Introduction to GPU Computing

Jeff Larkin
Cray Supercomputing Center of Excellence
larkin@cray.com

Goals for this tutorial

- Understand the architectural differences between GPUs and CPUs and the associated trade-offs
- Recognize several GPU programming models and how/when to use each
- Understand how to analyze GPU performance
- Recognize very basic GPU optimizations

This tutorial is not…

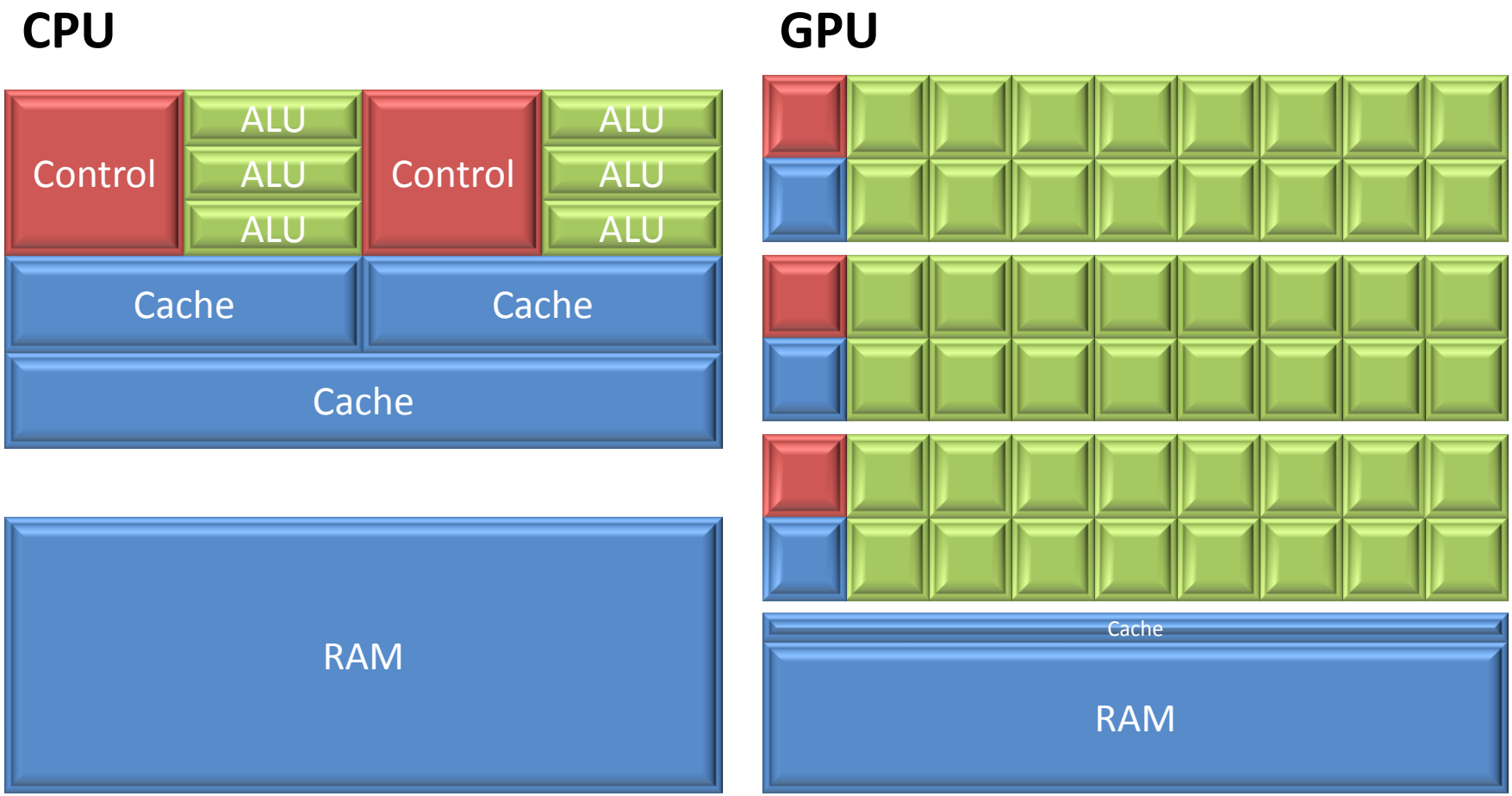
- A deep-dive on GPU programming
- The be all and end all on GPU optimization
- A recipe for getting 10, 100, 1000X speed-ups for your application

GPU ARCHITECTURE BASICS

Section Goals

- Recognize the differences between CPU/GPU architectures
- Identify when one architecture may be better suited than the other.

CPU/GPU Architectures



CPU/GPU Architectures

CPU

- Large memory, directly accessible
- Each core has own, independent control logic
  - Allows independent execution
- Coherent caches between cores
  - Can share & synchronize

GPU

- Relatively small memory, must be managed by CPU
- Groups of compute cores share control logic
  - Saves space, power, …
- Shared cache & synchronization within groups
  - None between groups

Play to your strengths

CPU

- Tuned for serial execution with short vectors
- Multiple independent threads of execution
- Branch-prediction
- Memory latency hidden by cache & prefetching
  - Requires regular data access patterns

GPU

- Tuned for highly parallel execution
- Threads work in lockstep within groups
  - Much like vectors
- Serializes branchy code
- Memory latency hidden by swapping away stalled threads
  - Requires 1000s of concurrent threads

GPU Glossary

Hardware	Software
(CUDA) Core	Thread/Work Unit
Streaming Multiprocessor (SM)	Thread Block/Work Group

- A *Grid* is a group of related *Thread Blocks* running the same kernel
- A *Warp* is Nvidia's term for 32 *Threads* running in lock-step
- *Warp Diversion* is what happens when some threads within a warp stall due to a branch
- *Shared Memory* is a user-managed cache within a *Thread Block*
- *Occupancy* is the degree to which all of the GPU hardware can be used in a *Kernel*
  - Heavily influenced by registers/thread and threads/block
- *Stream* is a series of data transfers and kernel launches that happen in series

GPU PROGRAMMING MODELS

Section Goals

- Introduce several GPU programming models
- Discuss why someone may choose one programming paradigm over the others.

Explicit/Implicit GPU Programming

Explicit

- Bottom-up approach
- Explicit Kernel written from threads' perspective
- Memory management controlled by programmer
- Thread Blocks & Grid defined by programmer
- GPU code usually distinct from CPU code

Implicit

- Traditional Top-down programming
  - *Big Picture*
- Compiler handles memory and thread management
  - May be guided by programmer
- CPU & GPU may use the same code
  - Easier code maintenance

GPU Programming Models

- Explicit
  - CUDA C (Free from Nvidia)
  - CUDA Fortran (Commercial from PGI)
  - OpenCL (Free from Multiple Vendors)
- Implicit
  - Proposed OpenMP Directives (Multiple Vendors)
  - PGI Directives (Commercial from PGI)
  - HMPP Directives (Commercial from CAPS)
  - Libraries (CUBLAS, MAGMA, etc.)

Multi-node Programming

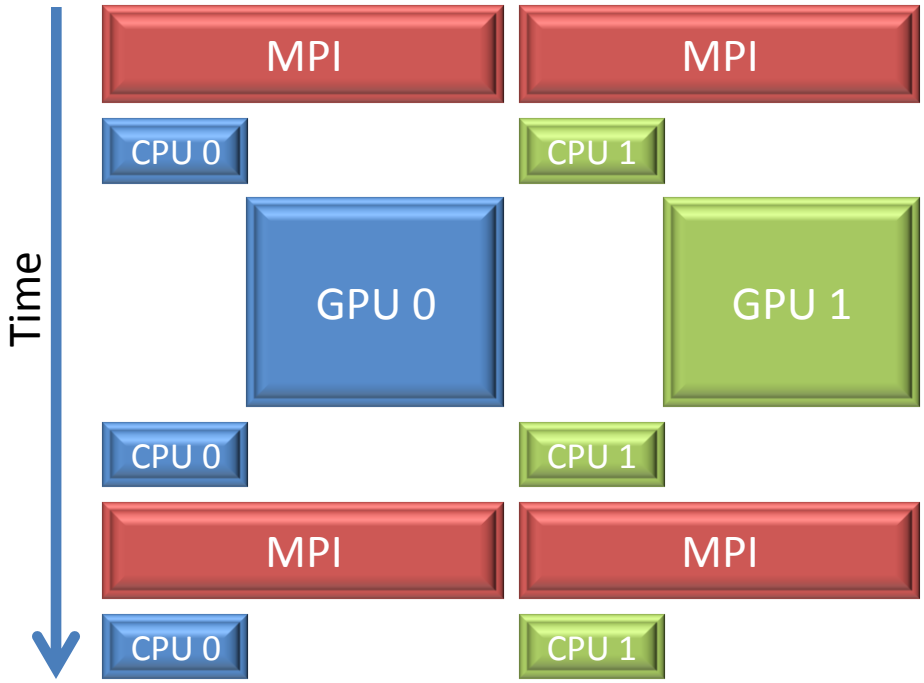
- GPU papers & tutorials usually focus on 1 node, what about the rest of the machine?

