Performance Optimization

- Host-device data transfers
  - Page-locked transfers
  - Asynchronous transfers
- Device memory
  - Coalescing
  - Shared memory
  - Textures
- Execution Configuration
  - Thread-level parallelism
  - Instruction-level parallelism7

Coalescing

- Device data accessed by a group of 16 or 32 sequential threads can result in as little as a single transaction if certain requirements are met
  - Alignment
  - Stride
  - Generation of GPU architecture

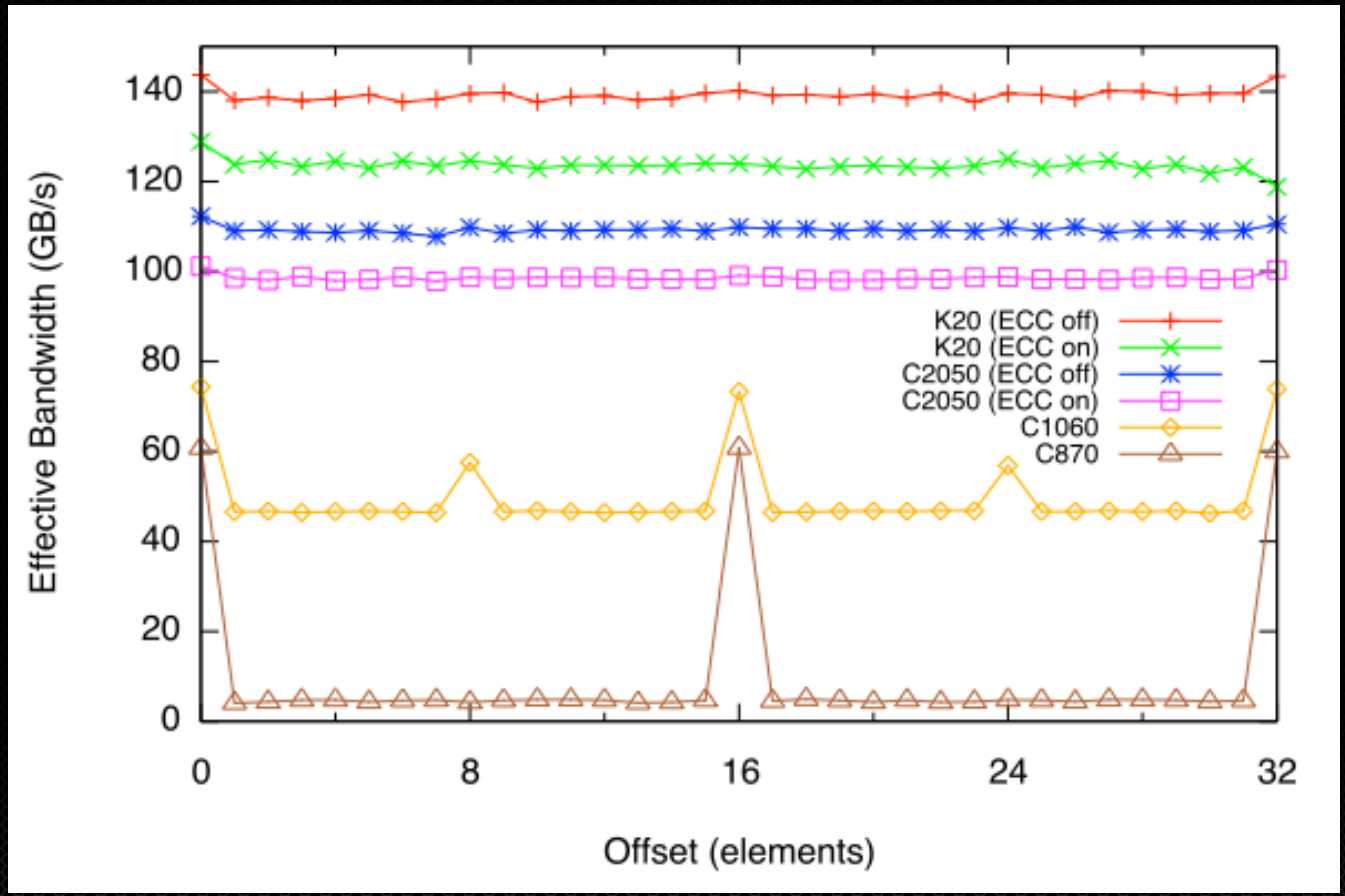
Misaligned Data Access

- Use array increment kernel with variable misalignment

```
attributes(global) subroutine offset(a, s)
  real :: a(*)
  integer, value :: s
  integer :: i
  i = blockDim%x*(blockIdx%x-1)+threadIdx%x + s
  a(i) = a(i)+1
end subroutine offset
```

Array `a()` *allocated with padding to avoid out-of-bounds accesses*

Misaligned Data Access



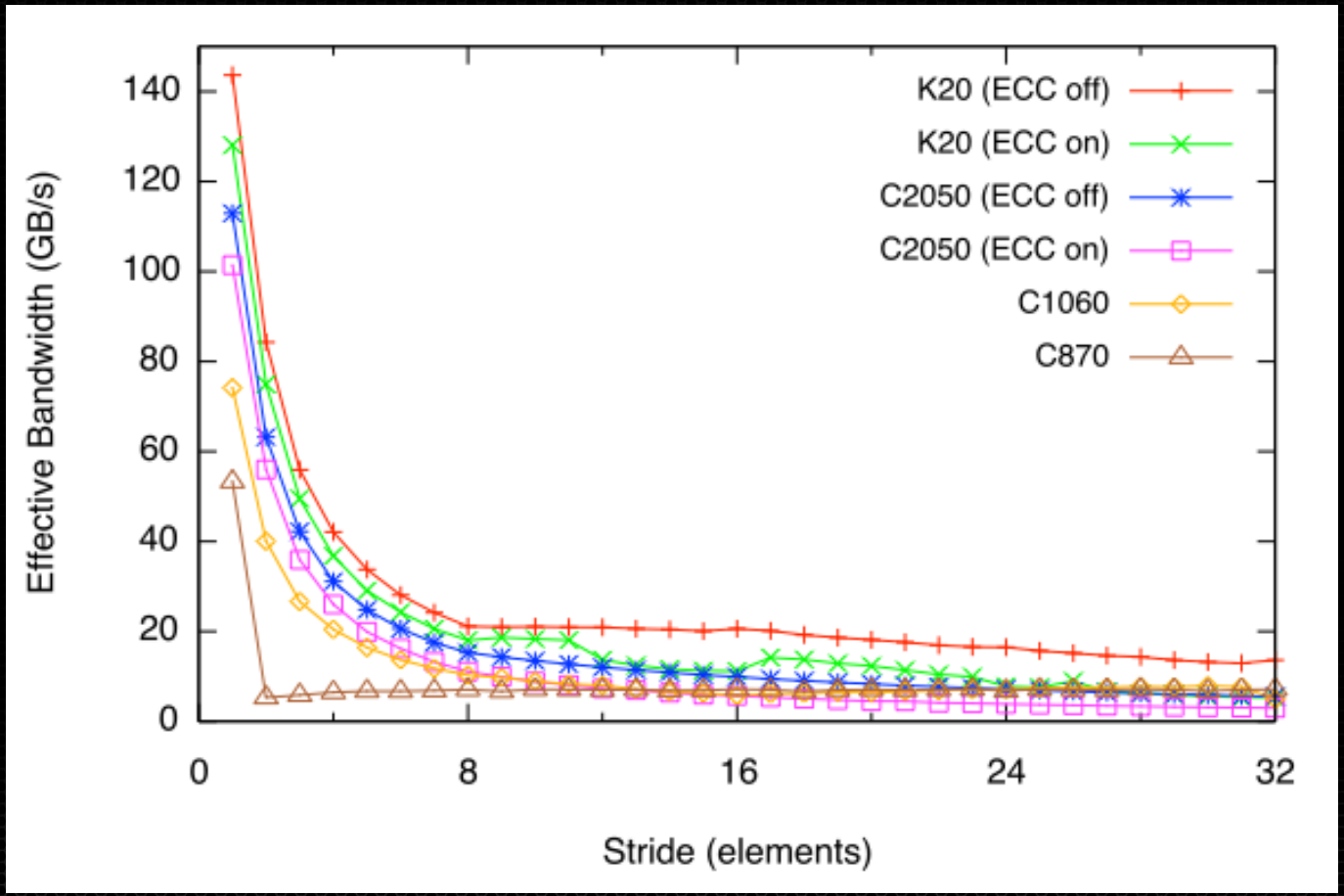
Strided Data Access

- Use array increment kernel with variable stride

```
attributes(global) subroutine stride(a, s)
  real :: a(*)
  integer,  value :: s
  integer :: i
  i = (blockDim%x*(blockIdx%x-1)+threadIdx%x) * s
  a(i) = a(i)+1
end subroutine stride
```

