# Card Sizes

| | Tesla C870 | Tesla C1060 | Tesla C2050 | Tesla K10 | Tesla K20 |
|---|---|---|---|---|---|
| Compute Capability | 1.0 | 1.3 | 2.0 | 3.0 | 3.5 |
| Max Threads per Thread Block | 512 | 512 | 1024 | 1024 | 1024 |
| Max Threads per SM | 768 | 1024 | 1536 | 2048 | 2048 |
| Max Thread Blocks per SM | 8 | 8 | 8 | 16 | 16 |

Tesla K40:  2880 processors; 12 GB memory

# Data bigger than grid

Maximum grid sizes

Compute capability 1.0, 1D and 2D grids supported

Compute capability 2, 3, 3D grids too.

Grid sizes: 65,535 x 65,535 x 65,535

$2^{31}$-1 x 65,535 x 65,535 (CUDA Compute Capability 3.0)

For large data sets, you can use many copies of the grid.

(see gridStride example later)

# ThreadId for Various Grids

1D grid of 1D blocks

```
__device__ int getGlobalIdx_1D_1D()
{
    return blockIdx.x *blockDim.x + threadIdx.x;
}
```

1D grid of 2D blocks

```
__device__ int getGlobalIdx_1D_2D()
{
    return blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x +
threadIdx.x;
}
```

1D grid of 3D blocks

```
__device__ int getGlobalIdx_1D_3D()
{
    return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
    + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
}
```

Martin Peniak

2D grid of 1D blocks

```
__device__ int getGlobalIdx_2D_1D()
{
    int blockId   = blockIdx.y * gridDim.x + blockIdx.x;
    int threadId = blockId * blockDim.x + threadIdx.x;
    return threadId;
}
```

2D grid of 2D blocks

```
 __device__ int getGlobalIdx_2D_2D()
{
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;
    int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) +
    threadIdx.x;
    return threadId;
}
```

2D grid of 3D blocks

```
__device__ int getGlobalIdx_2D_3D()
{
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;
    int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                + (threadIdx.z * (blockDim.x * blockDim.y))
                + (threadIdx.y * blockDim.x)
                + threadIdx.x;
    return threadId;
}
```

3D grid of 1D blocks

```
__device__ int getGlobalIdx_3D_1D()
{
        int blockId = blockIdx.x + blockIdx.y * gridDim.x
                        + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * blockDim.x + threadIdx.x;
        return threadId;
}
```

3D grid of 2D blocks

```
__device__ int getGlobalIdx_3D_2D()
{
        int blockId = blockIdx.x + blockIdx.y * gridDim.x
                        + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * (blockDim.x * blockDim.y)
                         + (threadIdx.y * blockDim.x)
                         + threadIdx.x;
        return threadId;
}
```

3D grid of 3D blocks

```
__device__ int getGlobalIdx_3D_3D()
{
        int blockId = blockIdx.x + blockIdx.y * gridDim.x
                       + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                         + (threadIdx.z * (blockDim.x * blockDim.y))
                         + (threadIdx.y * blockDim.x)
                         + threadIdx.x;
        return threadId;
}
```

# Thread Blocks

thread blocks are required to execute independently in arbitrary order

threads in same block (up to 1024) can use the same shared memory

synchronization in a block is achieved with __syncthreads(), which creates a barrier in the block

# Intrinsic CUDA math functions

Single and double precision:  basically same as C++

sqrtf(), cbrtf(x),  expf(x), logf(x), sinf(x), cosf(x), tanf(x), sinh(x), cosh(x), tanh(x)
erfcf(x), j0f(x), j1f(x), y0f(x)

Also have device only functions accessed by

nvcc xxxx.cu **-use_fast_math**

Faster, but less accurate

| Operator/Function | Device Function |
|---|---|
| `x/y` | `__fdividef(x,y)` |
| `sinf(x)` | `__sinf(x)` |
| `cosf(x)` | `__cosf(x)` |
| `tanf(x)` | `__tanf(x)` |
| `sincosf(x,sptr,cptr)` | `__sincosf(x,sptr,cptr)` |
| `logf(x)` | `__logf(x)` |
| `log2f(x)` | `__log2f(x)` |
| `log10f(x)` | `__log10f(x)` |
| `expf(x)` | `__expf(x)` |
| `exp10f(x)` | `__exp10f(x)` |
| `powf(x,y)` | `__powf(x,y)` |