- High-level MPI parallelism between nodes
  - You're probably already doing this
- Loose, on-node parallelism via threads
  - Most codes today are using MPI, but threading is becoming more important
- Tight, on-node, vector parallelism
  - SSE/AVX on CPUs
  - GPU threaded parallelism

<u>Programmers need to expose the same parallelism with/without GPUs</u>

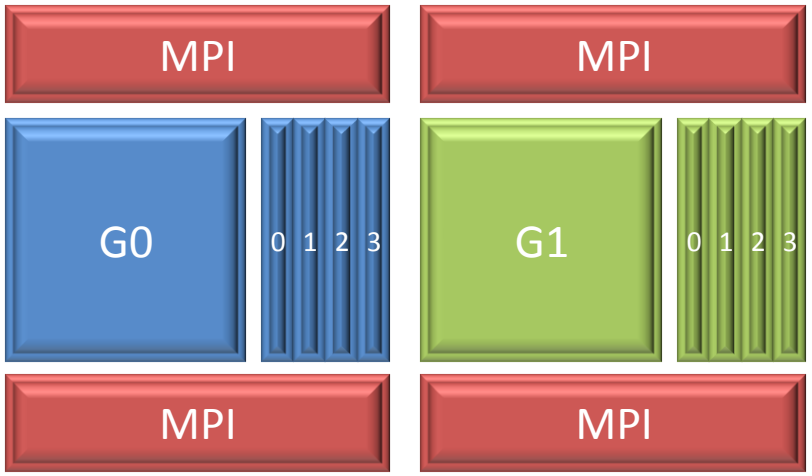
Using the Machine Efficiently

So-So Hybridization



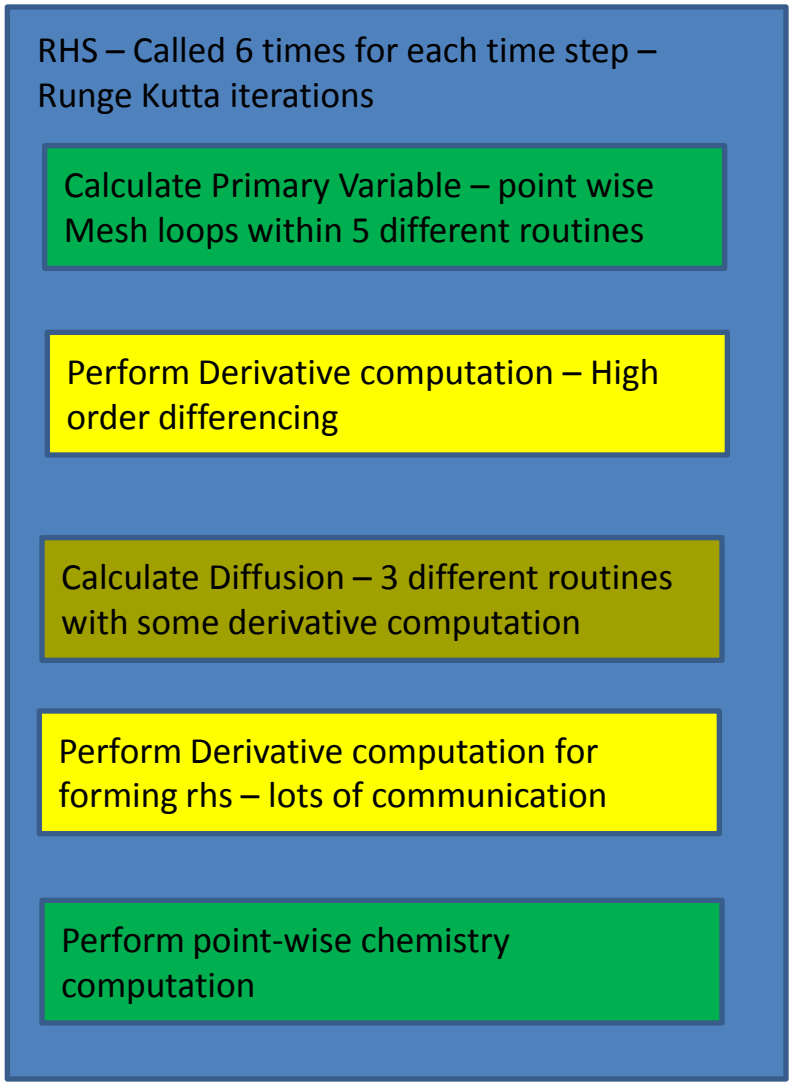
- Neglects the CPU
- Suffers from Amdahl's Law

Better Hybridization



- Overlap CPU/GPU work and data movement.
- Even better if you can overlap communication too!