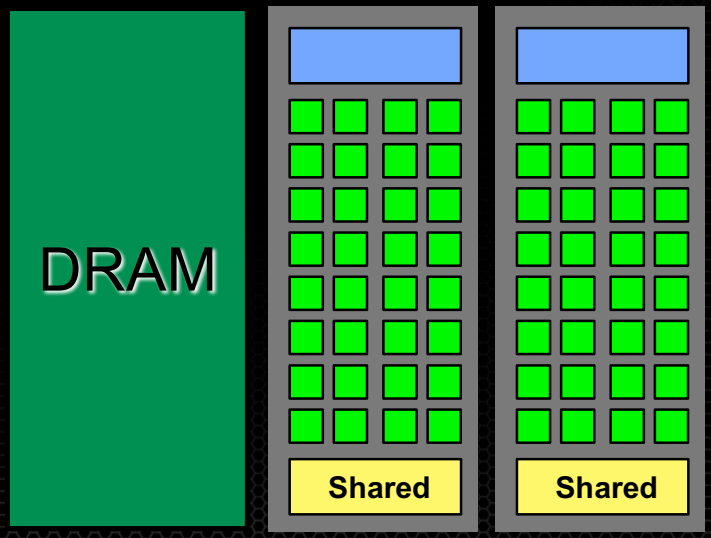
Array `a()` *allocated with padding to avoid out-of-bounds accesses*

Strided Data Access



Shared Memory

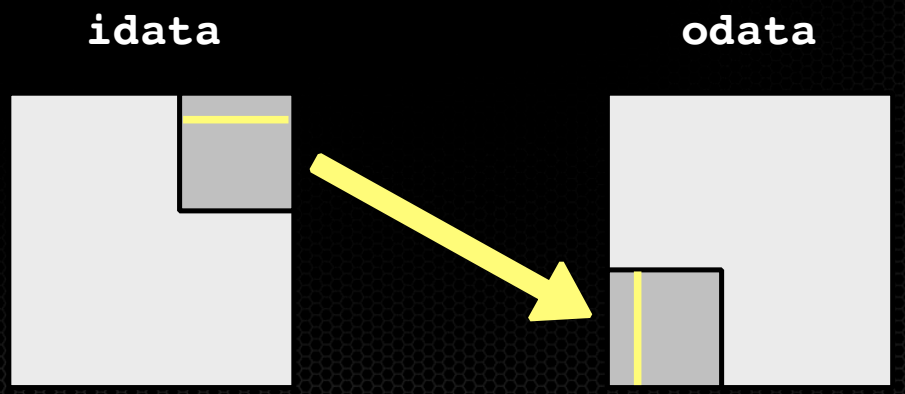
- On-chip
- All threads in a block have access to same shared memory
- Used to reduce multiple loads of device data
- Used to accommodate coalescing



Matrix Transpose (transpose.cuf)

```
attributes(global) subroutine transposeNaive(odata, idata)
   real, intent(out) :: odata(ny,nx)
   real, intent(in) :: idata(nx,ny)
   integer :: x, y

   x = (blockIdx%x-1) * blockDim%x + threadIdx%x
   y = (blockIdx%y-1) * blockDim%y + threadIdx%y
   odata(y,x) = idata(x,y)
 end subroutine transposeNaive
```



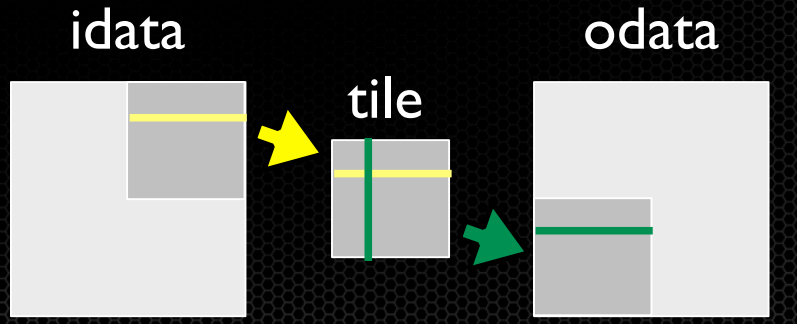
Matrix Transpose - Shared Memory

```
attributes(global) subroutine transposeCoalesced(odata, idata)
  real, intent(out) :: odata(ny,nx)
  real, intent(in) :: idata(nx,ny)
  real, shared :: tile(TILE_DIM, TILE_DIM)
  integer :: x, y

  x = (blockIdx%x-1)*blockDim%x + threadIdx%x
  y = (blockIdx%y-1)*blockDim%y + threadIdx%y
  tile(threadIdx%x, threadIdx%y) = idata(x,y)

  call syncthreads()

  x = (blockIdx%y-1)*blockDim%y + threadIdx%x
  y = (blockIdx%x-1)*blockDim%x + threadIdx%y
  odata(x,y) = tile(threadIdx%y, threadIdx%x)
end subroutine transposeCoalesced
```



Shared Memory Bank Conflicts

- Shared memory is divided into banks that can be accessed simultaneously
- Successive 32-bit words are assigned to successive banks
- Requests for different data in the same bank by threads in a warp (SIMD unit of 32 successive threads) are serialized
- Common workaround is to pad shared memory arrays

```
real, shared :: tile(TILE_DIM +1, TILE_DIM)
```

Constant Memory

- Small (64KB), read-only in a kernel, written by the host
  - assignment or API
  - used for small coefficient tables, common pointers
- Hardware cached
  - separate 16KB constant cache

Textures

- Different pathway for accessing data in device DRAM
- Read-only by device code
- Cached on chip in texture cache
- Uses F90 pointer notation
  - Declare texture at module scope in module where used

```
real, texture, pointer :: aTex(:)
```

  - Bind and unbind texture in host code using pointer notation
- Equivalent to tex1Dfetch() in CUDA C
  - No filtering or wrapping modes

Texture Memory as a Read-only Cache

CUDA Fortran module

```
real, texture, pointer :: q(:)
```

CUDA Fortran host code

```
real, device, target :: p(:)
. . .
allocate(p(n))
q => p
```

CUDA Fortran device code

```
i = threadIdx%x
j = (blockIdx%x-1)*blockDim%x + i
s(j) = s(j) + q(i)
```

CUDA C/C++ global

```
texture<float, 1,
   cudaReadModeElementType> tq;
```

CUDA C/C++ host code

```
float *d_p;
. . .
cudaMalloc((void **) d_p, size);
cudaChannelFormatDesc desc =
   cudaCreateChannelDesc(32, 0, 0,
         0, cudaFormatKindFloat);
cudaBindTexture(0, tq, d_p,
                  desc, size);
```

CUDA C/C++ device code

```
i = threadIdx.x;
j = blockIdx.x*blockDim.x + i;
s[j] = s[j] + tex1Dfetch(tq, i);
```

Texture Memory Performance (Measured Gbytes/sec)

```
module memtests
integer, texture, pointer :: t(:)
contains
  attributes(global) subroutine stridem(m,b,s)
  integer, device :: m(*), b(*)
  integer, value :: s
  i = blockDim%x*(blockIdx%x-1) + threadIdx%x
  j = i * s
  b(i) = m(j) + 1
  return
  end subroutine

  attributes(global) subroutine stridet(b,s)
  integer, device :: b(*)
  integer, value :: s
  i = blockDim%x*(blockIdx%x-1) + threadIdx%x
  j = i * s
  b(i) = t(j) + 1
  return
  end subroutine
end module memtests
```

	blockDim = 32		blockDim = 128	
stride	stridem	stridet	stridem	stridet
1	53.45	50.94	141.93	135.43
2	49.77	49.93	100.37	99.61
3	46.96	48.07	75.25	74.91
4	44.25	45.90	60.19	60.00
5	39.06	39.89	50.09	49.96
6	37.14	39.33	42.95	42.93
7	32.88	35.94	37.56	37.50
8	32.48	32.98	33.38	33.42
9	29.90	32.94	30.01	33.38
10	27.43	32.86	27.28	33.07
11	25.17	32.77	24.98	32.91
12	23.23	33.19	23.07	33.01
13	21.57	33.13	21.40	32.26
14	20.15	32.98	19.97	31.76
15	18.87	32.80	18.72	30.80
16	17.78	32.66	17.60	31.83
20	14.38	33.37	14.23	26.84
24	12.08	33.71	11.94	24.30
28	10.41	33.38	10.27	19.97
32	9.15	33.42	9.02	20.19