# CUDA Libraries

CUDA Math Library

CUBLAS: BLAS for CUDA
#include <cublas.h>

CUFFT:  1D, 2D, 3D FFT
#include <cufft.h>
http://devblogs.nvidia.com/parallelforall/cudacasts-episode-8-accelerate-fftw-apps-cufft-55/

CUSPARSE: sparse matrix routines

CURAND: random numbers

NPP:  Nvidia Performance Primitives: signal and image processing

CUSP: Open Source Algorithms for sparse linear algebra: developer.nvidia.com/cusp

CULA:  Linear Algebra (lapack)
http://www.culatools.com/user/signup.php

MAGMA:  http://icl.cs.utk.edu/magma/index.html

https://developer.nvidia.com/gpu-accelerated-libraries

# Hour Exam

Oct 20,  Kreiger 307

Know C++, OpenMP, CUDA up to Oct 14
In-class exam.  Will be using the computers.

# Homework

Convert your openMP heat conduction problem into a CUDA solved problem.
Compare timings between CUDA and your openMP implementation.

# myPiGPU.cu

# Tools to check CUDA programs

nvidia-smi  (on command line)

cuda-memcheck  (on comand line)

command-line CUDA profiler  (logger)

nvprof  (on command line)

cuda-gdb (Linux and mac)

# nvidia-smi

http://devblogs.nvidia.com/parallelforall/cudacasts-episode-7-nvidia-smi-accounting/
https://developer.nvidia.com/nvidia-system-management-interface

System Management Interface (for Linux and Windows)

Information about the GPUs:  next page

```
==============NVSMI LOG==============

Timestamp                           : Fri Jun 22 11:32:26 2012

Driver Version                      : 295.45

Attached GPUs                       : 1

GPU 0000:01:00.0
    Product Name                    : Tesla C2070
    Display Mode                    : Enabled
    Persistence Mode                : Disabled
    Driver Model
        Current                     : N/A
        Pending                     : N/A
    Serial Number                   : 0322211066758
    GPU UUID                        : GPU-ed89e2d3-e674-1a30-8dd0-eaee6789ee19
    VBIOS Version                   : 70.00.44.00.02
    Inforom Version
        OEM Object                  : 1.0
        ECC Object                  : 1.0
        Power Management Object     : 1.0
    PCI
        Bus                         : 0x01
        Device                      : 0x00
        Domain                      : 0x0000
        Device Id                   : 0x06D110DE
        Bus Id                      : 0000:01:00.0
        Sub System Id               : 0x077210DE
        GPU Link Info
            PCIe Generation
                Max                 : 2
                Current             : 2
            Link Width
                Max                 : 16x
                Current             : 16x
    Fan Speed                       : 30 %
    Performance State               : P0
    Memory Usage
        Total                       : 5375 MB
        Used                        : 9 MB
        Free                        : 5365 MB
    Compute Mode                    : Default
    Utilization
        Gpu                         : 0 %
        Memory                      : 0 %
    Ecc Mode
        Current                     : Enabled
        Pending                     : Enabled
    ECC Errors
        Volatile
            Single Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Total               : 0
            Double Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Total               : 0
        Aggregate
            Single Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Total               : 0
            Double Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Total               : 0
    Temperature
        Gpu                         : 52 C
    Power Readings
        Power Management            : N/A
        Power Draw                  : N/A
        Power Limit                 : N/A
    Clocks
        Graphics                    : 573 MHz
        SM                          : 1147 MHz
        Memory                      : 1494 MHz
    Max Clocks
        Graphics                    : 573 MHz
        SM                          : 1147 MHz
        Memory                      : 1494 MHz
    Compute Processes               : None
```

# cuda-memcheck

Usage: **cuda-memcheck program_name**

Monitors hundreds of thousands of threads running concurrently on each GPU

Reports detailed information about global memory access errors such as out of bounds accesses and misaligned memory accesses, including instruction offet in CUDA Function/Kernel name or source file and line number

Reports runtime executions errors such as device or user stack overflows and Illegal instructions

Misaligned or Out of range accesses to shared and local memory

Reports detailed information about potential race conditions between accesses to shared memory. Including severity information about hazard, block and thread index, CUDA function/kernel name & instruction offset, source file and line number and data values being written