Original S3D

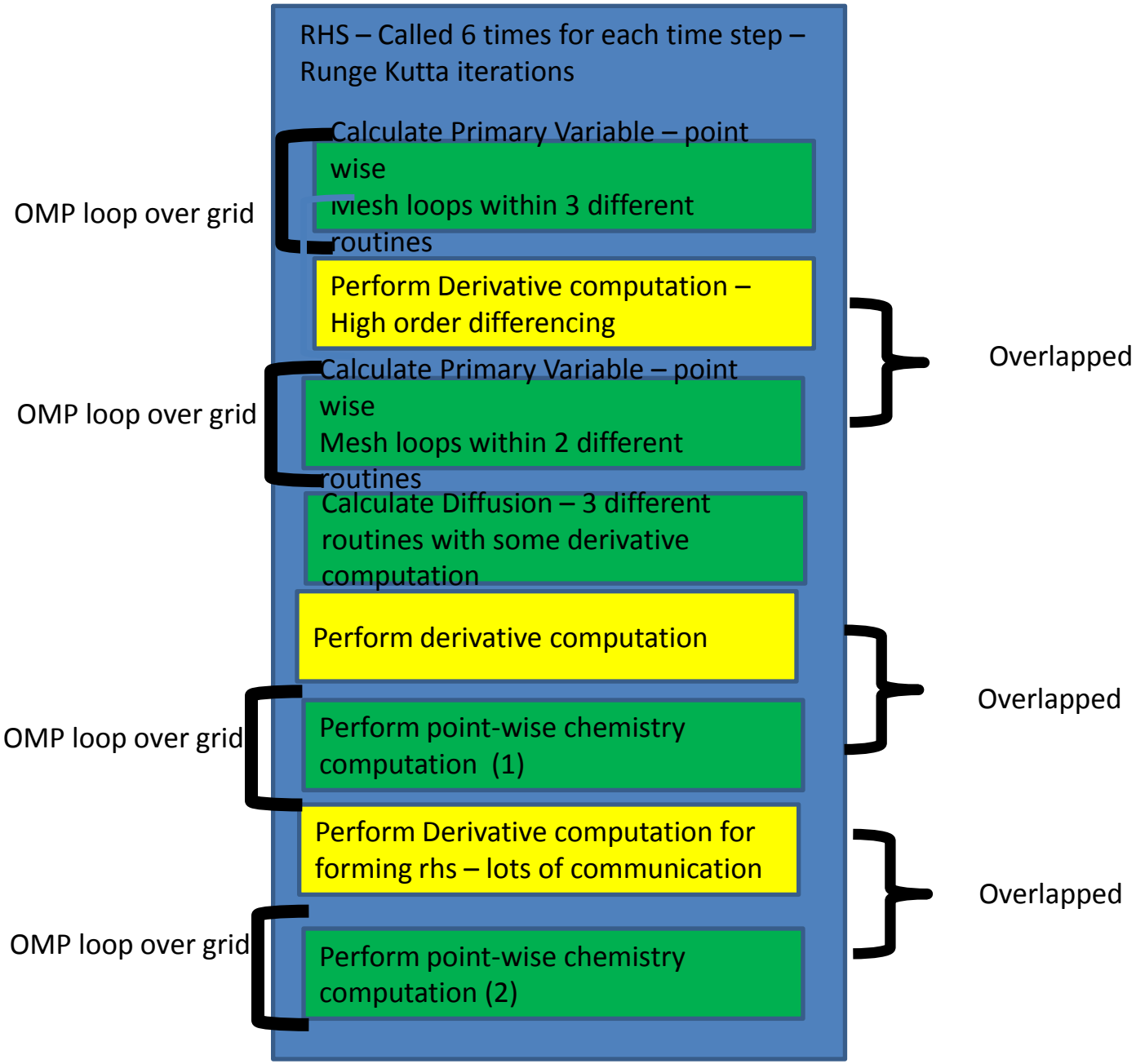


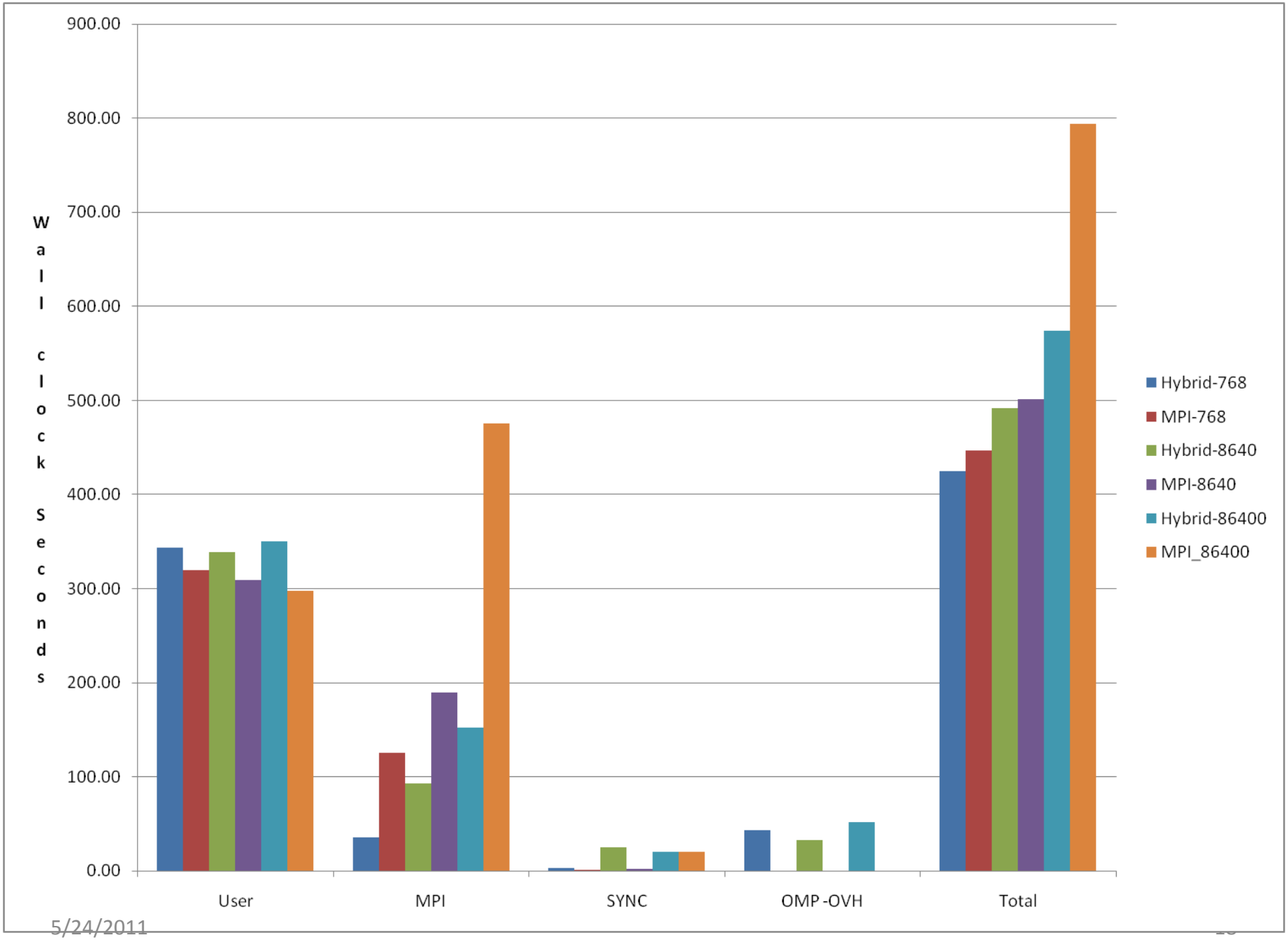
All major loops are at low level of the Call tree

Green – major computation – point-wise

Yellow – major computation – Halos 5 zones thick

Restructured S3D for multi-core systems




900.00
800.00
700.00
600.00
500.00
400.00
300.00
200.00
100.00
0.00
Wall clock Seconds
User
MPI
SYNC
OMP -OVH
Total
Hybrid-768
MPI-768
Hybrid-8640
MPI-8640
Hybrid-86400
MPI_86400

Explicit: CUDA C/Fortran & OpenCL

- Programmer writes a *kernel* in C/Fortran that will be run on the GPU
  - This is essentially the loop body from original CPU code
- GPU memory must be explicitly allocated, freed, and filled from CPU memory over PCIe
  - Generally results in 2 variables referring to every pertinent array, one in each memory domain (hostA, devA)
- Programmer declares how to decompose into thread blocks and grid
  - Must understand limits of thread block size and how to maximize occupancy
- CPU code launches kernel on device.
  - May continue to work while GPU executes kernel(s)

CUDA C Example

Host Code

```
double a[1000], *d_a;
dim3 block( 1000, 1, 1 );
dim3 grid( 1, 1, 1 );

cudaMalloc((void**)&d_a, 1000*sizeof(double));
cudaMemcpy(d_a, a,
      1000*sizeof(double),cudaMemcpyHostToDev
      ice);

scaleit_kernel<<<grid,block>>>(d_a,n);

cudaMemcpy(a, d_a,
      1000*sizeof(double),cudaMemcpyDeviceToH
      ost);

cudaFree(d_a);
```

Allocate & Copy to GPU

Launch

Copy Back & Free

GPU Code

```
__global__
void scaleit_kernel(double *a,int n)
{
  int i = threadIdx.x;

  if (i < n)
    a[i] = a[i] * 2.0l;
}
```

My Index

Calculate Myself

CUDA Fortran Example

Host Code

```
subroutine scaleit(a,n)
 real(8),intent(inout) :: a(n)
 real(8),device        :: d_a(n)
 integer,intent(in)    :: n
 type(dim3)            :: blk, grd

 blk = dim3(1000,1,1)
 grd = dim3(1,1,1)

 d_a = a
 call scaleit_kernel<<<grd,blk>>>(d_a,n)
 a = d_a
end subroutine
```

Declare on Device

Copy To Device

Launch & Copy Back

GPU Code

```
attributes(global)&
 subroutine scaleit_kernel(a,n)
   real(8),intent(inout)     :: a(n)
   integer,intent(in),value :: n
   integer I

   i = threadIdx%x

   if (i.le.n) then
     a(i) = 2.0 * a(i)
   endif
 end subroutine scaleit_kernel
```

My Index

Calculate Myself

Implicit: Directives

- Programmer adds directives to existing CPU code
- Compiler determines
  - Memory management
  - Thread management
- Programmer adds directives to guide compiler
  - Higher-level data regions
  - Partial array updates
  - Improved thread blocking

Proposed OpenMP Directives Example

```
real*8 a(1000)
integer i
!$omp acc_region_loop acc_copy(a)
do i=1,1000
  a(i) = 2 * a(i)
enddo
!$omp end acc_region_loop
```

Build for device, Copy a on and off

Implicit: Libraries

- Calls to existing Math libraries replaced with accelerated libraries
  - BLAS, LAPACK
  - FFT
  - Sparse kernels
- Unless application spends very high % of runtime in library calls, this will need to be combined with other methods

