, cooling**
- **D – numerical smoothing**
- **f – Coriolis force (earth's rotation)**

## Navier Stokes Equations

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -c_p\theta\frac{\partial \pi}{\partial x} + fv - f'w + D_u + turb_u$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\overset{\theta}{c}_p\theta\frac{\partial \pi}{\partial y} - fu + D_v + turb_v$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -c_p\theta\frac{\partial \pi'}{\partial z} + g\frac{\theta'}{\overline{\theta}} + f'u + D_w + turb_w$$

$$\frac{\partial \theta}{\partial t} + u\frac{\partial \theta}{\partial x} + v\frac{\partial \theta}{\partial y} + w\frac{\partial \theta}{\partial z} = D_\theta + turb_\theta + S_\theta$$

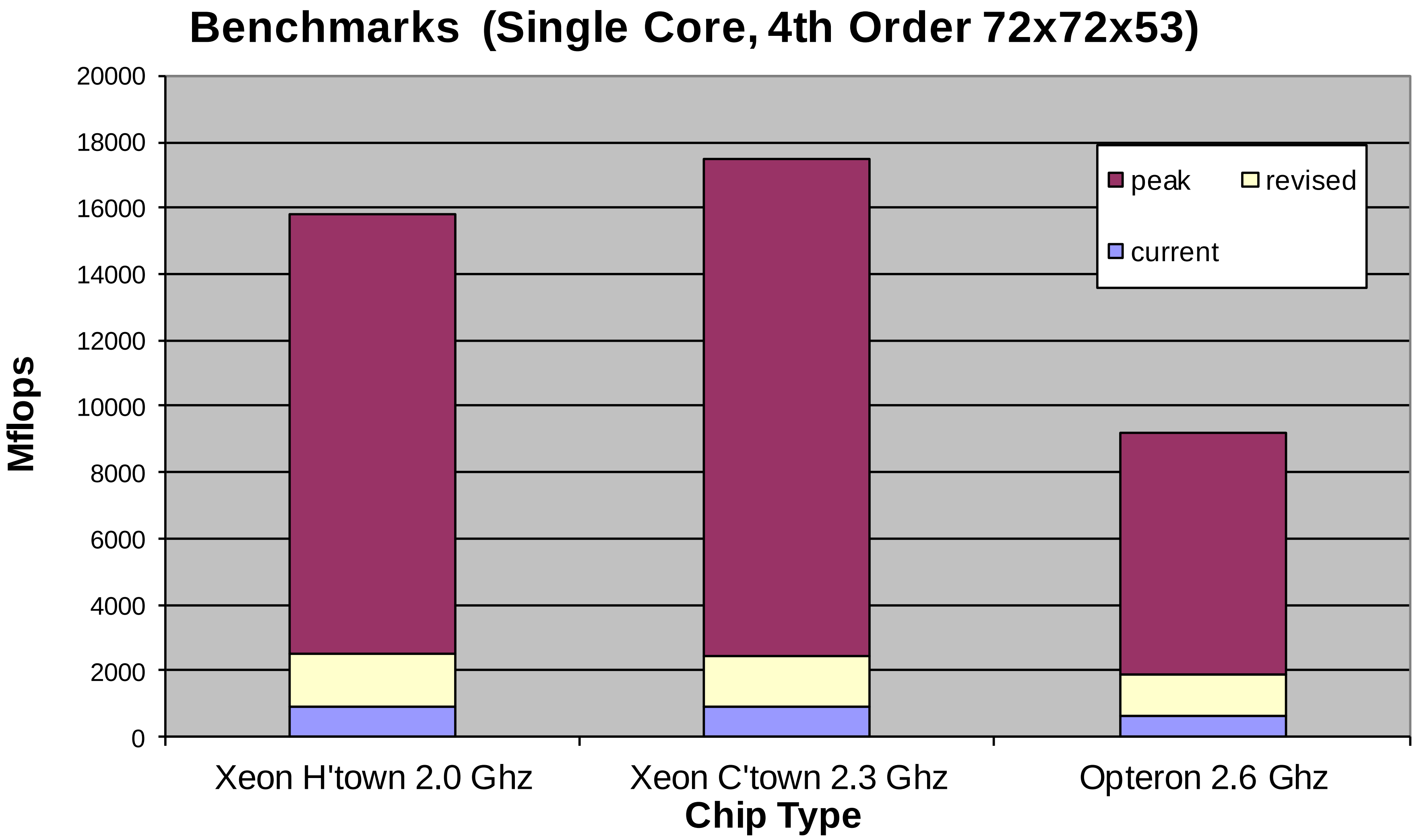
$$\frac{\partial \pi}{\partial t} + u\frac{\partial \pi}{\partial x} + v\frac{\partial \pi}{\partial y} + w\frac{\partial \pi}{\partial z} = -\frac{R_d}{c_v}\pi\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right) + \frac{R_d}{c_v}\frac{\pi}{\theta}\frac{d\theta}{dt}$$