Displays stack back traces on host and device for errors

Reports if any CUDA API calls returned errors

**http://docs.nvidia.com/cuda/cuda-memcheck/index.html**

# Command-Line CUDA Profiler

http://docs.nvidia.com/cuda/profiler-users-guide/index.html#compute-command-line-profiler-overview
>tcsh
>setenv COMPUTE_PROFILE 1

enables the command-line CUDA profiler

> myPiGPU 10000  creates a text log file: cuda_profile_0.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 650M
# CUDA_CONTEXT 1
method,gputime,cputime,occupancy
method=[ _Z4termPdi ] gputime=[ 56.992 ] cputime=[ 41.308 ] occupancy=[ 1.000 ]
method=[ memcpyDtoH ] gputime=[ 54.464 ] cputime=[ 2048.779 ]

First method line:  establishes the DEVICE constant:
double PI25D= 3.14159265358979323846264 3

Next:
cudaMalloc(&d_part_sum, N*sizeof(double));

gputime:  microseconds; cputime: milliseconds

# nvprof

Simple profiler:  gives summary of kernel calls and memcpy; tells where time is spent.

http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview
NOTE:  nvprof and command-line profiler are mutually exclusive:
setenv COMPUTE_PROFILE 0

```
Number of intervals: 10000
interval size = 0.0001
CUDAMemory: 80000
==16115== NVPROF is profiling process 16115, command: a.out 10000
 blocks: 79
time for parallel computation: 5.4e-05
pi = 3.141792649
error = -0.0002
time for total computation: 0.000284
==16115== Profiling application: a.out 10000
==16115== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 56.49%  51.808us         1  51.808us  51.808us  51.808us  [CUDA memcpy DtoH]
 43.51%  39.904us         1  39.904us  39.904us  39.904us  term(double*, int)
```

# cuda-gdb

Based on GNU gdb,  gnu debugger

Compile program with -g
 > nvcc -g  -G myPiGPU.cu -o myPiGPU   (N defined in problem)

> cuda-gdb myPiGPU
(cuda-gdb) break main
(cuda-gdb) break myPiGPU.cu:83
(cuda-gdb) print N
(cuda-gdb) run
(cuda-gdb) continue
(cuda-gdb) print blockIdx
(cuda-gdb) print gridDim
(cuda-gdb) delete b    (delete breakpoints)

http://developer.download.nvidia.com/compute/cuda/2_1/cudagdb/CUDA_GDB_User_Manual.pdf

# CUDA Occupancy Calculator

A2

Enter a name for a cell range, or select a named range from the list

## CUDA Occupancy Calculator

**Click Here for detailed instructions on how to use this occupancy calculator.**

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

**Just follow steps 1, 2, and 3 below! (or click here for help)**

| | | |
|---|---|---|
| 1.) Select Compute Capability (click): | 3.0 | (Help) |
| 1.b) Select Shared Memory Size Config (bytes) | 32768 | |
| **2.) Enter your resource usage:** | | |
| Threads Per Block | 128 | (Help) |
| Registers Per Thread | 32 | |
| Shared Memory Per Block (bytes) | 4096 | |

(Don't edit anything below this line)

| | |
|---|---|
| **3.) GPU Occupancy Data is displayed here and in the graphs:** | |
| **Active Threads per Multiprocessor** | **1024** |
| **Active Warps per Multiprocessor** | **32** |
| **Active Thread Blocks per Multiprocessor** | **8** |
| **Occupancy of each Multiprocessor** | **50%** |

| **Physical Limits for GPU Compute Capability:** | **3.0** |
|---|---|
| Threads per Warp | 32 |
| Warps per Multiprocessor | 64 |
| Threads per Multiprocessor | 2048 |
| Thread Blocks per Multiprocessor | 16 |
| Total # of 32-bit registers per Multiprocessor | 65536 |
| Register allocation unit size | 256 |
| Register allocation granularity | warp |
| Registers per Thread | 63 |
| Shared Memory per Multiprocessor (bytes) | 32768 |
| Shared Memory Allocation unit size | 256 |
| Warp allocation granularity | 4 |
| Maximum Thread Block Size | 1024 |

| **Allocated Resources** | Per Block | Limit Per SM | = Allocatable Blocks Per SM |
|---|---|---|---|
| Warps (Threads Per Block / Threads Per Warp) | 4 | 64 | 16 |
| Registers (Warp limit per SM due to per-warp reg count) | 4 | 64 | 16 |
| Shared Memory (Bytes) | 4096 | 32768 | 8 |

Note: SM is an abbreviation for (Streaming) Multiprocessor

| **Maximum Thread Blocks Per Multiprocessor** | Blocks/SM | * Warps/Block | = Warps/SM |
|---|---|---|---|
| Limited by Max Warps or Max Blocks per Multiprocessor | 16 | | |
| Limited by Registers per Multiprocessor | 16 | | |
| **Limited by Shared Memory per Multiprocessor** | **8** | **4** | **32** |