Libraries Example

```
info = cublas_set_matrix(lda, na, sizeof_Z, a, lda, devA, lda)

info = cula_device_zgetrf(m,m,devA+idx2f(ioff+1,ioff+1,lda)*sizeof_Z,lda,devIPVT)
info = cula_device_zgetrs('n',m,ioff,devA+idx2f(ioff+1,ioff+1,lda)*sizeof_Z,lda,devIPVT,
    &      devA+idx2f(ioff+1,1,lda)*sizeof_Z,lda)
call cublas_zgemm('n','n',n,ioff-k+1,na-ioff,cmone,devA+idx2f(joff+1,ioff+1,lda)*sizeof_Z,lda,
    &      devA+idx2f(ioff+1,k,lda)*sizeof_Z,lda,cone,devA+idx2f(joff+1,k,lda)*sizeof_Z,lda)
call cublas_zgemm('n','n',blk_sz(1),blk_sz(1)-k+1,na-blk_sz(1),
    &      cmone,devA+idx2f(1,blk_sz(1)+1,lda)*sizeof_Z,lda,
    &      devA+idx2f(blk_sz(1)+1,k,lda)*sizeof_Z,lda,cone,devA,lda)

info = cublas_get_matrix(lda, na, sizeof_Z, devA, lda, a, lda)
```

PERFORMANCE ANALYSIS

Section Goals

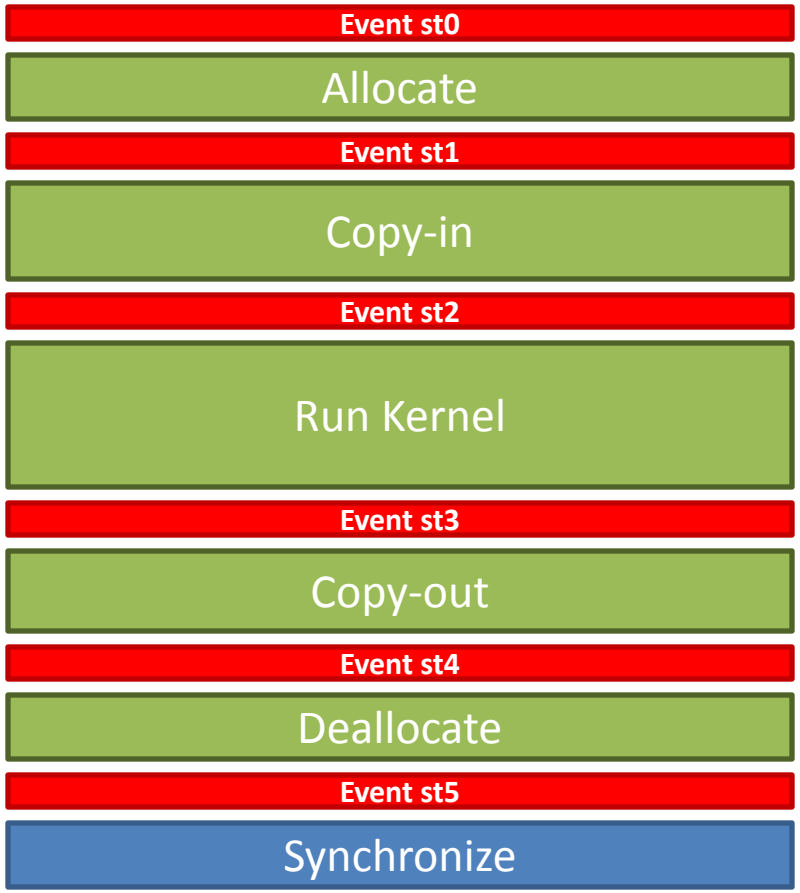
- Understand multiple options for gathering GPU performance metrics

- Increasing number of tools available, I'll cover 3 methods
  - Explicit event instrumentation
  - CUDA Profiler
  - CrayPAT Preview

CUDA Event API

- Most CUDA API calls are asynchronous: explicit CPU timers won't work
- CUDA allows inserting events into the stream
  - Insert an event before and after what needs to be timed
  - Synchronize with events
  - Calculate time between events
- Introduces small driver overhead and may synchronize asynchronous calls
  - Don't use in production

CUDA Event Example




```
ierr = cudaEventRecord(st0,0)
allocate(d_a(n))
ierr = cudaEventRecord(st1,0)
d_a = a
ierr = cudaEventRecord(st2,0)
call &
    scaleit_kernel<<<grd,blk>>>&
    (d_a,n)
ierr = cudaEventRecord(st3,0)
a = d_a
ierr = cudaEventRecord(st4,0)
deallocate(d_a)
ierr = cudaEventRecord(st5,0)
...
ierr = cudaEventSynchronize(st2)
ierr = cudaEventSynchronize(st3)
ierr = cudaEventElapsedTime &
       (et, st2, st3)
write(*,*)'Kernel Time',et
```

CUDA Profiler

- Silently built-in to CUDA driver and enabled via environment variable
  - Works with both CUDA and Directives programs
- Returns time of memory copies and kernel launches by default
  - Also reports kernel occupancy
  - Can be configured to report many other metrics
- All metrics are recorded at driver level and high resolution
  - May add small kernel overhead and synchronize asynchronous operations.

CUDA Profiler Example

```
# Enable Profiler
$ export CUDA_PROFILE=1
$ aprun ./a.out
$ cat cuda_profile_0.log

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla M2090
# TIMESTAMPFACTOR fffff6f3e9b1f6c0
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 2.304 ] cputime=[ 23.000 ]
method=[ _Z14scaleit_kernelPdi ] gputime=[ 4.096 ] cputime=[
   15.000 ] occupancy=[ 0.667 ]
method=[ memcpyDtoH ] gputime=[ 3.072 ] cputime=[ 34.000 ]
```

CUDA Profiler Example

```
# Customize Experiment
$ cat exp.txt
l1_global_load_miss
l1_global_load_hit
$ export CUDA_PROFILE_CONFIG=exp.txt
$ aprun ./a.out
$ cat cuda_profile_0.log

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla M2090
# TIMESTAMPFACTOR fffff6f4318519c8
method,gputime,cputime,occupancy,l1_global_load_miss,l1_global_load_hit
method=[ memcpyHtoD ] gputime=[ 2.240 ] cputime=[ 23.000 ]
method=[ _Z14scaleit_kernelPdi ] gputime=[ 4.000 ] cputime=[ 36.000 ]
   occupancy=[ 0.667 ] l1_global_load_miss=[ 63 ] l1_global_load_hit=[
   0 ]
method=[ memcpyDtoH ] gputime=[ 3.008 ] cputime=[ 33.000 ]
```

CrayPAT Prototype

- Luiz DeRose is giving a tutorial on CrayPAT future work at CUG (you're missing it right now)
- The goal of the CrayPAT team is to make instrumenting applications and understanding the results as simple as possible
  - No code modification
  - Derived metrics
  - Optimization suggestions
  - …
- Several new tools are being developed that will help with accelerator development

CrayPAT Preview: Performance Stats

```
5||||   1.3% |  21.836221 |  21.630958 |  6760.318 | 6760.318 |  3201 |collisionb_
||||||-------------------------------------------------------------------------------
6|||||   1.1% |  18.888240 |  18.708450 |     0.000 | 6507.596 |  1400 |collisionb_(exclusive)
|||||||-------------------------------------------------------------------------------
7||||||   0.4% |   7.306387 |   7.291820 |     0.000 |    0.000 |   200 |collisionb_.ASYNC_KERNEL@li.599
7||||||   0.4% |   7.158172 |   7.156827 |     0.000 |    0.000 |   200 |collisionb_.ASYNC_KERNEL@li.568
7||||||   0.2% |   3.799065 |   3.799065 |     0.000 | 6507.596 |   200 |collisionb_.SYNC_COPY@li.593
7||||||   0.0% |   0.527203 |   0.376397 |     0.000 |    0.000 |   200 |lbm3d2p_d_.ASYNC_COPY@li.129
7||||||   0.0% |   0.073654 |   0.064766 |     0.000 |    0.000 |   200 |collisionb_.ASYNC_COPY@li.703
7||||||   0.0% |   0.013917 |   0.011082 |     0.000 |    0.000 |   199 |grad_exchange_.ASYNC_COPY@li.428
7||||||   0.0% |   0.009707 |   0.008366 |     0.000 |    0.000 |   200 |collisionb_.ASYNC_KERNEL@li.581
7||||||   0.0% |   0.000134 |   0.000127 |     0.000 |    0.000 |     1 |collisionb_.ASYNC_COPY@li.566
6|||||   0.2% |   2.947981 |   2.922508 |  6760.318 |  252.722 |  1801 |grad_exchange_
|||||||-------------------------------------------------------------------------------
7||||||   0.1% |   2.485119 |   2.485119 |  6507.596 |    0.000 |   200 |collisionb_.SYNC_COPY@li.596
7||||||   0.0% |   0.107396 |   0.107396 |     0.000 |  126.361 |   200 |grad_exchange_.SYNC_COPY@li.472
7||||||   0.0% |   0.103009 |   0.103009 |   126.361 |    0.000 |   200 |grad_exchange_.SYNC_COPY@li.452
7||||||   0.0% |   0.065731 |   0.065731 |     0.000 |  126.361 |   200 |grad_exchange_.SYNC_COPY@li.439
7||||||   0.0% |   0.061754 |   0.061754 |   126.361 |    0.000 |   200 |grad_exchange_.SYNC_COPY@li.485
7||||||   0.0% |   0.056946 |   0.045612 |     0.000 |    0.000 |   200
     |grad_exchange_.ASYNC_KERNEL@li.453
7||||||   0.0% |   0.029640 |   0.028101 |     0.000 |    0.000 |   200
     |grad_exchange_.ASYNC_KERNEL@li.430
7||||||   0.0% |   0.025947 |   0.014719 |     0.000 |    0.000 |   200
     |grad_exchange_.ASYNC_KERNEL@li.486
7||||||   0.0% |   0.012368 |   0.011011 |     0.000 |    0.000 |   200 |grad_exchange_.ASYNC_COPY@li.496
7||||||   0.0% |   0.000070 |   0.000056 |     0.000 |    0.000 |     1 |grad_exchange_.ASYNC_COPY@li.428
```