Others variables include soil, cloud and precipitation processes

## Past Optimization Attempts

- **Vector processors: 50-90% peak – fast memory**
  - **Loop Merging – removing operators from the code**
  - **Loop Fusion – helps the compiler vectorize code**
- **Scalar: 10-60% peak – memory bound**
  - **Loop Merging – reduces number of loads and stores**
  - **Supernoding/Tiling**
    - **Data/cache reuse**
      - **Rearrange computations for maximum data reuse**
- **References:**
  - **OSCER Symposium (Weber: 2005,2006,2008)**
  - **Linux Cluster Institute (Weber and Neeman: 2006)**

# Past CPU Results

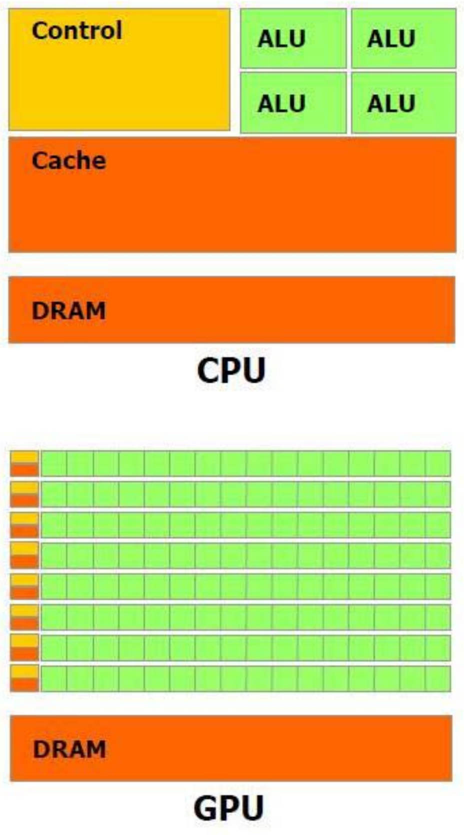


## The GPU

(graphics processing unit)

# NVIDIA Tesla C1060 GPU

- Tesla C1060 GPU has 240 cores
- 30 Multiprocessors (MP)
  - 8 cores each
- Registers on each MP
  - Accessible to all 8 cores
- Goal: Utilize all GPU cores
  - >80% core utilization on loops



## CPU - GPU Comparison

- Single core CPUs capable of ~10 GFLOP/s
- Multicore capable of ~100's GFLOP/s
- But CPU memory bandwidth severely restricts real world performance of multicore CPUs for memory intensive applications
- GPUs offer > 1 TFLOP/s potential
- The coding style for GPGPUs is very different
- "New language" (CUDA / OpenCL) needed for programming on GPU
- CUDA = Compute Unified Device Architecture

## Learning CUDA

- **“Manual” GPGPU programming is hard (at first)**
  - **“Automatic” GPGPU programming – add directives surrounding loops**
    - **Compiler attempts to parallelize the loop**
    - **Rarely yields best results**
  - **“Manual” GPGPU programming – write GPU code yourself**
    - **More difficult**
    - **Faster code, more customizable**
    - **Disclaimer: this is not a CUDA class**
      - **Goal: understand basic CUDA/GPGPU programming concepts well enough to decide whether or not it has the potential accelerate your code**

# CUDA Threads

- **Thread = a single instance of computation**
  - **One thread per processor-core at a time**
- **CUDA allows you to specify the thread organization, count, and indexing**
  - **You control which threads are responsible for which portion of the task**

## GPU Memory Components

- Global Memory
  - Main memory, 4 GB for the NVIDIA Tesla C1060
  - About 200 cycles to access (vs. 50 cycles for a CPU)
- Registers
  - 64KB per multiprocessor (vs. 512 B for Pentium 4 CPU)
  - 1 cycle to access
- Shared registers (AKA “shared memory”)
  - 16 KB per multiprocessor
  - Can be allocated for each block of threads
  - All threads within block can access all data in shared registers, even if another thread fetched it
  - Allows for data reuse – this is important