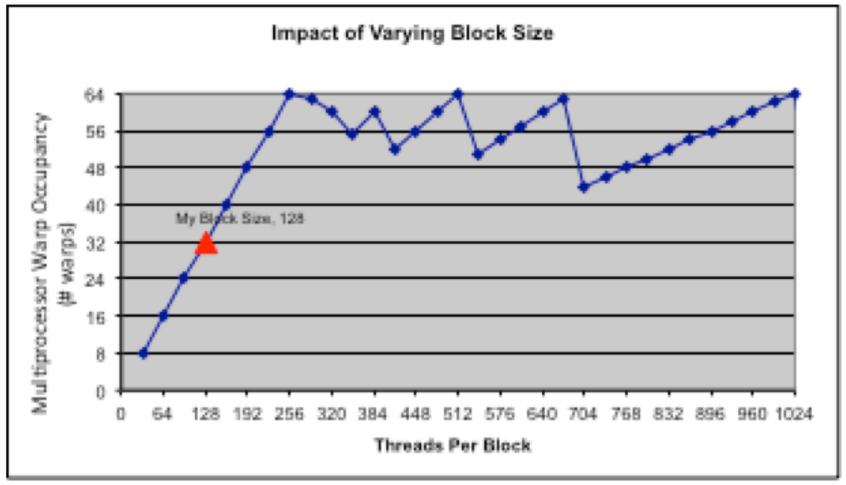
Note: Occupancy limiter is shown in orange

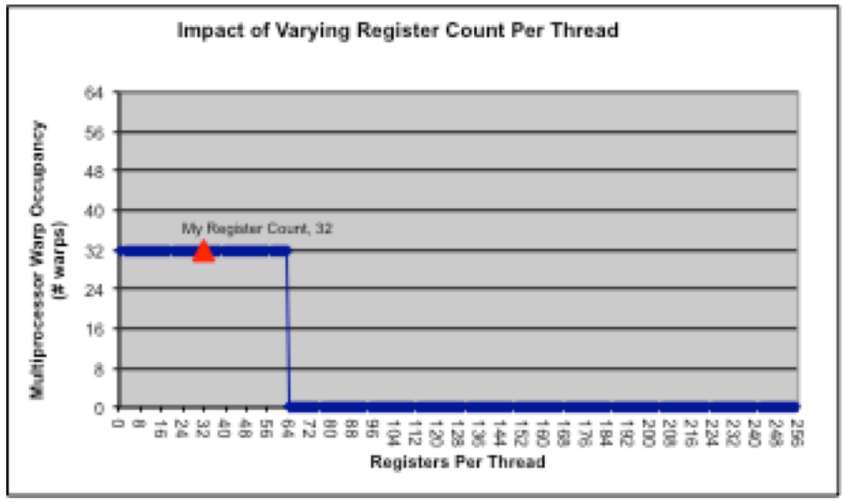
**Physical Max Warps/SM = 64**

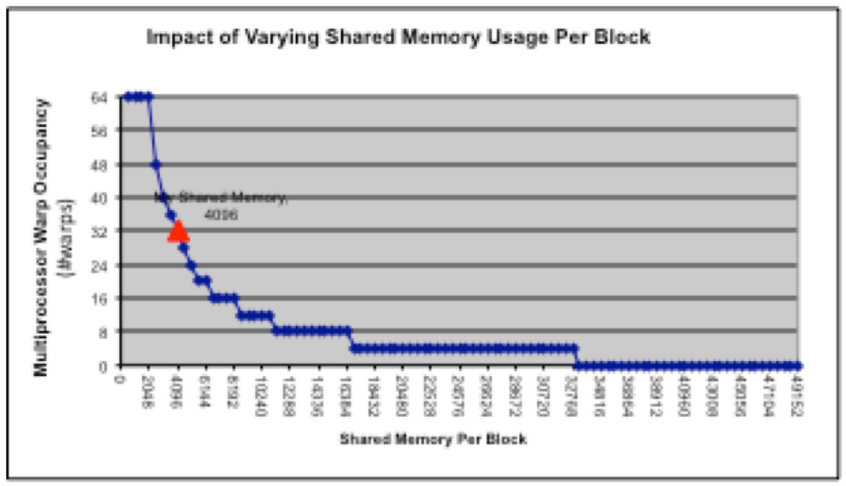
**Occupancy = 32 / 64 = 50%**

| | |
|---|---|
| CUDA Occupancy Calculator | |
| Version: | 5.1 |
| Copyright and License | |