Texture Example

- strideTex.cuf

Alternative to texture fetches

- Use INTENT(IN) as attribute for dummy arguments

```
attributes(global) subroutine test( a, b )
integer, device :: a(*)
integer, device, intent(in) :: b(*)
```

- Compile with -Mcuda=keeptx,cc35
  - Verify with search for ld.global.nc operations in ptx file
- Compile with -Mcuda=keepbin,cc35
  - Verify with “cuobjdump -dump-sass” on the .bin file

Performance Optimization

- Host-device data transfers
  - Page-locked transfers
  - Asynchronous transfers
- Device memory
  - Coalescing
  - Shared memory
  - Textures
- Execution Configuration
  - Thread-level parallelism
  - Instruction-level parallelism

Execution Configuration

- GPUs are high latency, 100s of cycles per device memory request
- For good performance, you need to ensure there is enough parallelism to hide this latency
- Such parallelism can come from:
  - Thread-level parallelism
  - Instruction-level parallelism

Thread-Level Parallelism

- Execution configuration dictates number of threads per block
  - Limit on number of threads per block for each architecture
- Number of concurrent blocks on a multiprocessor limited by
  - Register use per thread
  - Shared memory use per thread block
  - Limit on number of threads per multiprocessor
- Occupancy
  - Ratio of actual to maximum number of concurrent threads per multiprocessor

Thread-Level Parallelism

```
attributes(global) subroutine copy(odata, idata)
  real :: odata(*), idata(*), tmp
  integer :: i
  i = (blockIdx%x-1)*blockDim%x + threadIdx%x
  tmp = idata(i)
  odata(i) = tmp
end subroutine copy
```

Thread-Level Parallelism

- Run on K20
  - Maximum of 2048 concurrent threads per multiprocessor
  - Maximum of 16 concurrent blocks per multiprocessor
  - Maximum of 1024 threads per block

Thread Block Size	Occupancy	Bandwidth (GB/s)
32	0.25	96
64	0.5	125
128	1	136
256	1	137
512	1	137
1024	1	133

Thread-Level Parallelism

- Mimic high resource use
  - Specify enough shared memory so only one thread block can reside on a multiprocessor at a time

```
call copy<<<grid, threadBlock, 0.9*smBytes>>>(b_d, a_d)
```

	No Shared Memory		Shared Memory	
Thread Block	Occupancy	Bandwidth	Occupancy	Bandwidth
32	0.25	96	0.016	8
64	0.5	125	0.031	15
128	1	136	0.063	29
256	1	137	0.125	53
512	1	137	0.25	91
1024	1	133	0.5	123

Instruction-Level Parallelism

- Have each thread process multiple elements

```
attributes(global) subroutine copy_ILP(odata, idata)
  real :: odata(*), idata(*), tmp(ILP)
  integer :: i,j

  i = (blockIdx%x-1)*blockDim%x*ILP + threadIdx%x
  do j = 1, ILP
    tmp(j) = idata(i+(j-1)*blockDim%x)
  enddo
  do j = 1, ILP
    odata(i+(j-1)*blockDim%x) = tmp(j)
  enddo
end subroutine copy_ILP
```

Instruction-Level Parallelism

	No Shared Memory		Shared Memory		
Thread Block Size	Occupancy	Bandwidth	Occupancy	Bandwidth No ILP	Bandwidth ILP = 4
32	0.25	96	0.016	8	26
64	0.5	125	0.031	15	50
128	1	136	0.063	29	90
256	1	137	0.125	53	125
512	1	137	0.25	91	140
1024	1	133	0.5	123	139

Calling CUBLAS from CUDA Fortran

- Module which defines interfaces to CUBLAS from CUDA Fortran
  - use cublas
- Interfaces in three forms
  - Overloaded BLAS interfaces that take device array arguments
    - call saxpy(n, a_d, x_d, incx, y_d, incy)
  - Legacy CUBLAS interfaces
    - call cublasSaxpy(n, a_d, x_d, incx, y_d, incy)
  - Multi-GPU version (CUDA 4.0) that utilizes a handle h
    - istat = cublasSaxpy_v2(h, n, a_d, x_d, incx, y_d, incy)
- Mixing the three forms is allowed

Calling CUBLAS from CUDA Fortran

```
program cublasTest
  use cublas
  implicit none

  real, allocatable :: a(:,:),b(:,:),c(:,:)
  real, device, allocatable :: a_d(:,:),b_d(:,:),c_d(:,:)
  integer :: k=4, m=4, n=4
  real :: alpha=1.0, beta=2.0, maxError

  allocate(a(m,k), b(k,n), c(m,n), a_d(m,k), b_d(k,n), c_d(m,n))

  a = 1; a_d = a
  b = 2; b_d = b
  c = 3; c_d = c

  call cublasSgemm('N','N',m,n,k,alpha,a_d,m,b_d,k,beta,c_d,m) ! or sgemm(..)

  c=c_d
  write(*,*) 'Maximum error: ', maxval(abs(c-14.0))

  deallocate (a,b,c,a_d,b_d,c_d)

end program cublasTest
```

CUDA Fortran Profiling and Debugging

- Some examples of profiling CUDA Fortran and the current status of CUDA Fortran debugging.

Performance Measurement

- CUDA Event management
- pgcollect / pgprof
- nvvp visual profiler; nvprof text profiler
- others (TAU, Vampir, ...)

Event Management

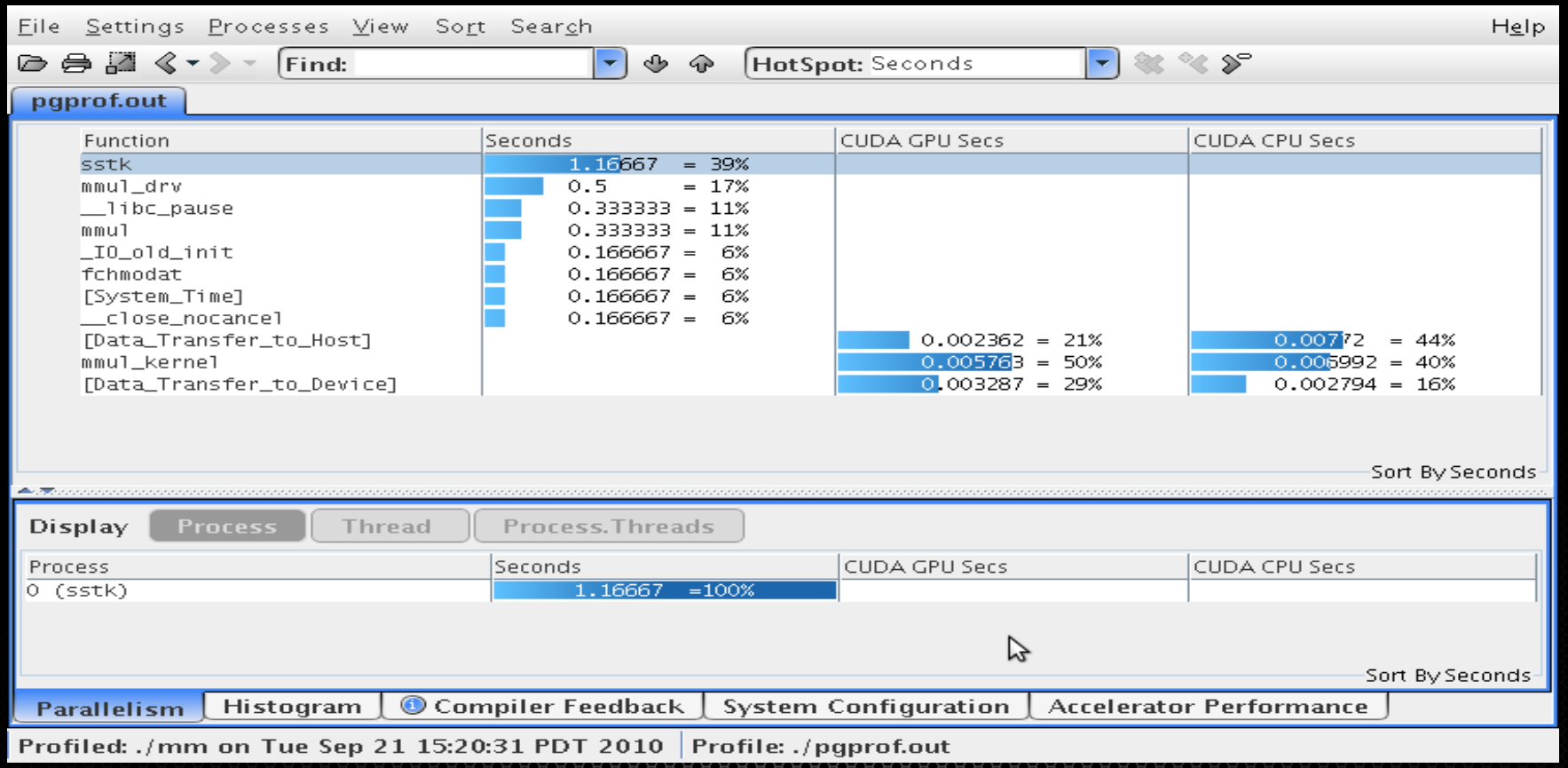
- integer cudaEventCreate(event)
  type(cudaEvent), intent(out) :: event
- integer cudaEventRecord(event, stream)
  type(cudaEvent), intent(in) :: event
  integer, intent(in) :: stream
- integer cudaEventSynchronize(event)
  type(cudaEvent), intent(in) :: event
- integer cudaEventElapsedTime(time,e1,e2)
  real, intent(out) :: time
  type(cudaEvent), intent(in) :: e1, e2
- integer cudaEventDestroy(event)
  type(cudaEvent), intent(in) :: event