## General CUDA Program Format

- Step 1 – copy data from CPU main memory to GPU global memory (from host to device)
- Step 2 – threads run code inside kernel function
  - Each thread fetches some data from global memory and stores it in registers
  - Each thread performs computations
  - Each thread stores a result in global memory
- Step 3 – copy results from device back to host

## Simple CUDA example

- We want to increment each element in a 1-dimensional array of integers

- CPU Approach

1. Create/initialize array
2. Perform loop

```
do i = 1,n
    array(i) = array(i)+1
end do
```

- GPU Approach

1. Create/initialize array
2. Copy array data to GPU memory
3. Create n threads
4. Have each thread do the following:

```
array[threadIDX] =
                 array[threadIDX] + 1
```

5. Copy array back to host

- threadIDX is the thread's unique thread index
- Threads may execute in any order

## Simple CUDA Example

**Any questions at this point?**

# Weather Model Equations

- U,V,W represent winds
- Theta $\theta$ represents temperature
- $\pi$ represents pressure
- T – Time
- X – east west direction
- Y – north south direction
- Z – vertical direction
- Turb – turbulence terms (what can't be measured/predicted)
- S – Source terms, condensation, evaporation, heating, cooling
- D – numerical smoothing
- f – Coriolis force (earth's rotation)

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -c_p\theta\frac{\partial \pi}{\partial x} + fv - f'w + D_u + turb_u$$

$\theta$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -c_p\theta\frac{\partial \pi}{\partial y} - fu + D_v + turb_v$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -c_p\theta\frac{\partial \pi'}{\partial z} + g\frac{\theta'}{\overline{\theta}} + f'u + D_w + turb_w$$

$$\frac{\partial \theta}{\partial t} + u\frac{\partial \theta}{\partial x} + v\frac{\partial \theta}{\partial y} + w\frac{\partial \theta}{\partial z} = D_\theta + turb_\theta + S_\theta$$

$$\frac{\partial \pi}{\partial t} + u\frac{\partial \pi}{\partial x} + v\frac{\partial \pi}{\partial y} + w\frac{\partial \pi}{\partial z} = -\frac{R_d}{c_v}\pi\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right) + \frac{R_d}{c_v}\frac{\pi}{\theta}\frac{d\theta}{dt}$$

## Solving Weather Model Equations

- **CPU Version**

```
DO k = 3,nz-2
  DO j = 3,ny-2
    DO i = 3,nx-2

      u(i,j,k,2)= -u(i,j,k,2)*...          ( 150 operations )
      !  compute uadv u                  (... 18 operations ...)
      !  compute vadv u                  (... 16 operations ...)
      !  compute wadv u                  (... 16 operations ...)
      !  compute cmixx u                 (... 33 operations ...)
      !  compute cmixy u                 (... 33 operations ...)
      !  compute cmixz u                 (... 33 operations ...)

      v(i,j,k,2)= -v(i,j,k,2)*...          ( 148 operations )
      w(i,j,k,2)= -w(i,j,k,2)*...          ( 100 operations )
      p(i,j,k,2)= -p(i,j,k,2)*...          ( 49 operations  )
      pt(i,j,k,2)= -pt(i,j,k,2)*...        ( 148 operations )
```

- **Normally, these computations are done separately, why combine them?**
  - **Data reuse!**