This example is taken from a real user application and "ported" using proposed OpenMP extensions.

CrayPAT Preview: Data Transfer Stats

```
Host | Host Time |  Acc Time |  Acc Copy |  Acc Copy | Calls |Group='ACCELERATOR'
 Time % |         |         |   In (MB) |  Out (MB) |       | PE
 100.0% | 42.763019 | 42.720514 | 21877.192 | 20076.420 |   703 |Total
|-------------------------------------------------------------------------------
| 100.0% | 42.763019 | 42.720514 | 21877.192 | 20076.420 |   703 |ACCELERATOR
||------------------------------------------------------------------------------
5||||  4.6% | 31.319188 | 31.318755 | 19425.659 | 19425.659 |   140 |recolor_
||||||----------------------------------------------------------------------------
6|||||  4.5% | 30.661050 | 30.660616 | 18454.376 | 19425.659 |   139 |recolor_(exclusive)
|||||||---------------------------------------------------------------------------
7||||||  2.4% | 16.761967 | 16.761967 |    0.000 | 19425.659 |    20 |recolor_.SYNC_COPY@li.790
7||||||  1.9% | 13.227889 | 13.227889 | 18454.376 |    0.000 |    19 |recolor_.SYNC_COPY@li.793
7||||||  0.1% |  0.668515 |  0.668480 |    0.000 |    0.000 |    20 |recolor_.ASYNC_KERNEL@li.781
7||||||  0.0% |  0.002122 |  0.002059 |    0.000 |    0.000 |    20 |lbm3d2p_d_.ASYNC_COPY@li.118
7||||||  0.0% |  0.000332 |  0.000105 |    0.000 |    0.000 |    20 |recolor_.ASYNC_COPY@li.794
7||||||  0.0% |  0.000116 |  0.000057 |    0.000 |    0.000 |    20 |recolor_.ASYNC_COPY@li.789
7||||||  0.0% |  0.000110 |  0.000060 |    0.000 |    0.000 |    20 |recolor_.ASYNC_COPY@li.781
|||||||===========================================================================
6|||||  0.1% |  0.658138 |  0.658138 |   971.283 |    0.000 |     1 |streaming_exchange_
7|||||      |         |         |         |         |       | recolor_.SYNC_COPY@li.793
```

Full PCIe data transfer information without any code modifications.

Cray Tools: More Information

- Cray is developing a lot of tools that deserve more time than this tutorial allows, so…

- Go to “Cray GPU Programming Tools” BOF at 4:15 on Wednesday (Track 15B)
- Talk to Luiz DeRose and/or Heidi Poxon while you’re here.

BASIC OPTIMIZATIONS

Basic Optimizations

OCCUPANCY

Calculating Occupancy

- Occupancy is the degree to which the hardware is saturated by your kernel
  - Generally higher occupancy results in higher performance
- Heavily affected by
  - Thread decomposition
  - Register usage
  - Shared memory use
- Nvidia provides an “occupancy calculator” spreadsheet as part of the SDK
  - Live example to follow

Calculating Occupancy

1. Get the register count

```
ptxas info    : Compiling entry function
    'laplace_sphere_wk_kernel3' for 'sm_20'
ptxas info    : Used 36 registers, 7808+0 bytes
    smem, 88 bytes cmem[0], 768 bytes cmem[2]
```

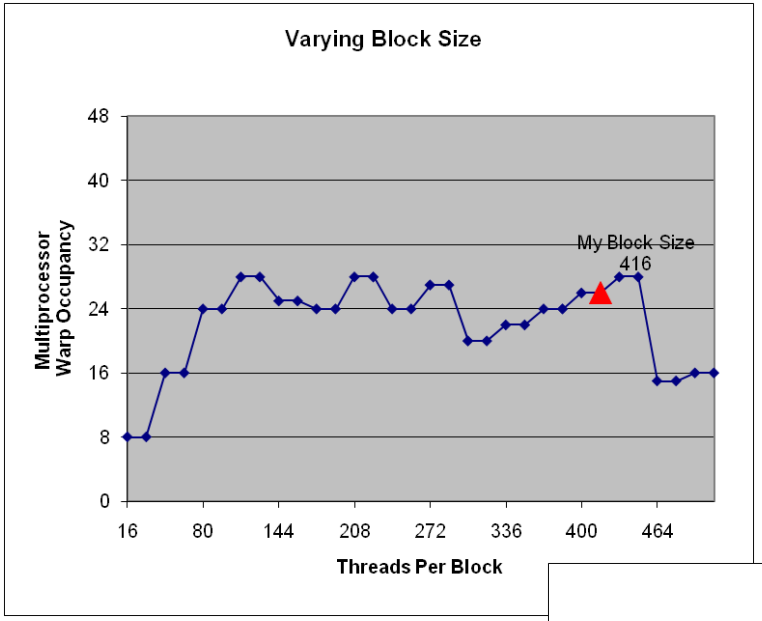
2. Get the thread decomposition

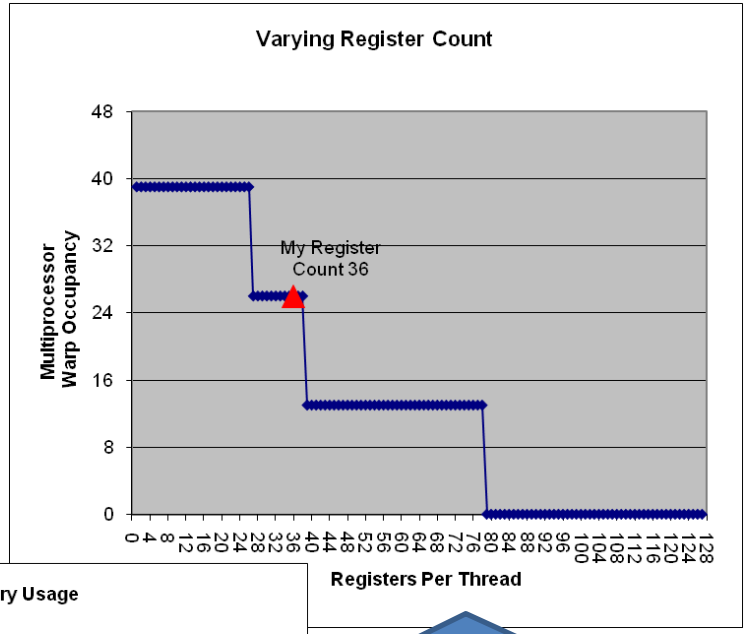
```
 blockdim = dim3( 4, 4, 26)
 griddim = dim3(101, 16, 1)
```