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



# CUDA Timing

**cudaEvent_t start, stop;**
**cudaEventCreate(&start);**
**cudaEventCreate(&stop);**

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

**cudaEventRecord(start);**
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
**cudaEventRecord(stop);**

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

**cudaEventSynchronize(stop);** //stop CPU until event recorded.
**float milliseconds = 0;**
**cudaEventElapsedTime(&milliseconds, start, stop);** //milliseconds has the
//time for the saxpy kernel to execute

then print milliseconds

# Optimizing Data Transfers

PCI-e bus speed:  8 GB/s

GPU bus:  From deviceQueryDrv, memory clock rate: 2508 MHz
and bus width: 128 bits

$2508 * 10^6 * 128 \text{ bits} * 1 \text{ byte}/8 \text{ bits} * 2 / 10^9 = 80 \text{ GB/s}$

2 is the double data rate of the DDR memory; $10^9$ converts to GB/s

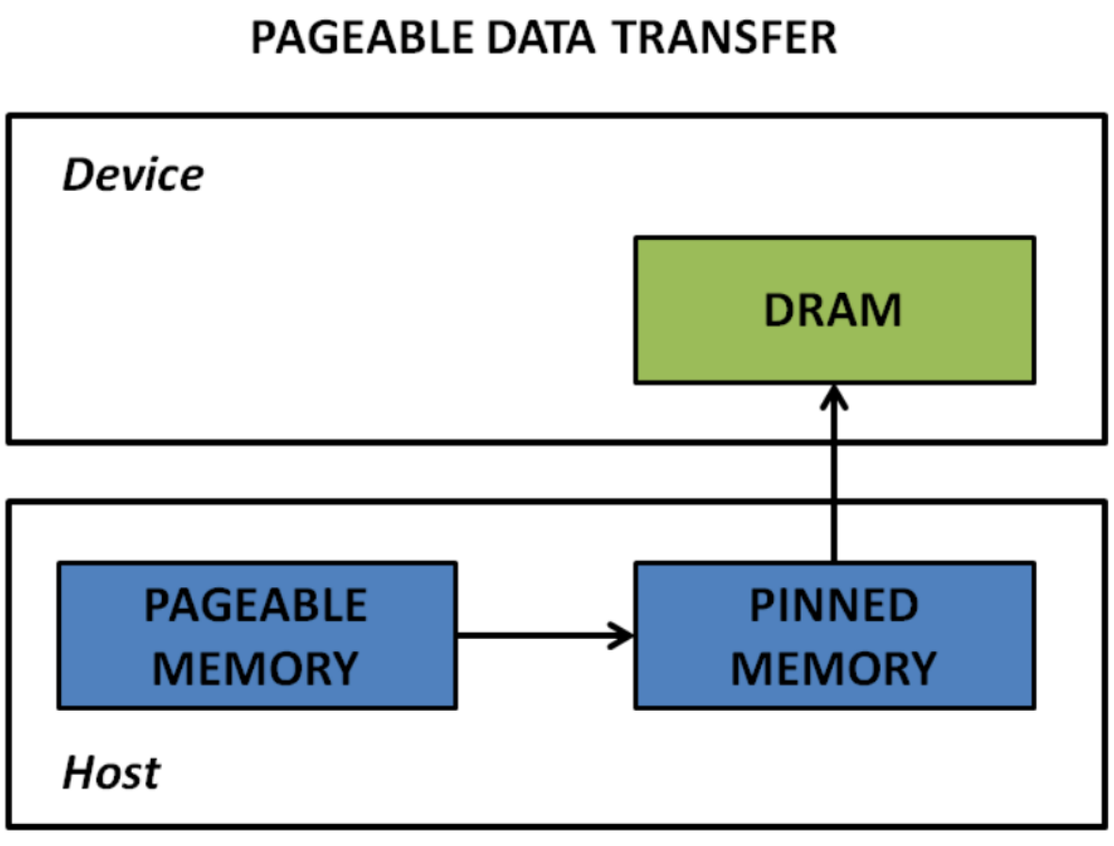
Using pinned-memory

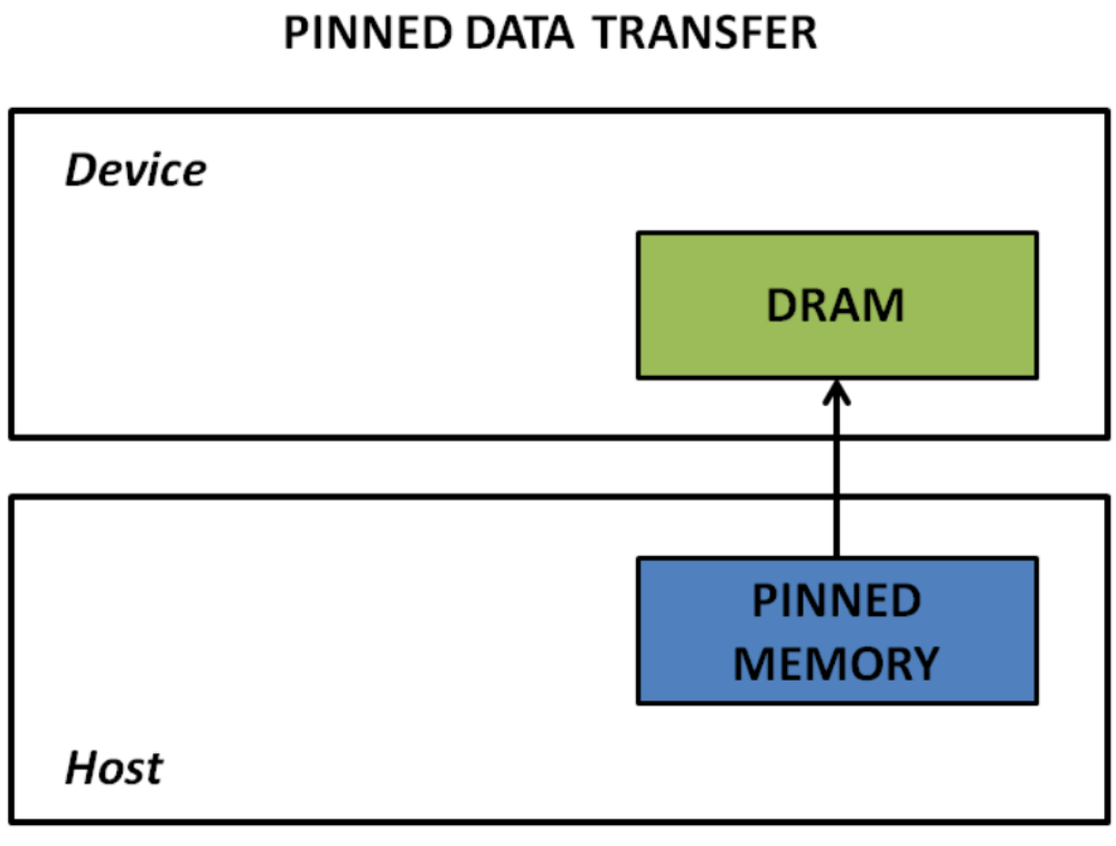
# Page-locked (Pinned) Memory

CPU can access larger data sets than their memory can hold by implementing a virtual memory system (non-locked memory). Pages of memory can be temporarily saved on disk and swapped in and out when needed.

When data is needed by GPU, it is copied to a page-locked pinned memory buffer and passed to Direct Memory Access. The cost is the time to copy the data to pinned memory, the data transfer, and the deletion of page-locked memory.

There is usually enough space on the CPU to use page-locked memory; then the DMA is made to GPU without involving the CPU.