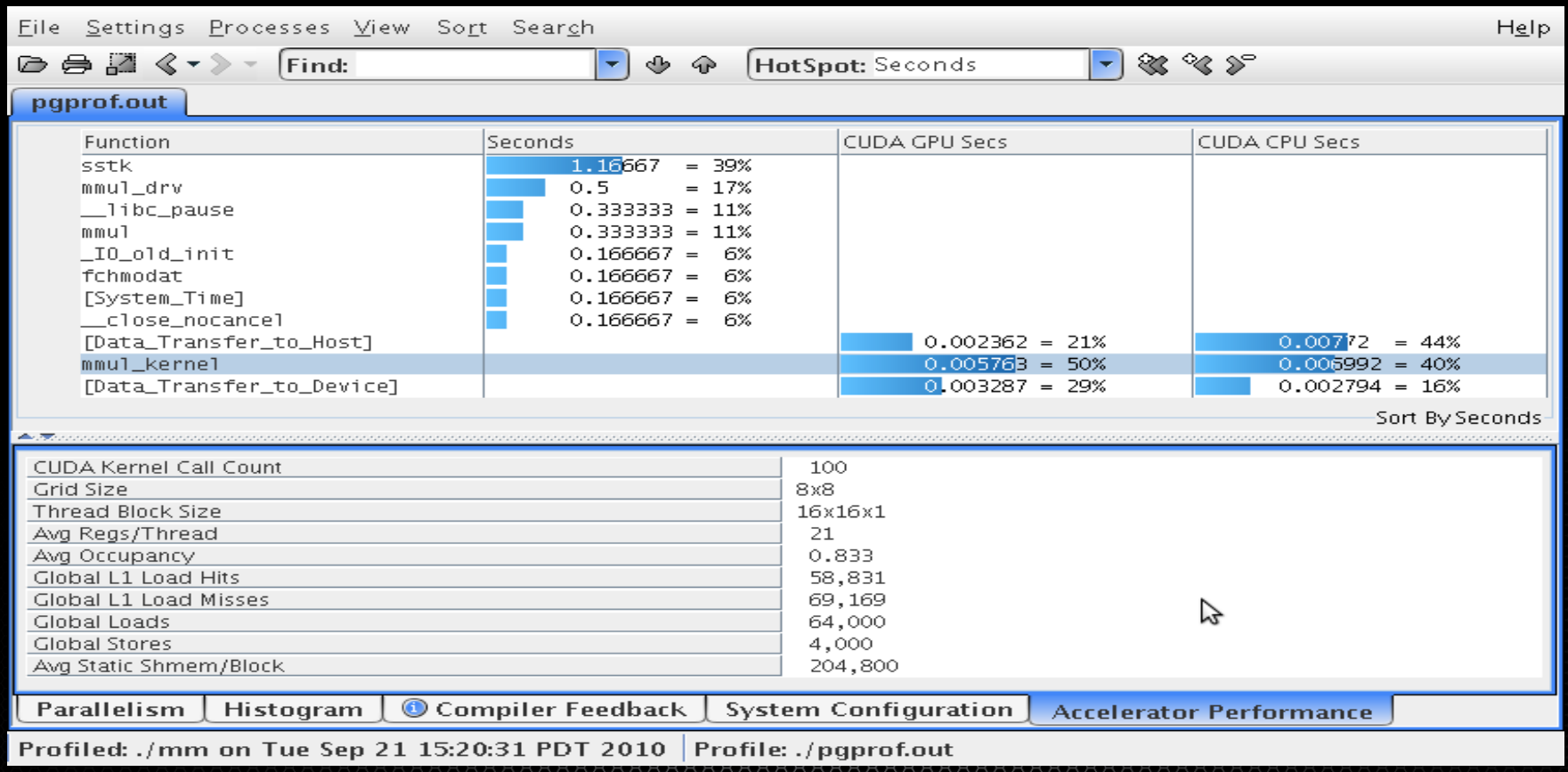
Events Example

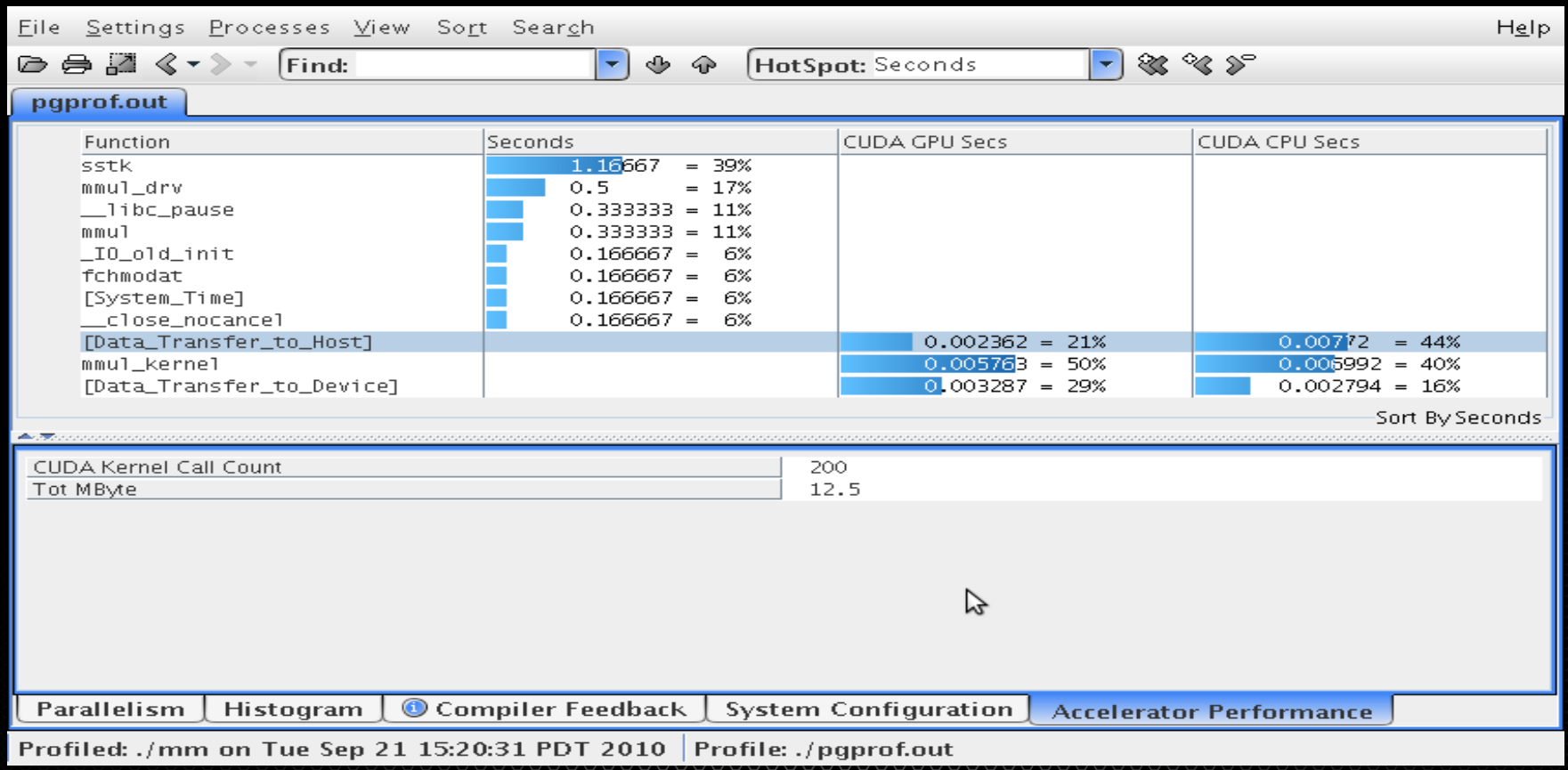
```
! timing experiment
time = 0.0
istat = cudaEventRecord(start, 0)
do j = 1, NREPS
  call sgemmNN_16x16<<<blocks, threads>>>(dA, dB, dC, m, N, k, alpha, beta)
end do
istat = cudaEventRecord(stop, 0)
istat = cudaDeviceSynchronize()
istat = cudaEventElapsedTime(time, start, stop)
time = time / (NREPS*1.0e3)
```

pgcollect / pgprof

- pgcollect [-cuda] a.out
  - Use -cuda to collect compute profile data
- pgprof [-exe a.out] [pgprof.out]