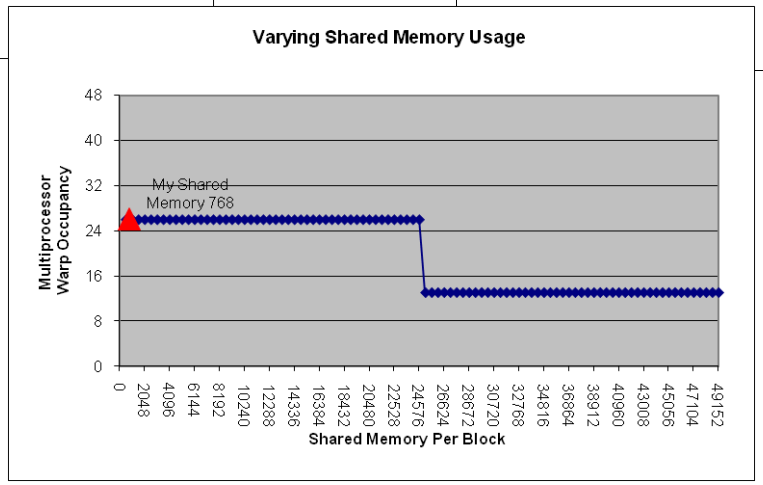
3. Enter into occupancy calculator

Result: 54%

Improving the Results



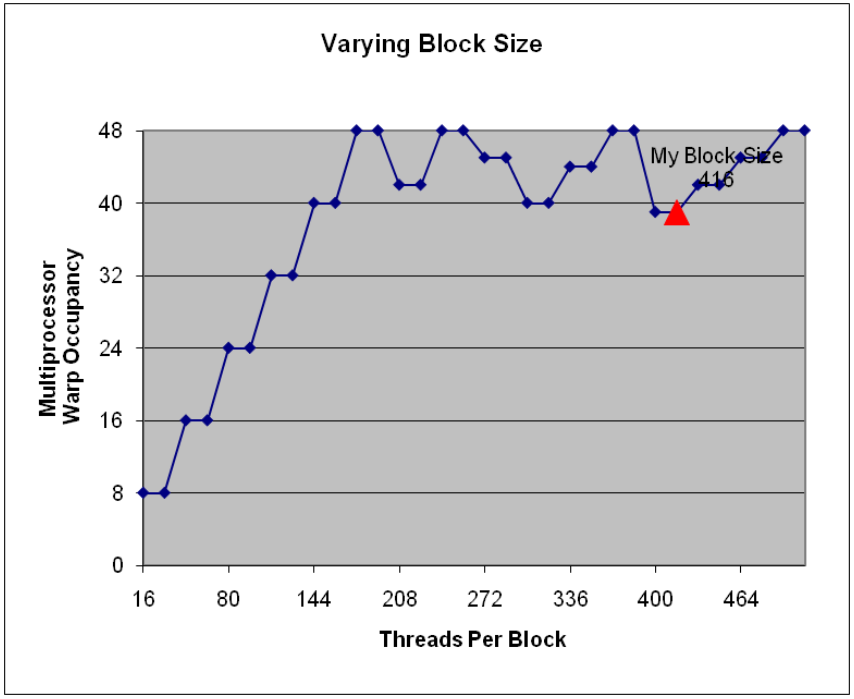




Varying #threads or shared memory use has little effect

Reducing registers per thread may increase occupancy.

Reducing Registers/Thread



- Maximum number of registers/thread can be set via compiler flag
- Reducing the number of registers/thread to 18 increases occupancy to 81%
- Time Before: 924us
- Time After: 837us
- Improvement: ~10%
- Occupancy isn’t a silver bullet

Occupancy Case Study

- Results from a Finite Difference Kernel, provided by Paulius Micikevicius of Nvidia
- Default compilation
  - 46 registers, no spills to lmem
  - runs a single 32x16 threadblock per SM concurrently
  - Occupancy: 33%
  - 3,395 MCells/s throughput (39.54ms)

Occupancy Case Study cont.

- Reducing Maximum Registers to 32
  - Set maximum register count via compiler flag
  - 32 registers, 44 bytes spilled to lmem
  - runs two 32x16 threadblocks per SM concurrently
  - Occupancy: 67%
  - 4,275 MCells/s (31.40ms)

- Improvement: ~26%

Basic Optimizations

ASYNCHRONICITY

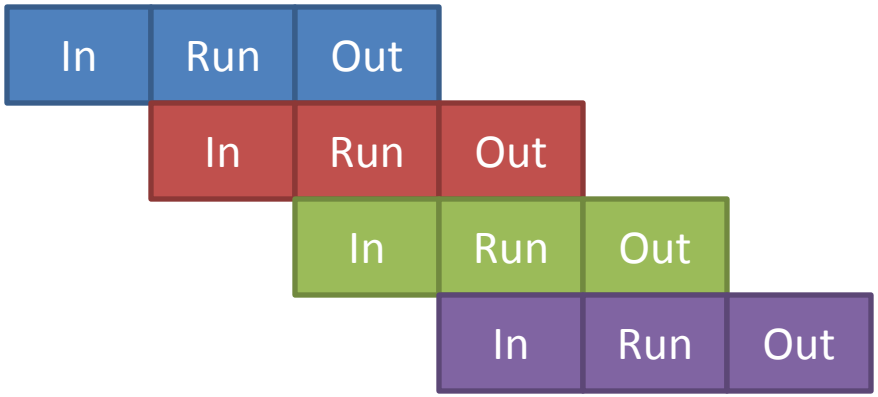
Asynchronous Execution

- Most GPU Operations are Asynchronous from the CPU code
  - Hint: The CPU can be busy doing other things
- Current Hardware can handle 1 Copy-in, 1 Kernel, and 1 Copy-out simultaneous, if in separate streams
  - Hint: Data transfer costs can be hidden by running multiple streams and asynchronous tranfers

Asynchronous Execution with Streams

- Synchronous Execution (1 Stream):

- Asynchronous Execution (3 Streams):



- If data cannot remain resident on device, streaming may allow GPU to offset transfer costs

Asynchronous Execution: Example

- Add some number of streams to existing code
- Use Asynchronous memory copies to copy part of data to/from device
  - GOTCHA: Host arrays must be "pinned" in order to use Async copies
- Add stream parameter to kernel launch

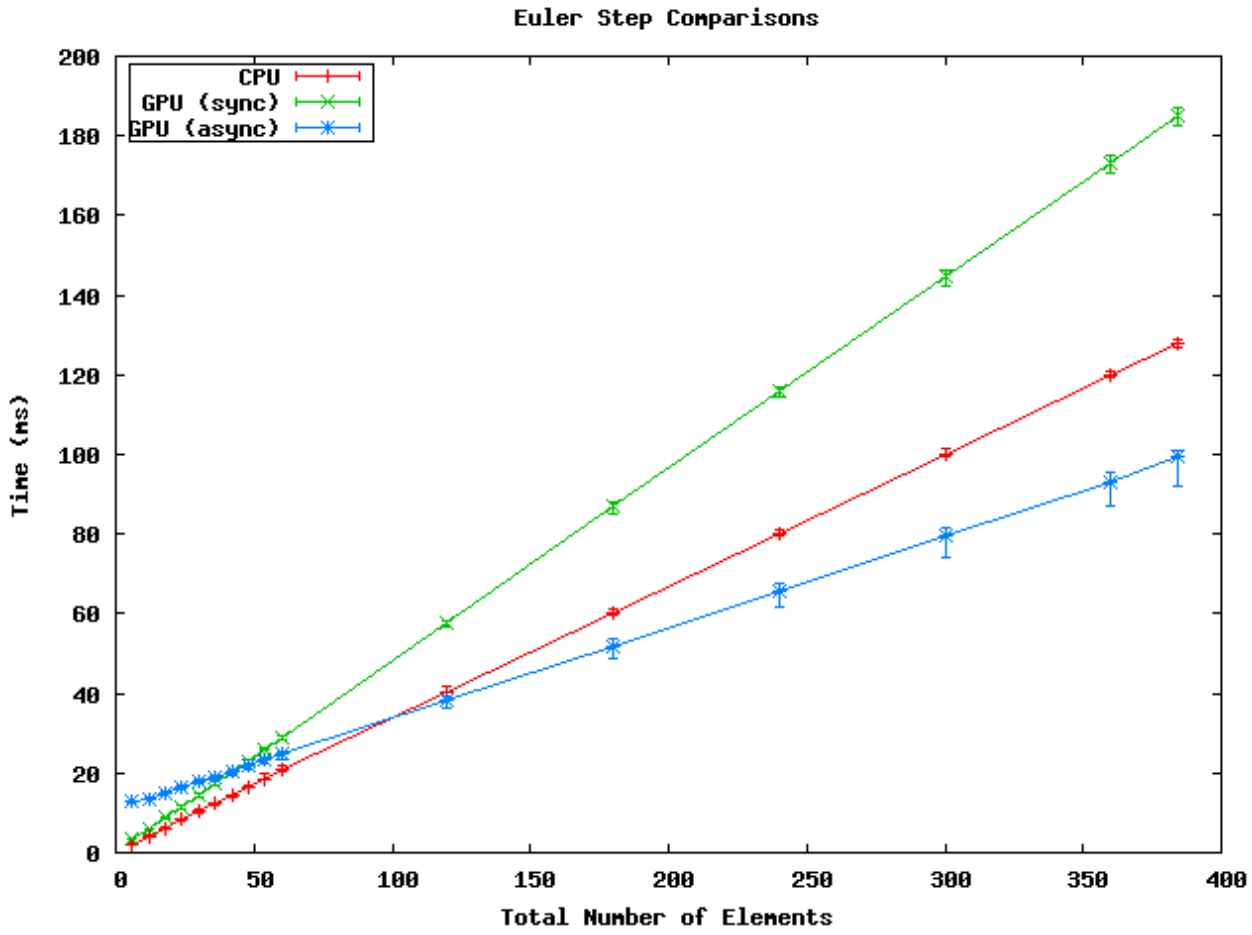
- Sync Time: 0.6987200
- Async Time: 0.2472000

```
integer :: streams(3)
integer :: ierr,j,mystream

do j=1,3
  ierr = cudaStreamCreate(streams(j))
 enddo

do j=1,m
  mystream = mod(j,3)
  ierr = cudaMemcpyAsync&
     (d_a(:,j),a(:,j),size(a(:,j)),streams(mystream))
  call
       scaleit_kernel<<<grd,blk,0,streams(mystrea
       m)>>>(d_a(:,j),n)
  ierr = cudaMemcpyAsync&
      (a(:,j),d_a(:,j),size(a(:,j)),streams(mystream))
 enddo
ierr = cudaStreamSynchronize(streams(1))
ierr = cudaStreamSynchronize(streams(2))
ierr = cudaStreamSynchronize(streams(3))
```