**595 operations total**

## Stencil Data Requirements

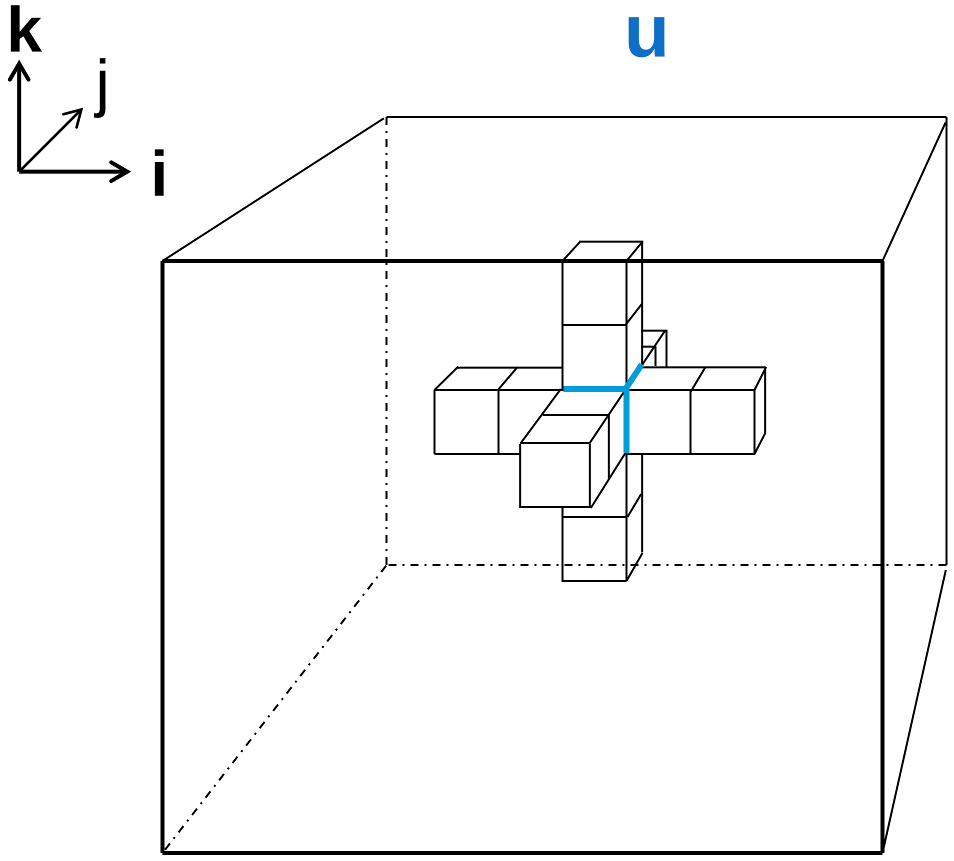
- **Subset of calculation – u array – uadv u**

```
u(i,j,k,2)= -u(i,j,k,2)*rk_constant1(n)

!  compute uadv u

    +tema4*( (u(i,j,k,1)+u(i+2,j,k,1))
            *(u(i+2,j,k,1)-u(i,j,k,1))
            +(u(i,j,k,1)+u(i-2,j,k,1))
            *(u(i,j,k,1)-u(i-2,j,k,1)) )
    -temb4*( (u(i+1,j,k,1)+u(i,j,k,1))
            *(u(i+1,j,k,1)-u(i,j,k,1))
            +(u(i,j,k,1)+u(i-1,j,k,1))
            *(u(i,j,k,1)-u(i-1,j,k,1)) )
```

- **Note: For every (i,j,k) element, this part of the update requires the value at (i,j,k), as well as 4 other values – the two on either side of (i,j,k) in the i direction: (i-2,j,k) (i-1,j,k) (i+1,j,k) (i+2,j,k)**

# U Calculation – elements needed

| | Arrays | Direction of adjacent values |
|---|---|---|
| uadv u | u | i |
| vadv u | u, v | j |
| wadv u | u, w | k |
| cmixx u | u, ubar | i |
| cmixy u | u, ubar | j |
| cmixz u | u, ubar | k |



**ALL** elements needed to update u(i,j,k)

## Global Memory Access
## (i-calculations only)

- Normal registers - each thread will fetch five elements from global memory
  - That's inefficient - each element would be fetched 5 times by 5 different threads

- Shared registers - Each thread copies one element into a "shared" array (stored in shared registers) that can be accessed by all threads
  - Shared arrays allocated/accessed within block
- Then each thread only performs one global fetch and can access all 5 elements it needs!

## Shared Memory Limitation

- Limited to 16 KB of shared registers
  - We're processing gigabytes of data
  - Need to break up the problem into smaller pieces that can be moved in and out of shared memory efficiently

- What else do we need to do to get maximum performance?

## Strategies for Performance

- Make sure global memory fetches are coalescing
  - When adjacent threads access adjacent locations in global memory, the fetches are “coalesced” automatically into a single large fetch
  - Absolutely necessary for good performance
  - Number 1 priority

