CSE 564: Visualization

CUDA Specifics

Klaus Mueller

Computer Science Department

Stony Brook University

Setup CUDA

Compute Unified Device Architecture

- Driver, Toolkit and SDK http://www.nvidia.com/object/cuda_get.html

Inside toolkit

- NVCC
- Visual Studio syntax highlighting
- CUDA BLAS (CUBLAS) and FFT (CUFFT) libraries

Other resources

- CUDA Visual Profiler
- CUDA-GDB for Linux

more later…

Function Qualifiers

Device Global, & Host

- To specify whether a function executes on the host or on the device
- __global__ must return void

Function	Exe on	Call from
__device__	GPU	GPU
__global__	GPU	CPU
__host__	CPU	CPU

```
__global__ void Func (float* parameter);
```

Variable Qualifiers

Shared, Device & Constant

- To specify the memory location on the device of a variable
- __shared__ and __constant__ are optionally used together with __device__

Variable	Memory	Scope	Lifetime
__shared__	Shared	Block	Block
__device__	Global	Grid	Application
__constant__	Constant	Grid	Application

```
__constant__    float ConstantArrary[16];
__shared__      float SharedArray[16];
__device__       ……
```

Execution Configuration

<<< Grids, Blocks>>>

- Kernel function must specify the number of threads for each call (dim3)

<<< Grids, Blocks, Shared>>>

- It can specify the number of bytes in shared memory that is dynamically allocated per block (size_t)

```
dim3 dimBlock(8, 8, 2);
dim3 dimGrid(10, 10, 1);
KernelFunc<<<dimGrid, dimBlock>>>(…);

KernelFunc<<< 100, 128 >>>(...);
```

Device Side Parameters

Threads get parameters from execution configuration

- Dimensions of the grid in blocks

```
dim3 gridDim;
```

- Dimensions of the block in threads

```
dim3 blockDim;
```

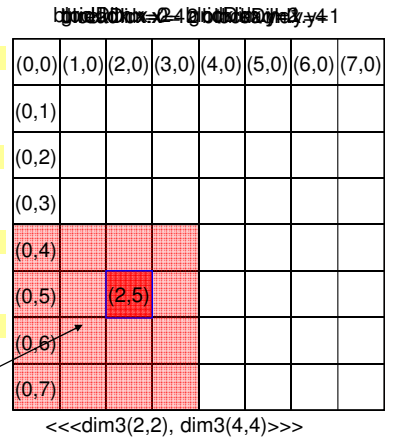
- Block index within the grid

```
dim3 blockIdx;
```

- Thread index within the block

```
dim3 threadIdx;
```

```
blockIdx.x *blockDim.x + threadIdx.x;
blockIdx.y *blockDim.y + threadIdx.y;
```



Anatomy of a Kernel Function Call

Define function as device kernel to be called from the host:

```
__global__ void KernelFunc(...);
```

Configuring thread layout and memory:

```
dim3    DimGrid(100,50);   // 5000 thread blocks
dim3    DimBlock(4,8,8);   // 256 threads per (3D) block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
```

Launch the kernel (<<, >> are CUDA runtime directives)

```
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

Program Constructs

Synchronization

- any call to a kernel function is asynchronous from CUDA 1.0 on
- explicit synchronization needed for blocking

Memory allocation on the device

- use cudaMalloc(*mem, size)
- resulting pointer may be manipulated on the host but allocated memory cannot be accessed from the host

Data transfer from and to the device

- use cudaMemcpy(devicePtr, hostPtr, size, HtoD) for host→device
- use cudaMemcpy(hostPtr, devicePtr, size, DtoH) for device→host

Number of CUDA devices

- use cudaGetDeviceCount(&count);

Program Constructs

Get device properties for `cnt` devices

```
for (i=0; i<cnt, i++)
    cudaGetDeviceProperties(&prop,i);
```

Some useful device properties (see CUDA spec for more)

- totalGlobalMemory
- warpSize
- maxGridSize
- multiProcessorCount
- ….

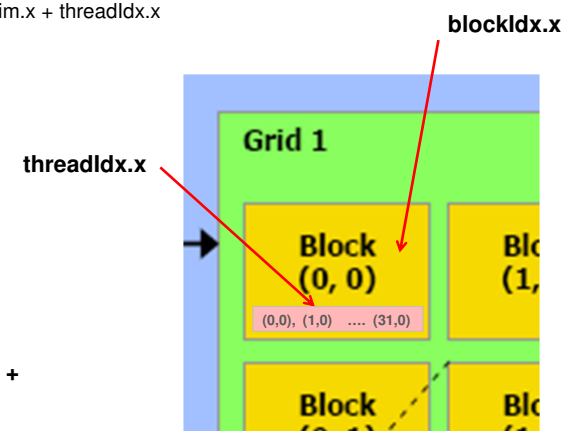
Example: Vector Add (CPU)