Asynchronous Case Study



CAVEAT: The above kernel over-emphasizes data transfer, thus necessitating streaming.

Basic Optimizations

SHARED MEMORY

Shared Memory

- Much like CPU cache, shared memory is much faster than global memory (up to 100X lower latency)
  - Staging Area
  - Scratch Pad
- 64KB Shared Memory sits on each SM
  - With Fermi, this is split between User-Manager and L1: 48/16 or 16/48
  - Split can be determined kernel to kernel
- If data is shared between threads in a thread block or reused well, staging it into shared memory may be beneficial
  - Think: Cache Prefetching

Simple Matrix Multiply

```
 attributes(global)&
  subroutine mm1_kernel(C,A,B,N)
    integer, value, intent(in) :: N
    real(8), intent(in) ::
A(N,N),B(N,N)
    real(8), intent(inout) :: C(N,N)

    integer i,j,k
    real(8) :: val

    i = (blockIdx%x - 1) * blockDim%x
+ threadIdx%x
    j = (blockIdx%y - 1) * blockDim%y
+ threadIdx%y

    val = C(i,j)
    do k=1,N
      val = val + A(i,k) * B(k,j)
    enddo
    C(i,j) = val
end
```

```
ptxas info    : Compiling entry
   function 'mm1_kernel' for
   'sm_20'
ptxas info    : Used 22
   registers, 60 bytes cmem[0]
```

- No shared memory use, totally relies on hardware L1

Kernel	Time (ms)	Occupancy
Simple	269.0917	67%

Tiled Matrix Multiply

```
integer,parameter :: M = 32
real(8),shared :: AS(M,M),BS(M,M)
real(8) :: val

val = C(i,j)

do blk=1,N,M
  AS(threadIdx%x,threadIdx%y) = &
  A(blk+threadIdx%x-1,blk+threadIdx%y-1)
  BS(threadIdx%x,threadIdx%y) = &
  B(blk+threadIdx%x-1,blk+threadIdx%y-1)
  call syncthreads()

  do k=1,M
    val = val + AS(threadIdx%x,k) &
              * BS(k,threadIdx%y)
  enddo
  call syncthreads()
enddo
C(i,j) = val
endif
```

```
ptxas info    : Compiling entry
   function 'mm2_kernel' for
   'sm_20'
ptxas info    : Used 18
   registers, 16384+0 bytes
   smem, 60 bytes cmem[0], 4
   bytes cmem[16]
```

- Now uses 16K of shared memory

Kernel	Time (ms)	Occupancy
Simple	269.0917	67%
Tiled	213.7160	67%

What if we increase the occupancy?

- With 32x32 blocks, we'll never get above 67%
- Reduce block size from 32x32 to 16x16?

Kernel	Time (ms)	Occupancy
Simple (32x32)	269.0917	67%
Tiled (32x32)	213.7160	67%
Simple (16x16)	371.7050	83%
Tiled (16x16)	209.8233	83%

- Reduce Max Registers to 18?

Kernel	Time (ms)	Occupancy
Simple (16x16)	371.7050	83%
Tiled (16x16)	209.8233	83%
Simple (16x16) 18 registers	345.7340	100%
Tiled (16x16) 18 registers	212.2826	100%

- Turns out the 16 is even worse.

Basic Optimizations

MEMORY COALESCING

Coalescing Memory Accesses

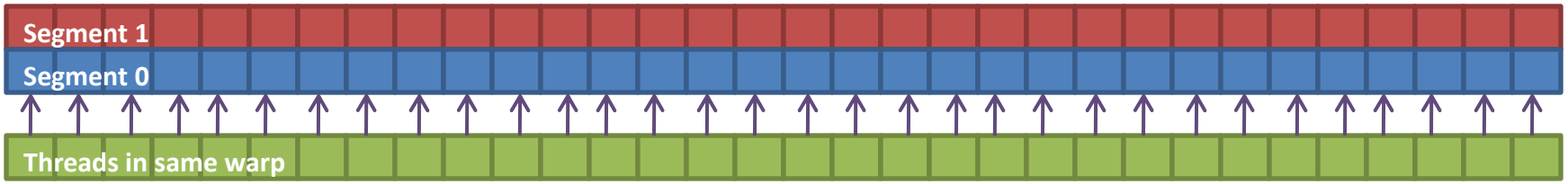
- The GPU will try to load needed memory in as few memory transactions as possible.
  - 128 B if possible
  - If not, 2 X 64 B
  - If not, 64 B may be split to 32 B
  - Continue until every thread has needed data
- Coalescing is possible if:
  - 128B aligned
  - All threads access elements in same segment

Why is coalescing important?

- Issuing 1 128B transaction reduces memory latency and better utilizes memory bandwidth
- L1/Shared Memory cache lines are 128B
  - Not using all fetched addresses wastes bandwidth
- Nvidia Guide: “Because of this possible performance degradation, memory coalescing is the most critical aspect of performance optimization of device memory.”

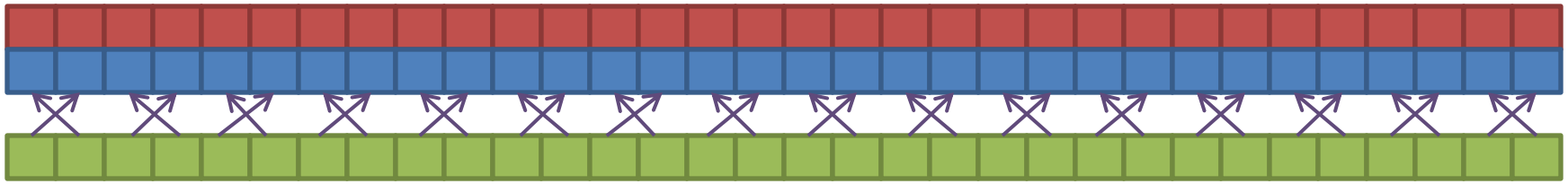
Coalescing Examples

Simple, Stride-1:

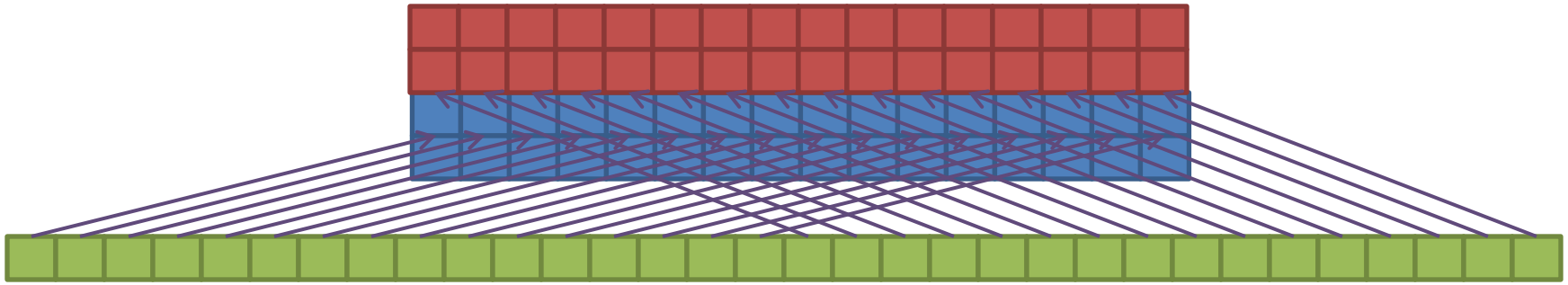


Every thread accesses memory within same 128B-aligned memory segment, so the hardware will coalesce into 1 transaction.