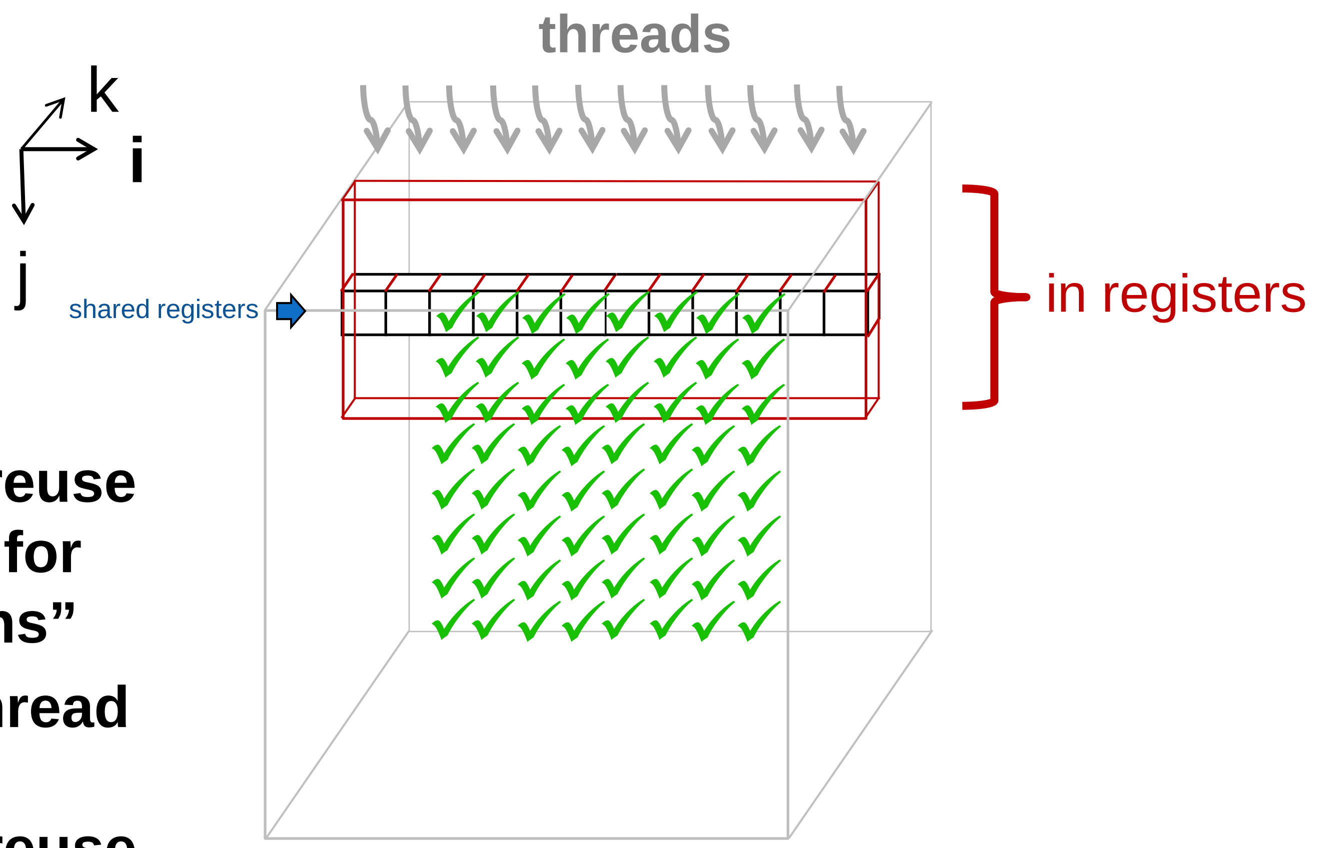
## Strategies for Performance

- Reuse as much data as possible
  - By using shared registers
    - Break problem into pieces that are small enough to fit into shared memory
  - By having threads perform cleverly designed loops
    - Not using shared registers
    - Loops within threads that solve the following problem…

## Data Reuse Through Looping

- To maximize coalescence, we need blocks of threads that are “long” in the i-direction.
- However, because of our size limitation on shared memory, this forces blocks to be “narrow” in the other two dimensions.
  - 64x1x1, for example
- This is a problem for the parts of the calculation that require neighboring data in the j and k directions
  - Can we still reuse data for these parts of the calculation?
  - Yes! Partially. We can combine the i and j calculations

## Data Reuse: Looping + Shared Registers



- Use shared registers to reuse data needed for “i-calculations”
- Have each thread loop in the j-direction to reuse data needed for the “j-calculations”

- Each element is only fetched once from global memory, and then used nine times (see animation)