```
void vectorAdd(float *A, float *B, float *C, int N) {
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i]; }

int main() {
    int N = 4096;
            // allocate and initialize memory
    float *A = (float *) malloc(sizeof(float)*N);
    float *B = (float *) malloc(sizeof(float)*N);
    float *C = (float *) malloc(sizeof(float)*N);
    init(A); init(B);

    vectorAdd(A, B, C, N);          // run kernel
    free(A); free(B); free(C);}     // free memory
```

from: Dana Schaa and Byunghyun Jang, NEU

Example: Vector Add (GPU)

```
__global__ void gpuVecAdd(float *A, float *B, float *C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x
    C[tid] = A[tid] + B[tid]; }
```



tid = blockId.x * blockDim.x + threadIdx.x

blockDim.x=32

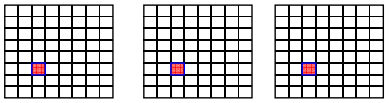
Example: Vector Add (GPU)

```
int main() {
    int N = 4096;       // allocate and initialize memory on the CPU
    float *A = (float *) malloc(sizeof(float)*N); *B = (float *) malloc(sizeof(float)*N); *C =
    (float*)malloc(sizeof(float)*N)
    init(A); init(B);
            // allocate and initialize memory on the GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, sizeof(float)*N);   cudaMalloc(&d_B, sizeof(float)*N);     cudaMalloc(&d_C,
    sizeof(float)*N);
    cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);    cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
            // configure threads
    dim3 dimBlock(32,1);
    dim3 dimGrid(N/32,1);
            // run kernel on GPU
    gpuVecAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
            // copy result back to CPU
    cudaMemcpy(C, d_C, sizeof(float)*N, DtoH);
            // free memory on CPU and GPU
    cudaFree(d_A);   cudaFree(d_B);    cudaFree(d_C);
    free(A);   free(B);   free(C); }
```

Example : Adding Matrix

C++

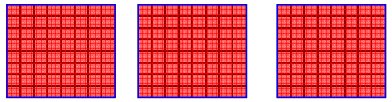
```
void AddMatrix (int *A, int *B, int *C,
                int w,  int d)
{   for ( int j = 0; j < d; j++)
      for( int i = 0; i < w; i++)
        { int index=j*w+i;
        C[index] = A[index] + B[index];
}}
```



CUDA

```
AddMatrix<<<1,dim3(w,d,1)>>>(A,B,C);
```

```
__global__ void AddMatrix (int *A,int *B,int *C)
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        int id = j*blockDim.x+i;
        C[id]=A[id]+B[id];
}
```



- A simple parallel computing kernel
- It rewrites “for” loops as execution parameters
- Block ←→ multiprocessor (w*d<=512)

Memory Management

Linear memory allocation

```
cudaMalloc(), cudaFree()
```

Linear memory copy

```
cudaMemcpy(),
```

Constant Memory copy

```
cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()
```

Memory query

```
cudaGetSymbolAddress(), cudaGetSymbolSize()
```

There are similar functions for pitched linear memory

Example : Adding Matrix

```
void * a,*b,*c;
cudaMalloc((void**)&a, w*d*sizeof(int));
cudaMalloc((void**)&b, w*d*sizeof(int));
cudaMalloc((void**)&c, w*d*sizeof(int));
…//load data into a, b;
cudaMemcpy(a, w*d*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(b, w*d*sizeof(int),cudaMemcpyHostToDevice);
dim3 dimBlock(BLOCKSIZE , BLOCKSIZE );
dim3 dimGrid( w/dimBlock.x, d/dimBlock.y );
AddMatrix <<<dimGrid, dimBlock>>> (a,b,c,w );
cudaMemcpy(c, w*d*sizeof(int),cudaMemcpyDeviceToHost);
cudaFree(a); cudaFree(b); cudaFree(c);
```

Allocate global memory (read and write)

Assume multiple of BLOCKSIZE

Launch kernel

Copy result

Free memory