File Settings Processes View Sort Search Help
Find:
HotSpot: Seconds
pgprof.out
Function Seconds CUDA GPU Secs CUDA CPU Secs
sstk 1.16667 = 39%
mmul_drv 0.5 = 17%
__libc_pause 0.333333 = 11%
mmul 0.333333 = 11%
_IO_old_init 0.166667 = 6%
fchmodat 0.166667 = 6%
[System_Time] 0.166667 = 6%
__close_nocancel 0.166667 = 6%
[Data_Transfer_to_Host] 0.002362 = 21% 0.00772 = 44%
mmul_kernel 0.005763 = 50% 0.005992 = 40%
[Data_Transfer_to_Device] 0.003287 = 29% 0.002794 = 16%
Sort By Seconds
Display Process Thread Process.Threads
Process Seconds CUDA GPU Secs CUDA CPU Secs
0 (sstk) 1.16667 =100%
Sort By Seconds
Parallelism Histogram Compiler Feedback System Configuration Accelerator Performance
Profiled: ./mm on Tue Sep 21 15:20:31 PDT 2010
Profile: ./pgprof.out


File Settings Processes View Sort Search Help
Find:
HotSpot: Seconds
pgprof.out
Function
Seconds
CUDA GPU Secs
CUDA CPU Secs
sstk 1.16667 = 39%
mmul_drv 0.5 = 17%
__libc_pause 0.333333 = 11%
mmul 0.333333 = 11%
_IO_old_init 0.166667 = 6%
fchmodat 0.166667 = 6%
[System_Time] 0.166667 = 6%
__close_nocancel 0.166667 = 6%
[Data_Transfer_to_Host] 0.002362 = 21% 0.00772 = 44%
mmul_kernel 0.005763 = 50% 0.005992 = 40%
[Data_Transfer_to_Device] 0.003287 = 29% 0.002794 = 16%
Sort By Seconds
CUDA Kernel Call Count 100
Grid Size 8x8
Thread Block Size 16x16x1
Avg Regs/Thread 21
Avg Occupancy 0.833
Global L1 Load Hits 58,831
Global L1 Load Misses 69,169
Global Loads 64,000
Global Stores 4,000
Avg Static Shmem/Block 204,800
Parallelism
Histogram
Compiler Feedback
System Configuration
Accelerator Performance
Profiled: ./mm on Tue Sep 21 15:20:31 PDT 2010
Profile: ./pgprof.out


File Settings Processes View Sort Search Help
Find:
HotSpot: Seconds
pgprof.out
Function
Seconds
CUDA GPU Secs
CUDA CPU Secs
sstk 1.16667 = 39%
mmul_drv 0.5 = 17%
__libc_pause 0.333333 = 11%
mmul 0.333333 = 11%
_IO_old_init 0.166667 = 6%
fchmodat 0.166667 = 6%
[System_Time] 0.166667 = 6%
__close_nocancel 0.166667 = 6%
[Data_Transfer_to_Host] 0.002362 = 21% 0.00772 = 44%
mmul_kernel 0.005763 = 50% 0.005992 = 40%
[Data_Transfer_to_Device] 0.003287 = 29% 0.002794 = 16%
Sort By Seconds
CUDA Kernel Call Count 200
Tot MByte 12.5
Parallelism
Histogram
Compiler Feedback
System Configuration
Accelerator Performance
Profiled: ./mm on Tue Sep 21 15:20:31 PDT 2010
Profile: ./pgprof.out

nvprof a.out

```
==30736== NVPROF is profiling process 30736, command: a.out
==30736== Profiling application: a.out
==30736== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 38.89%  393.41us         4  98.352us  98.176us  98.496us  [CUDA memcpy HtoD]
 38.88%  393.25us         4  98.312us  97.760us  98.688us  [CUDA memcpy DtoH]
  6.05%  61.184us         1  61.184us  61.184us  61.184us  testasync_31_gpu
  5.92%  59.872us         1  59.872us  59.872us  59.872us  testasync_41_gpu
  5.88%  59.520us         1  59.520us  59.520us  59.520us  testasync_51_gpu
  4.38%  44.320us         1  44.320us  44.320us  44.320us  testasync_21_gpu
```

nvprof -print-gpu-trace -csv a.out

Start	Duration	Grid X	Grid Y	Grid Z	Block X	Block Y	Block Z	Registers Per Thread	Static SMem	Dynamic SMem	Size	Throughput	Device	Context	Stream	Name
ms	us								B	B	MB	GB/s				
249.4308	98.304										1	10.17253	Tesla K40c (0)	1	8	[CUDA memcpy HtoD]
250.1725	44.064	977	1	1	256	1	1	19	0	0			Tesla K40c (0)	1	8	testasync_21_gpu [22]
250.22	98.272										1	10.17584	Tesla K40c (0)	1	8	[CUDA memcpy DtoH]
269.0212	98.528										1	10.1494	Tesla K40c (0)	1	9	[CUDA memcpy HtoD]
269.6442	60.032	977	1	1	256	1	1	19	0	0			Tesla K40c (0)	1	9	testasync_31_gpu [36]
269.7065	98.272										1	10.17584	Tesla K40c (0)	1	9	[CUDA memcpy DtoH]
288.3552	98.304										1	10.17253	Tesla K40c (0)	1	10	[CUDA memcpy HtoD]

Performance Optimization

- Profile the CPU application
  - ensure you are accelerating the right part of your code
- Look at data movement
  - look for hidden data transfers
- Look at expensive kernels

Analyzing Kernels

- Launch configuration
  - block size > 1
  - grid size > 1
- Occupancy
  - registers per thread
- Memory operations
  - too many memory loads/stores (no caching)
  - stride-1 in vector index

Performance Tuning

- Performance Measurement
- Choose an appropriately parallel algorithm
- Optimize data movement between host and GPU
  - frequency, volume, regularity
- Optimize device memory accesses
  - strides, alignment
  - use shared memory, avoid bank conflicts
  - use constant memory
- Optimize kernel code
  - redundant code elimination
  - loop unrolling
  - Optimize compute intensity
    - unroll the parallel loop

Host-GPU Data Movement

- Avoid movement altogether
- Move outside of loops
- Better to move a whole array than subarray
- Update halo regions rather than whole array
  - use GPU to move halo region to contiguous area?
- Use streams, overlap data / compute
  - requires pinned host memory

Occupancy

- How many simultaneously active warps / maximum (maximum is 24, 32 (Tesla-10), 48 (Fermi), 64 (Kepler))
- Limits
  - threads per multiprocessor
  - threads per thread block – 512 (Tesla) or 1024 (Fermi/Kepler)
  - thread blocks per multiprocessor – 8 (Tesla/Fermi) or 16 (Kepler)
  - register usage – 8K / 16K / 32K / 64K registers per multiprocessor
    - each warp uses 32n, so 16Kreg = 512wreg
  - shared memory usage – 16KB (Tesla) or 48KB (Fermi, Kepler)
- Low occupancy often leads to low performance
- High occupancy does not guarantee high performance

Execution Configuration

- Execution configuration affects occupancy
- Want many threads per thread block
  - multiple of 32
  - 64, 128, 192, 256
- Want many many thread blocks

Divergence

- Scalar threads executing in SIMD mode

```
if( threadidx%x <= 10 )then
  foo = foo * 2
else
  foo = 0
endif
```