# SAXPY again: Grid Stride

Old kernel:  one thread per element; single grid covers all elements

```
saxpy<<<4096,256>>>(1<<20, 2.0, x, y);
//each thread handles one of the n additions
```

---

New kernel:  grid stride

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
    for( int i = blockIdx.x * blockDim.x + threadIdx.x; i<n; i+=blockDim.x*gridDim.x)
//this loop lets card know that the number of threads
//can vastly exceed the size of the number of threads in grid
    {
        y[i] = a * x[i] + y[i];
    }
}  //Using grid stride; calculate over different smaller grids (64)  of 16384 threads

 int numSMs = 2;
 saxpy<<<numSMs*32,256>>>(n, 2.0, d_x, d_y);
```

For this 1D grid, the number of threads in grid is blockDim.x*GridDim.x, which is
  256 * 64 means 64 different grids to get 1<<20 (1,048,576)  threads

# Grid Stride is faster

Thread reuse saves some overhead of thread creation

saxpy<<1,1>>(n, 2.0, x, y)  is the serial version.  Very slow, but debug friendly
1 block, 1 thread per block

# Shared Memory

Very fast, accessible to all threads in block

Example:  threads in same block can access data obtained by global memory by other threads in the block

Need to be careful of race conditions.  These occur because threads are grouped by 32 thread bundles called warps for execution.  If thread a and b read data from global memory and save as shared memory.  Then thread a want to read b's element in shared memory.  If a and b are in different warps, then b may not be done writing before a wants to read it.

__syncthreads();  //to provide a barrier in block

Using Shared Memory with static and dynamic arrays.