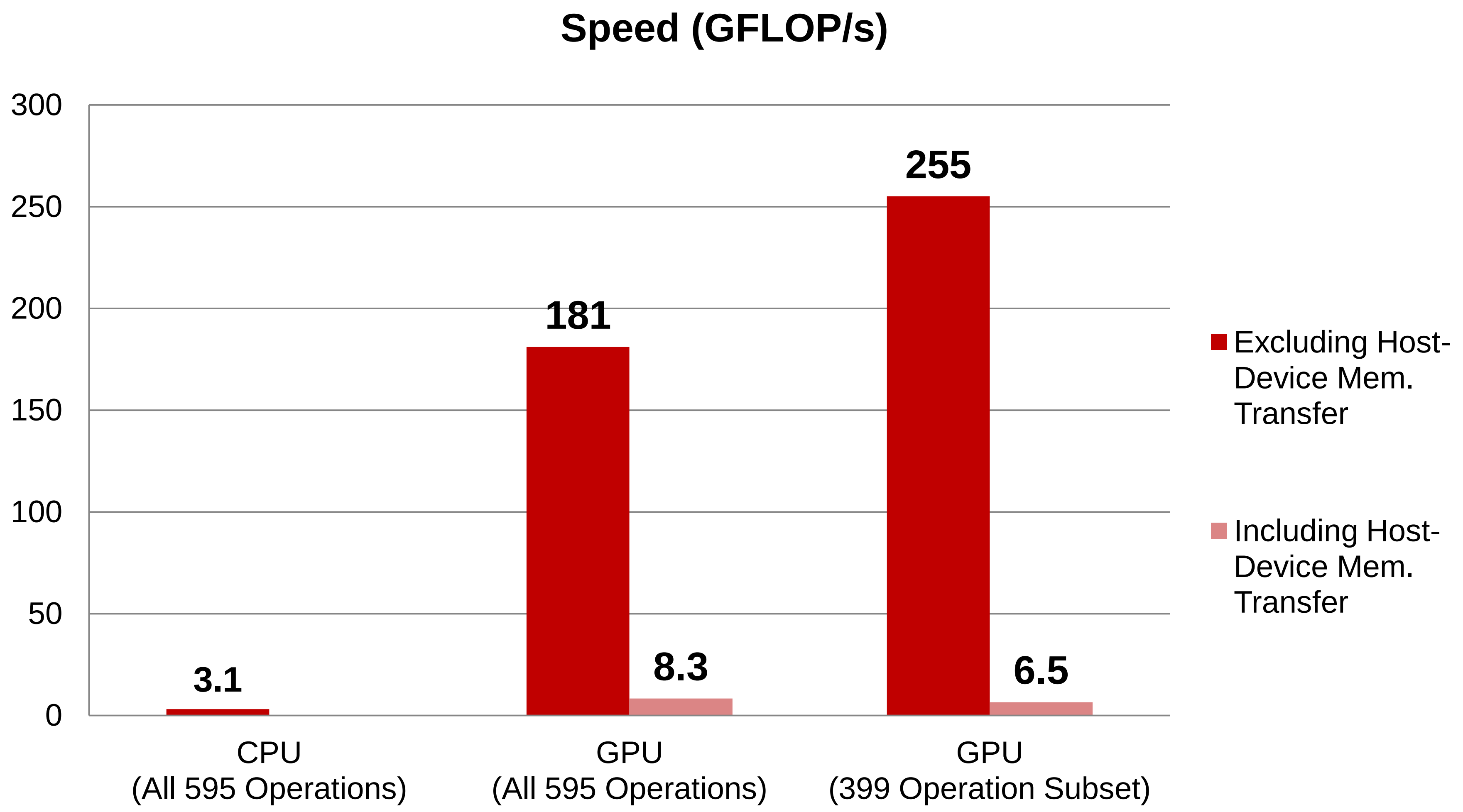
## Other Strategies for Performance

- "Hide" memory fetches with computations
  - Structure kernel so that data is being fetched while computations are being performed (the scheduler will try to help with this)
- Choose block dimensions that allow for maximum thread-scheduling efficiency
  - Multiples of 32 threads
  - Blocks that are "longer" in one dimension (i) to facilitates maximum coalescence

## Strategies for Performance

- Designing your program so that it uses all of these strategies is difficult
  - It's a bit like trying to design a car that is luxurious, safe, fast, agile, reliable, practical, inexpensive, visually appealing, and fuel efficient all at the same time
    - There are tradeoffs - you have to find the right balance
    - Experiment

# Weather Computation - Results



# Host-Device Data Transfer

- Huge bottleneck
- What can we do?
  - Hide data transfer behind CPU computations
    - Transfer data while CPU is performing other necessary work
  - Hide data transfer behind GPU computations
    - Transfer a piece of the data to the GPU
    - Begin performing GPU computations while the next piece of data is being transferred
- Currently working on this

## Evaluating Results

- How do we evaluate our results?
- Estimate theoretical hardware peak
  - 933 GFLOP/s for single precision
  - But we can't use some of the hardware
    - No texturing, reduces peak by about 33%
  - This number assumes we're taking advantage of the fused multiply-add instruction, but our computation doesn't have many multiply-adds
    - Reduces peak by about 50%
  - So achievable hardware peak is about 311 GFLOP/s
  - Kernel runs at 82% of peak, not bad!!!