```
__global__ void AddMatrix (int *A, int *B, int *C, int w)
{
        int i = blockIdx.x *blockDim.x + threadIdx.x;
        int j = blockIdx.y *blockDim.y + threadIdx.y;
        int id = j* w +i;
        C[id]=A[id]+B[id];
}
```

Define kernel

Get threadIdx, blockIdx here

Parallel and Synchronize

Threads execute in asynchronous manner in general

- Threads within one block share memory and can synchronize

```
void __syncthreads();
```

- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory
- No such function in CG. CG can do this by multi-pass render

CUDA Graphics API

CUDA Graphics API

- Texture (1D 2D 3D)

Texture Memory Advantages

Texture fetch versus global or constant memory read

- Cached, better performance if fetch with locality
- Not subject to the memory coalescing constraint for global and constant memory
- 2D address → (tex2D(tex, x, y))
- Filtering → (nearest neighbor or linear)
- Normalized coordinates → ([0,1] or [w, h])
- Handling boundary address → (clamp or wrap)
- Read only!

Binding textures

- CUDA arrays are opaque memory layouts optimized for texture fetching
- We can also bind texture on pitched linear memory and linear memory

1. Declaring texture reference, format and cudaArray

```
texture<Type, Dim, ReadMode> texRef;
cudaArray* cu_array;
cudaChannelFormatDesc cudaCreateChannelDesc<T>();
```

2. Memory management

```
cudaMallocArray(), cudaFreeArray(), cudaMemcpyToArray()
```

3. Bind/Unbind texture before/after texture fetching

```
cudaBindTextureToArray(), cudaUnbindTexture()
```

Example : Texture Example

```
texture<unsigned char, 2, cudaReadModeElementType> texr;
…

cudaChannelFormatDesc chDesc = cudaCreateChannelDesc<unsigned char>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &chDesc, w, h);
cudaMemcpyToArray( cuArray, 0, 0, input, data_size, cudaMemcpyHostToDevice );

cudaBindTextureToArray(texr, cuArray);
 …
//launch kernel. Inside kernel,  use tex2d(texr, idxX, idxY );
…//read result from device
cudaUnbindTexture(texr );
cudaFreeArray( cuArray );
cudaFree( data );
```

Bind texture before reference

Unbind texture after reference

Free cudaArray and memory

CUDA Performance

CUDA Performance

- Instruction Optimization
- Branching Overhead
- Global Memory Coalescing
- Shared Memory Bank Conflicts
- Streaming
- Strategy Summary

Instruction Optimization

Compiling with “-usefastmath”

Single or double precision

Unrolling loops

- Overhead by loop/branching is relatively high

```
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
    sum += data[sharMemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

- Results in 2-fold performance increase

```
sum = data[sharMemPos - 1] * d_Kernel[2]
      + data[sharMemPos + 0] * d_Kernel[1]
      + data[sharMemPos + 1] * d_Kernel[0];
```

Loop Unrolling

Compiler automatically unrolls small loops

Can use #pragma directive to enforce unrolling

```
#pragma unroll 5
for (int i = 0; i < n; ++i)
```

- unrolls loop into groups of 5
- *#pragma unroll 1* prevents compiler from unrolling
- *# pragma unroll* completely unrolls loop

Loop Unrolling: Example

Operation of original loop

- 2 floating point arithmetic instructions
- 1 branch instruction
- 2 address arithmetic instructions
- 1 loop counter increment

→ 1/3 instructions are calculation instructions

→ limits performance to 1/3 of peak performance

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

Loop unrolling

- eliminates overhead
- in addition, array indexing can now be done using offsets

```
Pvalue += Ms[ty][k] * Ns[k][tx] + …
          Ms[ty][k+15] * Ns[k+15][tx];
```

Branching

If, **Switch**, **Do**, **While** statements

Can lead to diverging threads within a warp

- Threads executing different branches
- Will serialize the execution paths (disturbs parallelism)
- Rejoin when all diverging paths have been executed

Remedies

- Group warps into similar execution flows beforehand
- Unroll loops
- Remove branches algorithmically

Optimizing Memory Usage

Minimizing data transfers with low bandwidth

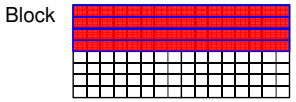
- Minimizing host & device transfer
- Maximizing usage of shared memory
- Re-computing can sometimes be cheaper than transfer