```
//a 1-D array of 0, 1, 2, ..., 63 is to be reversed: 63, 62..

#include <stdio.h>

__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];     //copy array d into shared memory array s
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];  //need the extern keyword as memory dynamically alloc.
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

```
int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];
   for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;   // r[] is the correct answer array
    d[i] = 0;
  }
 int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));

  // run version with static shared memory
```

```
cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
staticReverse<<<1,n>>>(d_d, n);
cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
for (int i = 0; i < n; i++)
  if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)\n", i, i, d[i], r[i]);
```

```
// run dynamic shared memory version

 cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);

 dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);  //third part of <<<>>> is shmem size

 cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);

 for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)\n", i, i, d[i], r[i]);
}
```

# Shared Memory

cuda-memcheck --tool racecheck  program_name
http://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool

The racecheck tool identifies three types of canonical hazards in a program. These are :

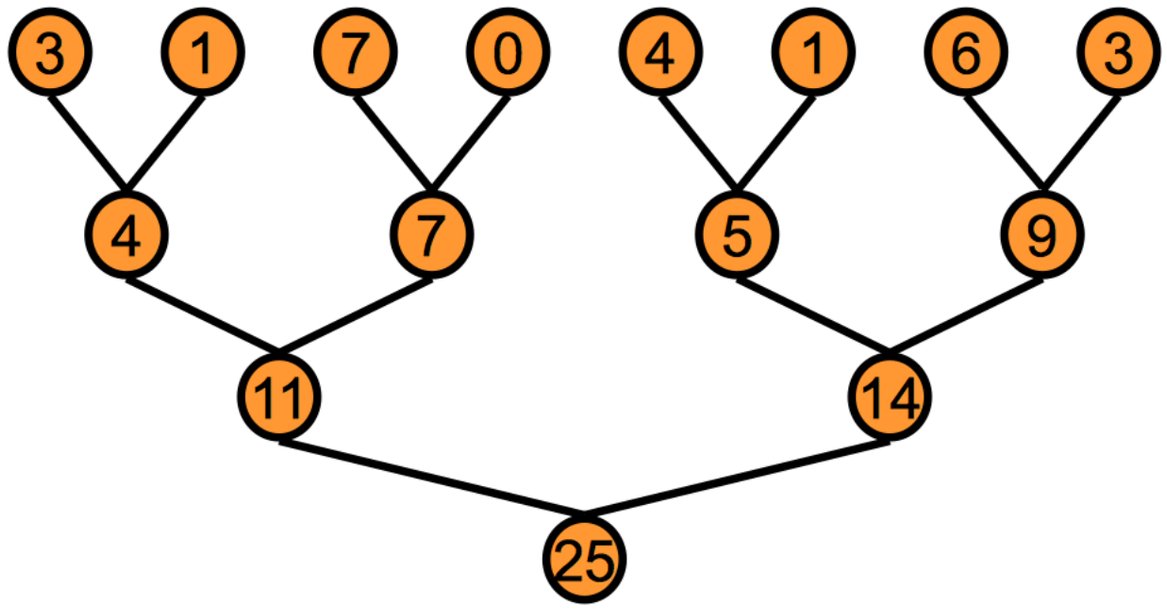
- Write-After-Write (WAW) hazards This hazard occurs when two threads attempt to write data to the same memory location. The resulting value in that location depends on the relative order of the two accesses.

- Read-After-Write (RAW) hazards This hazard occurs when two threads access the same memory location, with one thread performing a read and another a write. In this case, the writing thread is ordered before the reading thread and the value returned to the reading thread is not the original value at the memory location.

- Write-After-Read (WAR) hazards This hazard occurs when two threads access the same memory location, with one thread performing a read and the other a write. In this case, the reading thread reads the value before the writing thread commits it.

# Parallel Reduction

For very large array of numbers, need many thread blocks; each block does a portion of the array:




Need synchronization across all the thread blocks; but NO global synchronization (Expensive to build and would only allow running as many blocks as can fit on the card, rather than many more sequentially.)

One solution:  kernel decomposition.