- Each path taken

```
do i = 1, threadidx%x
  a(threadidx%x,i) = 0
enddo
```

- Only matters within a warp

Divergence

- Pad arrays to multiples of block size

```
i = (blockidx%x-1)*64 + threadidx%x
if( i <= N ) A(i) = ...
```

Global Memory

- Stride-1 accesses
  - address is aligned to mod(threadidx%x,16)
  - threadidx%x and threadidx%x+1 access consecutive addresses
- Using shared memory as data cache
  - Redundant data access within a thread
  - Redundant data access across threads
  - Stride-1 data access within a thread

Redundant Access Within a GPU Thread

```
! threadidx%x from 1:64
! this thread block does 256 'i' iterations
ilo = (blockidx%x-1)*256
ihi = blockidx*256 - 1
...
do j = jlo, jhi
  do i = ilo+threadidx%x, ihi, 64
    A(i,j) = A(i,j) * B(i)
  enddo
enddo
```

Redundant Access Within a GPU Thread

```
real,shared :: BB(256)
...
do ii = 0, 255, 64
  BB(threadidx%x+ii) = B(ilo+ii)
enddo
call syncthreads()
do j = jlo, jhi
  do i = ilo+threadidx%x, ihi, 64
    A(i,j) = A(i,j) * BB(i-ilo)
  enddo
enddo
```

Redundant Access Across GPU Threads

```
! threadidx%x from 1:64
i = (blockidx%x-1)*64 + threadidx%x
...
do j = jlo, jhi
  A(i,j) = A(i,j) * B(j)
enddo
```

Redundant Access Across GPU Threads

```
real, shared :: BB(64)

i = (blockidx%x-1)*64 + threadidx%x
...
do jb = jlo, jhi, 64
  BB(threadidx%x) = B(jb+threadidx%x)
  call syncthreads()
  do j = jb, min(jhi,jb+63)
    A(i,j) = A(i,j) * BB(j-jb+1)
  enddo
  call syncthreads()
enddo
```

stride-1 access!

Shared Memory

- 16 memory banks
- Use `threadidx%x` in leading (stride-1) dimension
- Avoid stride of 16

Unroll the Parallel Loop

- If thread ‘j’ and ‘j+1’ share data, where
  - j is a parallel index
  - j is not the stride-1 index
- Unroll two or more iterations of ‘j’ into the kernel

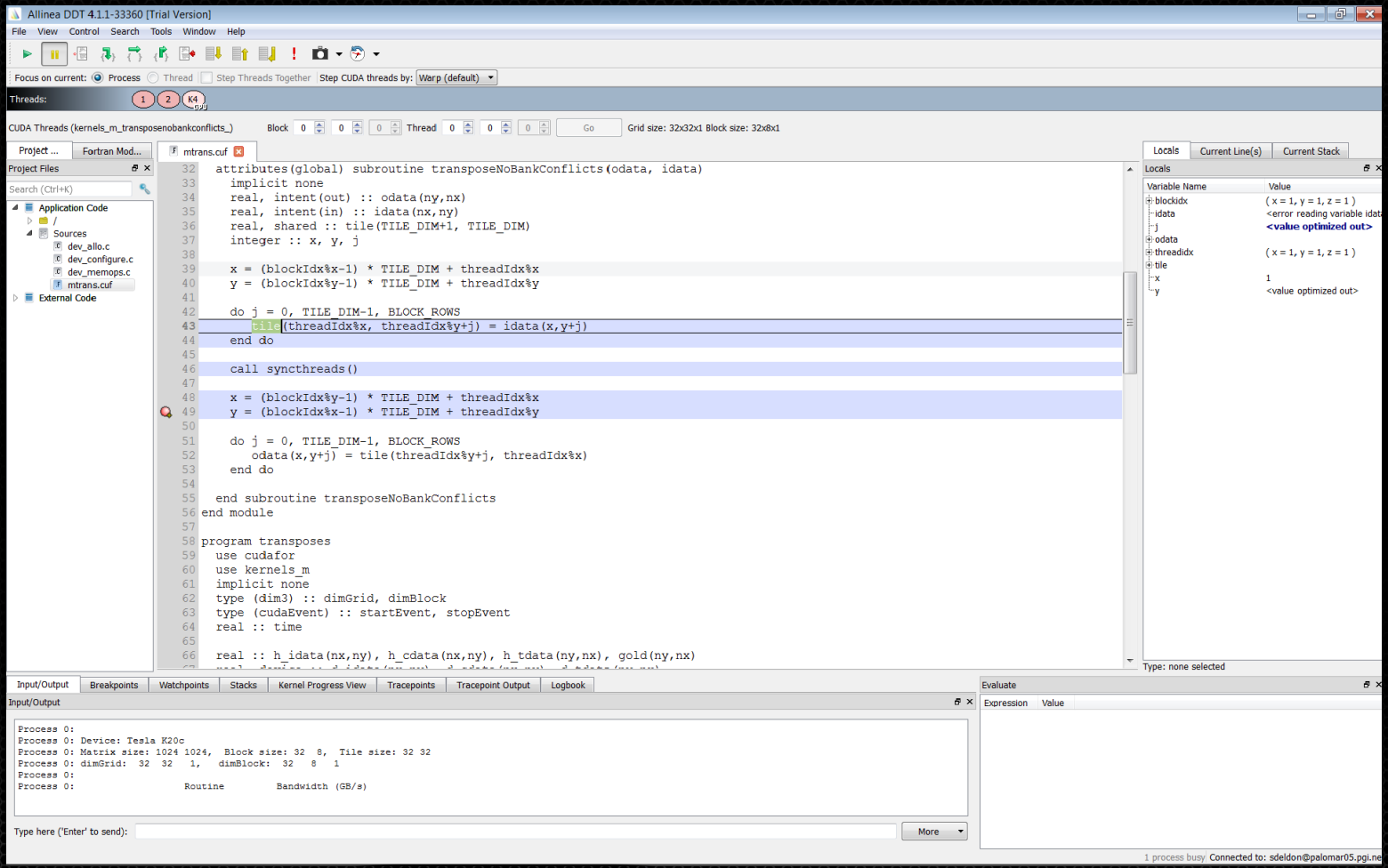
CUDA Fortran 2012/2013 Features

- PGI 2012 (CUDA 4.1 and 4.2)
  - Texture memory support
  - Support for double precision atomic add
- PGI 2013 (CUDA 5.0 and 5.5)
  - Support for dynamic parallelism
  - Relocatable device code, linking, device libraries
  - Full atomic datatype support
  - Kepler shuffle support

Latest Features in CUDA Fortran

- PGI 2014 (CUDA 6.0 and beyond)
  - DWARF generation, GPU-side debugging
  - Full shuffle datatype support
  - Overloaded reduction intrinsics
  - Support for Unified Memory
  - Improved interoperability with OpenACC

Debugging CUDA Fortran with Allinea DDT



Dwarf Generation Enables Tools

```
% pgf90 -g -O0 -Mcuda=cc35 saxpy.cuf
% cuda-memcheck ./a.out
========= CUDA-MEMCHECK
0: copyout Memcpy (host=0x68e9e0, dev=0x500200200, size=124)
FAILED: 4(unspecified launch failure)
========= Invalid __global__ read of size 4
=========     at 0x00000530 in /home/brentl/saxpy.cuf:10:m_saxpy_
=========     by thread (31,0,0) in block (0,0,0)

         9 ! if (i<=n) y(i) = alpha * x(i) + y(i)
        10   y(i) = alpha * x(i) + y(i)
```

Extending F90 Intrinsics to Device Data

- Evaluate sum() of device array and return result to host

```
! host code
sum_h = sum(a_d)
```

- Typical reduction uses two kernels
  - Calculate partial sum for threads within a block
  - Launch a one-block kernel to perform a final sum

Extending F90 Intrinsics to Device Data

- Small arrays transfer data to host and sum there
  - Limited parallelism for small array
  - Device to host transfer is mostly overhead
- Large arrays perform partial sum on device, final sum on host
- Cutoff for optimal performance depends on host system
  - Configurable via initialization routine

Stages of Partial Sum

- Reduce to grid of threads using grid-stride loop
- Reduce within warp using SHFL (shuffle) functions
- One thread in each warp writes the value to shared memory, all call syncthreads(), then warp 1 reads the values into registers and uses a set of SHFL operations again
- This results in gridDim%x partial sums which are moved to the host