Organizing memory accesses based on the optimal memory access patterns

- Important for global memory access (low bandwidth)
- Shared memory accesses are usually worth optimizing only in case they have a high degree of bank conflicts

Global Memory Coalescing

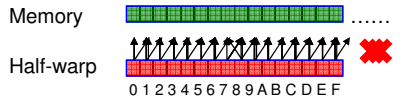
Warps & global memory



- Threads execute by warp (32)
- Memory read/write by half warp (16)
- Global memory is considered to be partitioned into segments of size equal to 32, 64, or 128 bytes and aligned to these sizes.
- Block width must be divisible by 16 for coalescing
- Check your hardware (Compute Capability 1.x)
- Greatly improves throughput (Can yield speedups of >10)

Global Memory Coalescing

Compute Capability 1.0 or 1.1



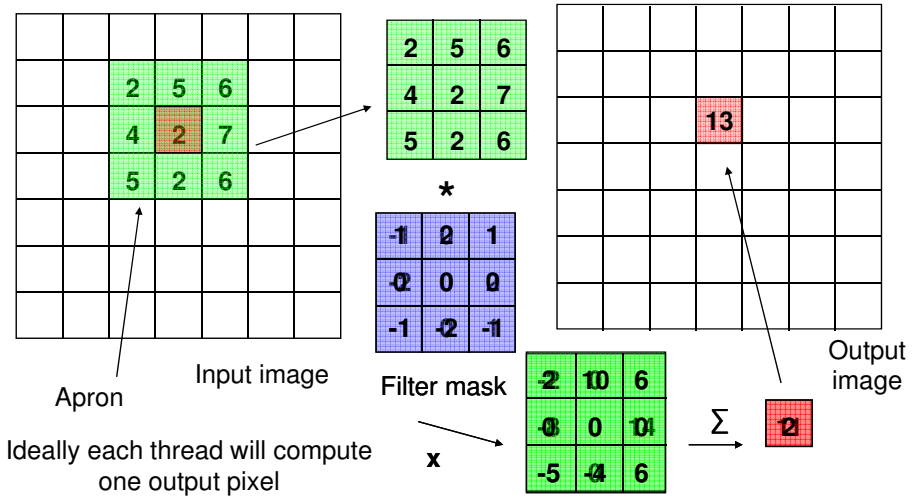
- Aligned 64 or 128 bytes segment
- Sequential warp
- Divergent warp
- See some good patterns in CUDA document and CUDA SDK samples

Compute Capability 1.2 or higher

- 32, 64 or 128 bytes segment
- Any pattern as long as inside segment

Returning Example: Sobel Filter

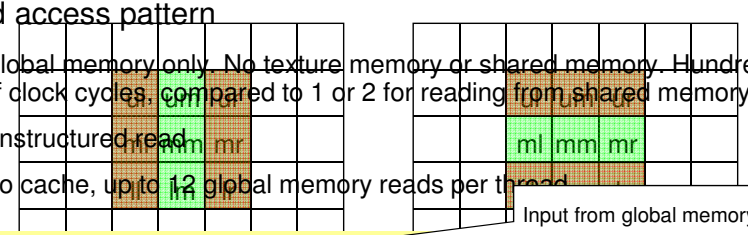
- Discrete convolution with Sobel mask



R/W Global Memory

Bad access pattern

- Global memory only. No texture memory or shared memory. Hundreds of clock cycles, compared to 1 or 2 for reading from shared memory
- Unstructured read
- No cache, up to 12 global memory reads per thread




```
__global__ void
SobelBadKernel(unsigned char* Input, unsigned char* output,unsigned int width, unsigned int
    height)
{
….//calculate the index for ur, ul, um, ml, mr, ll, lm, lr.
 float Horz=Input[ur] +Input[lr] +2.0*Input[mr] -2.0*Input[ml] -Input[ul] -Input[ll] ;
 float Vert=Input[ur] +Input[ul] +2.0*Input[um] -2.0*Input[lm] -Input[ll] -Input[lr] ;
 output[resultindex]   = abs(Horz)+abs(Vert);
}
```