Mark Harris, Nvidia

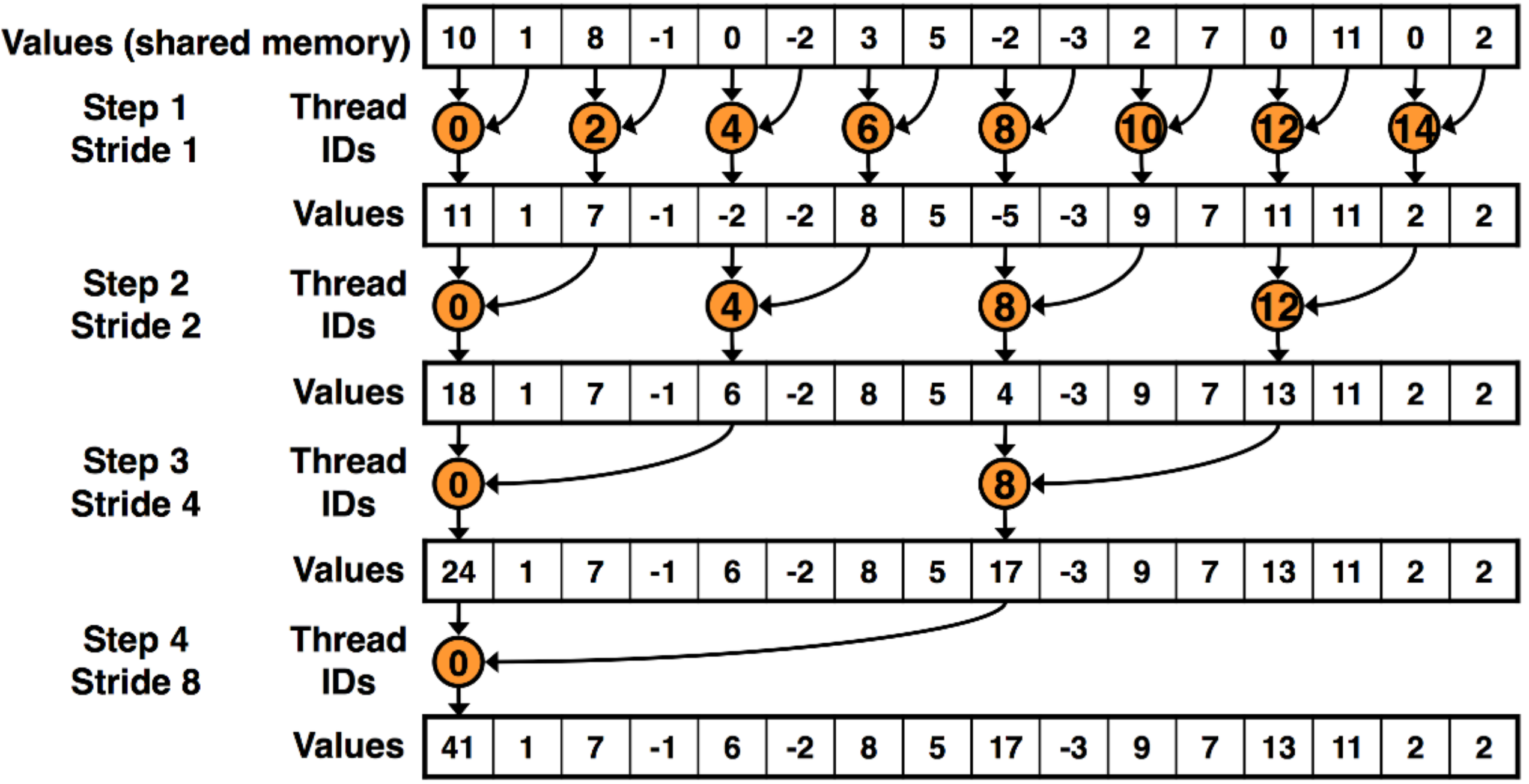
```
__global__ void reduce0(int *g_idata, int *g_odata) {
      extern __shared__ int sdata[];  //extern required for dyn alloc

// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();       //each block gets blockDim.x data

// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2) {
if (tid % (2*s) == 0) {
sdata[tid] += sdata[tid + s]; }
__syncthreads(); }                    //completes reduction

// write result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] = sdata[0]; }
// sum g_odata from 0 to numBlocks
```

# Parallel Reduction in Block



```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
        }
__syncthreads(); }
```

**Replaced with**

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
   int index = 2 * s * tid;
   if (index < blockDim.x) {
         sdata[index] += sdata[index + s];
   }
__syncthreads(); }
```

This gives speedup of 2.3

http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Can get speedup of 30.