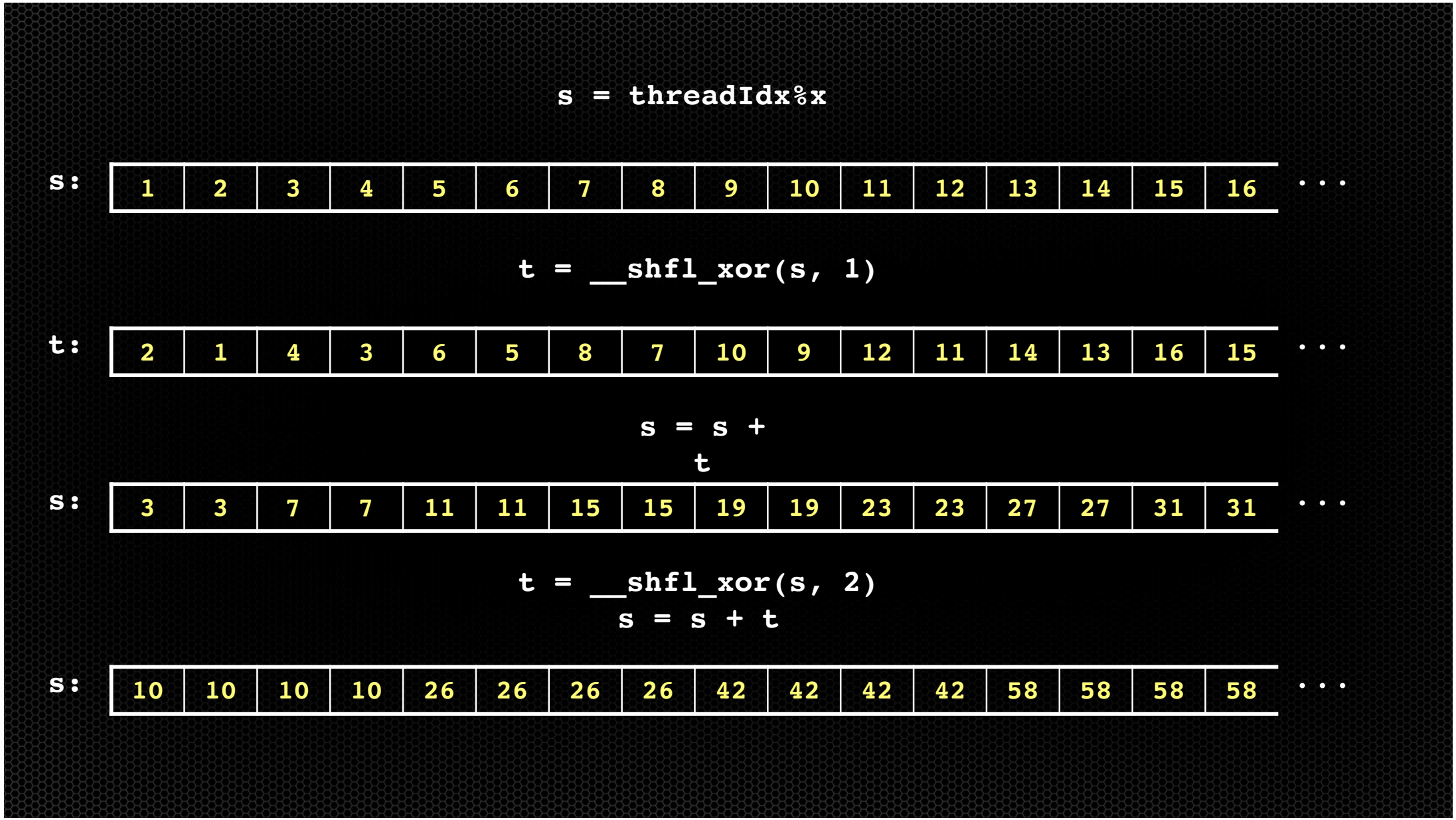
First Stage of Partial Sum

- Reduce to grid of threads using grid-stride loop

```
nGrid = gridDim%x*blockDim%x
s = 0.0
ig = (blockIdx%x-1)*blockDim%x + threadIdx%x
do i = ig, n, nGrid
   s = s + a(i)
end do
```

SHFL (shuffle) functions

- Intra-warp data exchange
- Threads can read other threads' registers
- No shared memory required, no synchronization barriers
- Requires compute capability >= 3.0


s = threadIdx%x
s:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
t = __shfl_xor(s, 1)
t:
2 1 4 3 6 5 8 7 10 9 12 11 14 13 16 15 ...
s = s + t
s:
3 3 7 7 11 11 15 15 19 19 23 23 27 27 31 31 ...
t = __shfl_xor(s, 2)
s = s + t
s:
10 10 10 10 26 26 26 26 42 42 42 42 58 58 58 58 ...

Reduce within warp using SHFL functions

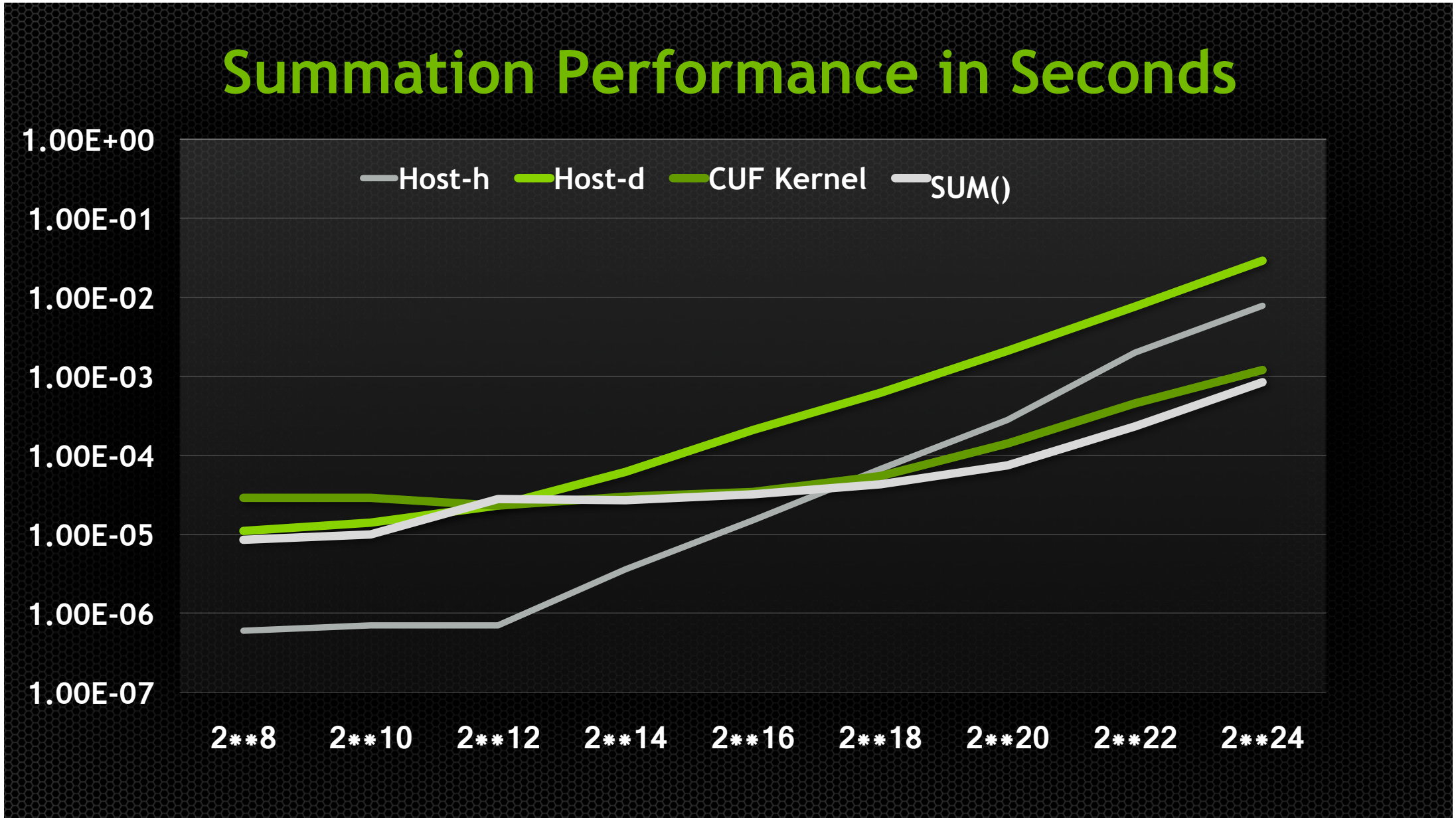
```
! reduce within warp
t = __shfl_xor(s, 1)
    s = s + t
t = __shfl_xor(s, 2)
    s = s + t
t = __shfl_xor(s, 4)
    s = s + t
t = __shfl_xor(s, 8)
    s = s + t
t = __shfl_xor(s, 16)
    s = s + t
```