Will This Coalesce?



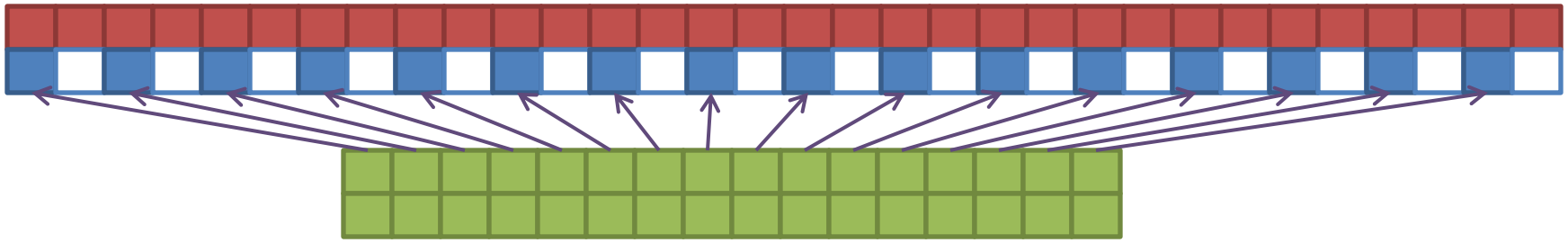
Yes! Every thread is still accessing memory within a single 128B segment and segment is 128B aligned.



No. Although this is stride-1, it is misaligned, accessing 2 128B segments. 2 64B transactions will result.

Will This Coalesce?

Stride-2, half warp:



Yes, but..

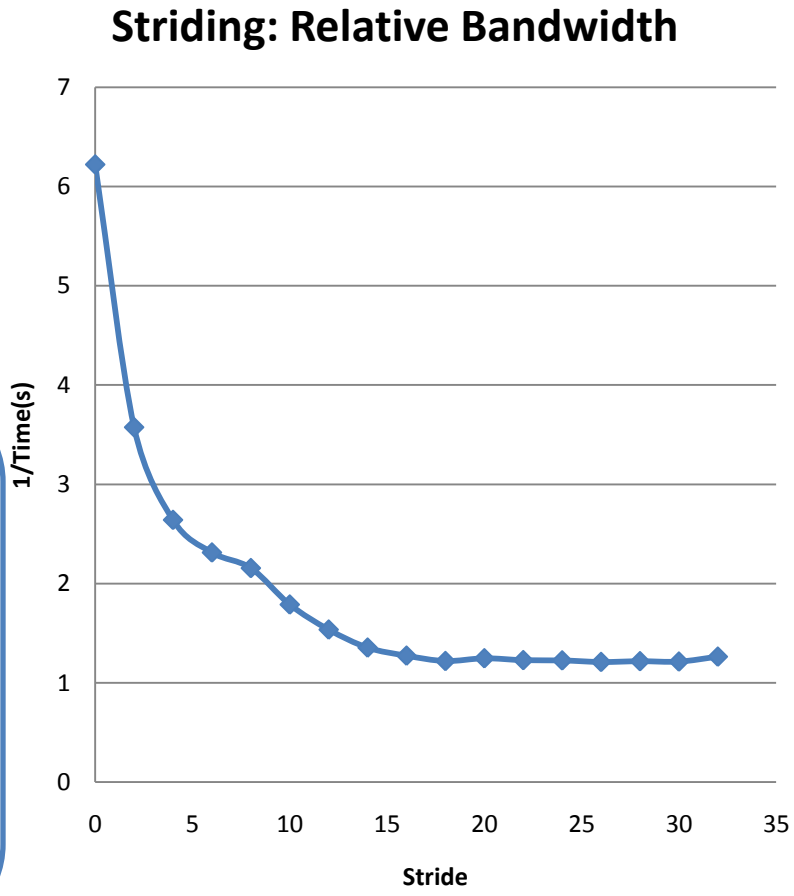
- Half of the memory transaction is wasted.
- Poor utilization of the memory bus.

Striding

- Striding results in more memory transactions and wastes cache line entries

```
 attributes(global)&
  subroutine stride_kernel(datin,
datout, st)
    integer,value :: st
    real(8) :: datin(n), datout(n)
    integer i

    i = (blockIdx%x * blockDim%x ) &
      + (threadIdx%x * st)
    datout(i) = datin(i)
  end subroutine stride_kernel
```

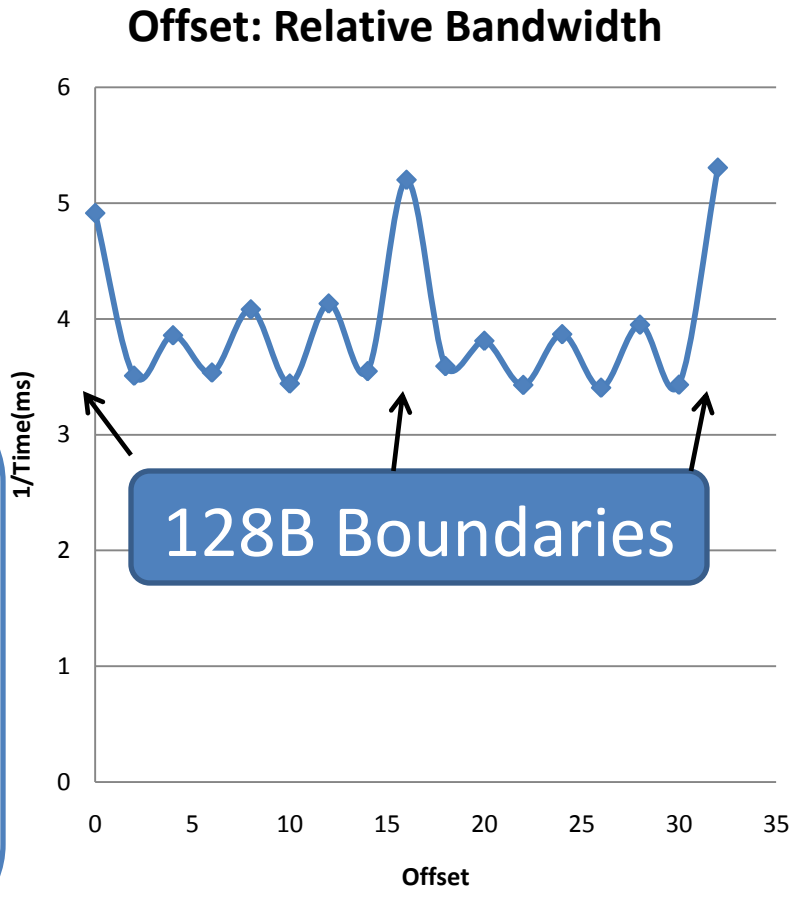


Offsets (Not 128B-aligned)

- Memory offsets result in more memory transactions by crossing segment boundaries

```
  attributes(global)&
  subroutine offset_kernel(datin,
datout, st)
    integer,value :: st
    real(8) :: datin(n), datout(n)
    integer i

    i = (blockIdx%x * blockDim%x ) &
      + threadIdx%x + st
    datout(i) = datin(i)
  end subroutine offset_kernel
```



ADDITIONAL RESOURCES

On The Web

- GTC 2010 Tutorials: http://www.nvidia.com/object/gtc2010-presentation-archive.html
- Nvidia CUDA online resources: http://developer.nvidia.com/cuda-education-training
- PGI CUDA Fortran: http://www.pgroup.com/resources/cudafortran.htm