## Estimating Application Speed Limit

- Determine theoretical application "speed limit"
- Based on global memory bandwidth and algorithm memory requirements
  - Even if our algorithm has 100% data reuse and we *completely hide* all operations behind data fetches, we would still need to fetch each element of data from global memory one time, and write our results back
  - Compute time required to move data

    T = (data moved) / (global memory bandwidth)
  - Compute speed limit (FLOP/s)

    ASL = (Algorithm FLOP Count) / T

## Application Speed Limit

- 786 GFLOP/s for the 399 operation subset (Tesla C1060 GPU)
  - Because this computation has such a high operation-to-memory-fetch ratio (OMFR), ~30:1, this "speed limit" is high
  - This is higher than our achievable hardware peak, which means our performance might increase if the GPU had faster multiprocessors
  - Suggests that our program is *not* memory bound
- This peak can be calculated before writing any code to find out if a particular computation is a good candidate for GPU acceleration
  - Increment array example: 12.8 GFLOP/s = poor candidate

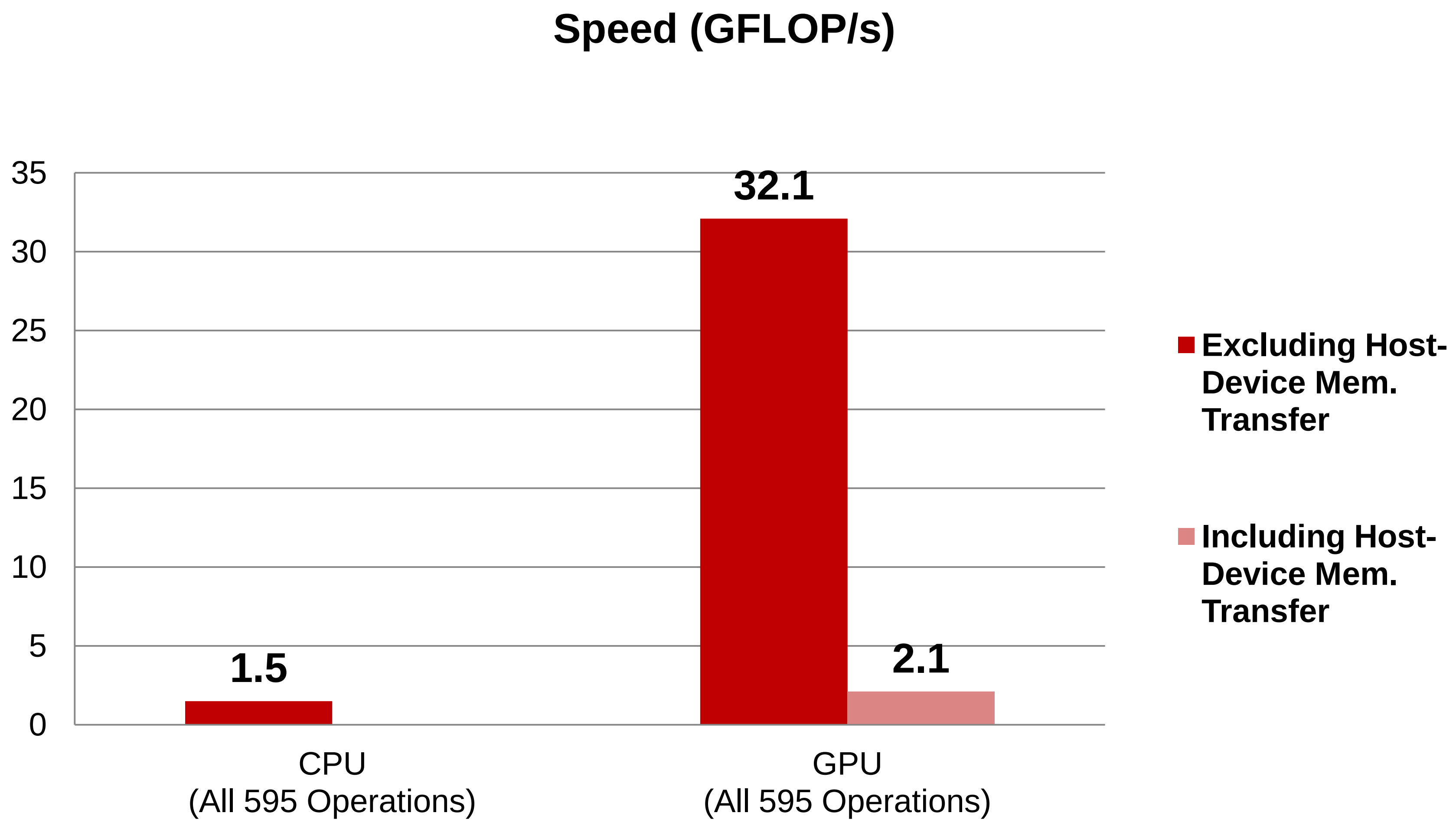
# Maxwell's Equations

$$\sigma\bar{\mathbf{E}} + \varepsilon\frac{\Delta\bar{\mathbf{E}}}{\Delta t} = \left[\left(\frac{\Delta H_z}{\Delta y} - \frac{\Delta H_y}{\Delta z}\right)\hat{\boldsymbol{\imath}} + \left(\frac{\Delta H_x}{\Delta z} - \frac{\Delta H_z}{\Delta x}\right)\hat{\boldsymbol{\jmath}} + \left(\frac{\Delta H_y}{\Delta x} - \frac{\Delta H_x}{\Delta y}\right)\hat{\boldsymbol{k}}\right]$$

$$\mu\frac{\Delta\bar{\mathbf{H}}}{\Delta t} = \left[\left(\frac{\Delta E_y}{\Delta z} - \frac{\Delta E_z}{\Delta y}\right)\hat{\boldsymbol{\imath}} + \left(\frac{\Delta E_z}{\Delta x} - \frac{\Delta E_x}{\Delta z}\right)\hat{\boldsymbol{\jmath}} + \left(\frac{\Delta E_x}{\Delta y} - \frac{\Delta E_y}{\Delta x}\right)\hat{\boldsymbol{k}}\right]$$

```
DO k = 3, nz-2
  DO j = 3, ny-2
    DO i = 3, nx-2
      hx(i,j,k) = da * hx(i,j,k) + db  * ( (ez(i,j,k)       - ez(i,j+1,k) ) * deny
                                         + (ey(i,j,k+1)  - ey(i,j,k)      ) * denz )