Last Stage of Partial Sum

- Reduce to gridDim%x partial sums, one from each thread block

```
if (BlockDim%x == 32) then
   if (threadIdx%x == 1) p(blockIdx%x) = s
else
   warpID = (threadIdx%x-1)/32+1
   laneID = iand(threadIdx%x,31)
   if (laneID == 1) p_s(warpID) = s
   call syncthreads()
   if (warpID == 1) then   ! Reduce in warp 1
      width = blockDim%x/32
      s = p_s(threadIdx%x)
      i = 1
      do while (i < width)
         t = __shfl_xor(s, i, width)
         s = s + t
         i = i*2
      end do
      if (threadIdx%x == 1) p(blockIdx%x) = s
   end if
end if
```

Summation Performance in Seconds



CUDA Runtime API Routines

- Device management
- Thread management
- Memory management
- Event management
- `Fortran: use cudafor`

Device Management

- integer cudaGetDeviceCount(icount)
  integer, intent(out) :: icount

- integer cudaSetDevice(inum)
  integer, intent(in) :: inum

- integer cudaGetDevice(inum)
  integer, intent(out) inum

Device Management

```
integer cudaGetDeviceProperties(prop,inum)
  type(cudaDeviceProp),intent(out) :: prop
  integer, intent(in) :: inum
```

```
integer cudaChooseDevice( inum, prop )
  type(cudaDeviceProp),intent(in) :: prop
  integer, intent(out) :: inum
```

Device Management

- `prop%name  [character]`
- `prop%major`
- `prop%minor`
- `prop%totalGlobalMem`
- `prop%regsPerBlock`
- `prop%maxThreadsPerBlock`
- `prop%maxThreadDim(3)`
- `prop%maxGridSize(3)`
- `prop%clockRate`
- `prop%totalConstMem`

Device Management

- `prop%warpSize`
- `prop%textureAlignment`
- `prop%deviceOverlap`
- `prop%multiProcessorCount`
- `prop%kernelExecTimeoutEnabled`
- `prop%memPitch`
- `prop%integrated`
- `prop%canMapHostMemory`
- `prop%computeMode`

Stream Management

- integer cudaStreamCreate(stream)
  integer,intent(out) :: stream
- integer cudaStreamQuery(stream)
  integer,intent(in) :: stream
- integer cudaStreamSynchronize(stream)
  integer,intent(in) :: stream
- integer cudaStreamDestroy(stream)
  integer,intent(in) :: stream

Memory Management

- integer cudaMallocHost(ptr, size)
  type(c_ptr),intent(out) :: ptr
  integer,intent(in) :: size
- integer cudaFreeHost(ptr)
  type(c_ptr),intent(in) :: ptr
- integer cudaMalloc(ptr, count)
  <type>,device,allocatable,dimension(:)::&
  ptr
  integer,intent(in) :: count
- integer cudaFree(ptr)
  <type>,device,dimension(*) :: ptr

Memory Management

- ```
  integer cudaMemcpy(dst,src,count,dir)
   <type>,<device><,dimension> :: dst,src
   integer, intent(in) :: count, dir
  ```

- dir values:

  ```
  cudaMemcpyHostToHost
  cudaMemcpyHostToDevice
  cudaMemcpyDeviceToHost
  cudaMemcpyDeviceToDevice
  ```
  ```

Memory Management

- integer cudaMallocPitch(ptr, pitch, w, h)
  <type>,device,allocatable,dimension(:,:)&
    :: ptr
  integer,intent(out) :: pitch
  integer,intent(in) :: w,h
- integer cudaMemset(ptr, value, count)
  <type>,device<,dimension(*)> :: ptr
  <type> :: value
  integer :: count
- integer cudaMemset2D(ptr,pitch,value,w,h)
  <type>,device<,dimension> :: ptr

CUDA Fortran with Managed Data

```
program pgi14x
  use cudafor
  integer, parameter :: n = 100000
  integer, managed :: a(n), b(n), c(n)
  a = [(i,i=1,n)]
  b = 1
!$cuf kernel do <<< *, * >>>
  do i = 1, n
    c(i) = a(i) + b(i)
  end do
  if (sum(c).ne.n*(n+1)/2+n) then
    print *,c(1),c(n)
  end if
end
```

What is CUDA Managed Data?

```
real, managed, allocatable, dimension(:,:) ::
         A,B,C

. . .

allocate (A(N,M), B(M,L), C(N,L))
. . .


call mm_kernel <<<dim3(N/16,M/16),dim3(16,16)>>>
              ( A, B, C, N, M, L)

deallocate ( A, B, C )

. . .
```

Host Code

```
attributes(global) subroutine mm_kernel
                ( A, B, C, N, M, L )
real :: A(N,M), B(M,L), C(N,L), Cij
integer, value :: N, M, L
integer :: i, j, kb, k, tx, ty
real, shared :: Asub(16,16),Bsub(16,16)
tx = threadidx%x
ty = threadidx%y
i = blockidx%x * 16 + tx
j = blockidx%y * 16 + ty
Cij = 0.0
do kb = 1, M, 16
   Asub(tx,ty) = A(i,kb+tx-1)
   Bsub(tx,ty) = B(kb+ty-1,j)
   call syncthreads()
   do k = 1,16
      Cij = Cij + Asub(tx,k) * Bsub(k,ty)
   enddo
   call syncthreads()
enddo
C(i,j) = Cij
end subroutine mmul_kernel
```

Device Code

Managed Data enables Derived Type Usage

```
type tCM
  integer, allocatable, device :: fine(:)
  real, allocatable, device :: mat_matrix(:,:,:)
  real, allocatable, device :: src_matrix(:,:,:)
end type
type(tCM), allocatable, managed :: cm_list(:)
```

Array of derived type can be allocated on the host and used on the device

CUDA Fortran/OpenACC Interoperability

- Calling CUDA Fortran kernels from accelerator data regions
- Using CUDA Fortran device data in compute regions

```
!$acc data create(x) copy(y)
a = 3.0
!$acc kernels loop
do i = 1, n
  x(i) = x_dev(i) + 1.0
  y(i) = y(i) + real(i)
end do
call mysaxpy <<<block, thread>>> (n, a, x, y)
!$acc end data
```

- Support of arguments with the device attribute in our OpenACC 2.0 Runtime Library Routines

Calling CUDA Fortran Subroutines From OpenACC Programs

```
use cublas

!$acc data region copyin(x)

! some compute regions . . .

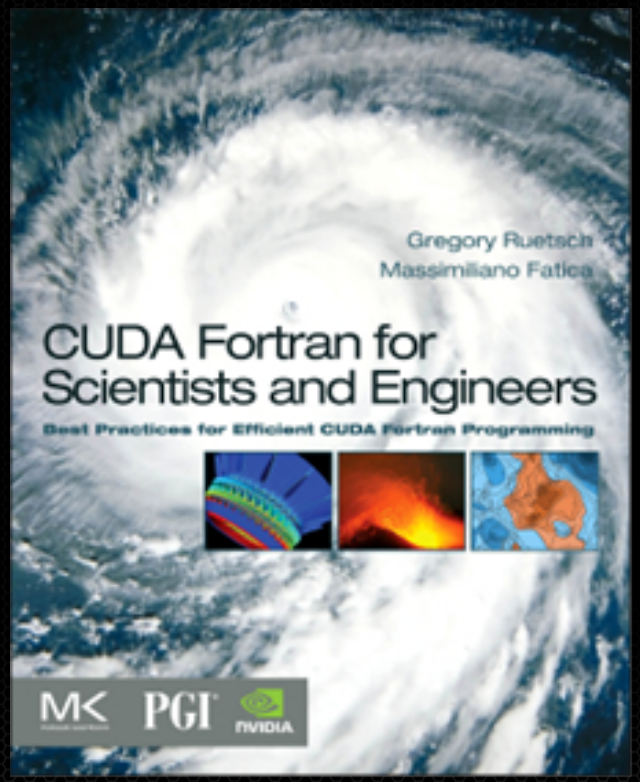
k = isamax(N,x,1)

! maybe some other compute regions. . .

!$acc end data region
```

- You can call CUDA routines by creating explicit interfaces to host-resident data and device-resident data specific functions for a generic call

CUDA Fortran Literature



- Book Released in 2013
- PGInsider articles at http://www.pgroup.com
- CUDA Fortran Reference Manual