      hy(i,j,k) = da * hy(i,j,k) + db  * ( (ez(i+1,j,k)   - ez(i,j,k)     ) * denx
                                        + (ex(i,j,k)       - ex(i,j,k+1) ) * denz )
      hz(i,j,k) = da * hz(i,j,k) + db  * ( (ey(i,j,k)       - ey(i+1,j,k) ) * denx
                                        + (ex(i,j+1,k) - ex(i,j,k)  ) * deny )

…
!! Three loops for ex, ey, ez
```

## Maxwell’s Equations - Results



## Evaluating Results

- Observed top speed – 32.1 GFLOP/s
- Achievable hardware peak – 311 GFLOP/s (unchanged)
- OMFR (operation-to-memory-fetch-ratio) – 2.67:1
  - vs. 30:1 for the weather calculations
- ASL (application speed limit) – 68.6 GFLOP/s
- The ASL is *less than* the achievable hardware peak
- Achieved 47% of the ASL
- The OMFR (and thus, the ASL) for this calculation is 11.2 times smaller than the OMFR for the weather calculations, and it runs 8 times slower
- This computation may be a reasonable candidate for GPU acceleration, but the speedup will be much greater for the weather calculations (due to their higher OMFR)

# Good Candidates for GPU Acceleration

- Easily parallelizable
  - Same set of independent operations are performed on each element in a domain (SIMD)
  - These operations can execute in any order
- Spatial locality
  - Individual operations require data that is nearby in the domain
  - Facilitates data reuse
- High operation-to-memory-fetch ratio
  - Calculate theoretical “speed limit” based on algorithm memory requirements and global memory bandwidth

## Potential Future Work

- Deal with the host-device memory transfer bottleneck
- Add other big time-step computations for weather computation
  - Turbulence, coriolis, buoyancy
  - Cloud physics
  - Radiation
- Include small time-step
  - Texture/interpolators for the pressure gradient
- Parallel version (MPI)

## Resources for Learning CUDA

- ***Programming Massively Parallel Processors: A Hands-On Approach*** **by Kirk and Hwu**
  - **On Books 24x7**
- **Online lecture slides and audio**
  - **ECE 498 AL (Univ. of Illinois)**
- **NVIDIA CUDA Programming Guide**
- **Portland Group CUDA Fortran Programming Guide and Reference**
- **Forums**
  - **Portland group**
  - **NVIDIA**

U.S. AIR FORCE