Output to another global memory

Reduce Global Memory Read

```
__device__ unsigned char ComputeSobel(
        unsigned char ul,
        unsigned char um,
        unsigned char ur,
        unsigned char ml,
        unsigned char mm, //not used
        unsigned char mr,
        unsigned char ll,
        unsigned char lm,
        unsigned char lr,
        float fScale ){
  short Horz = ur + 2*mr + lr - ul - 2*ml - ll;
  short Vert = ul + 2*um + ur - ll - 2*lm - lr;
  short Sum = (short) (fScale*(abs(Horz)+abs(Vert)));
  if ( Sum < 0 ) return 0; else if ( Sum > 255 ) return 255;
  return (unsigned char) Sum;}
```

2	5	6
4	2	7
5	2	6

Reduce 12 reads into 8 or 9 reads

Reading Texture Memory

Take advantage of CUDA (texture memory)

- Using cache (texture memory) to enhance performance
- Each kernel can compute more than one pixels. This can help to exploit locality for cache
- Texture memory itself is optimized for coalescing

Reading Texture Memory

- Texture memory only.
- No shared memory

```
unsigned char *pSobel = (unsigned char *) (((char *) pSobelOrig
for ( int i = threadIdx.x; i < w; i += blockDim.x ) {
   unsigned char pix00 = tex2D( tex, (float) i-1, (float) blockIdx.x
   unsigned char pix01 = tex2D( tex, (float) i+0, (float) blockIdx.x
   unsigned char pix02 = tex2D( tex, (float) i+1, (float) blockIdx.x-1 );
   unsigned char pix10 = tex2D( tex, (float) i-1, (float) blockIdx.x+0 );
   unsigned char pix11 = tex2D( tex, (float) i+0, (float) blockIdx.x+0 );
   unsigned char pix12 = tex2D( tex, (float) i+1, (float) blockIdx.x+0 );
   unsigned char pix20 = tex2D( tex, (float) i-1, (float) blockIdx.x+1
   unsigned char pix21 = tex2D( tex, (float) i+0, (float) blockIdx.x+1
   unsigned char pix22 = tex2D( tex, (float) i+1, (float) blockIdx.x+1 );
   pSobel[i] = ComputeSobel(pix00, pix01, pix02, pix10, pix11, pix12,
                            pix20, pix21, pix22, fScale );}
```

Global memory as output. Need consider coalescing when write back

One thread computes (width/ blockDim.x) pixels

Read from texture memory

Improve Caching?

Disadvantage

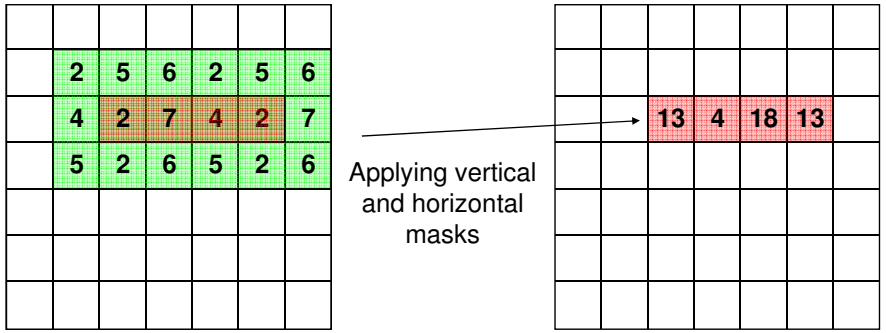
- Only using hardware cache to handle spatial locality
- A pixel may be still loaded 9 times in total due to cache miss

Take advantage of CUDA Shared Memory

- Shared memory can be as fast as register! As a user-controlled cache.

1. Together with texture memory, load a block of the image into shared memory
2. Each thread compute a consecutive rows of pixels (sliding window)
3. Writing result to global memory

Returning Example : Sobel Filter



Each thread will compute a number of consecutive rows of pixel

Computing all pixels inside one block (without apron)

Reading Shared Memory

- Shared memory + texture memory.

```
__shared__ unsigned char shared[];
kernel<<<blocks, threads, sharedMem>>>(…);
```

2	5	6	2	5	6
4	2	7	4	2	7
5	2	6	5	2	6

```
……// copy a large tile of pixels into shared memory
   __syncthreads();
……// read 9 pixels from shared memory
 out.x = ComputeSobel(pix00, pix01, pix02,  pix10, pix11, pix12, pix20, pix21, pix22, fScale );
……//read p00, p10, p20
out.y = ComputeSobel(pix01, pix02, pix00,  pix11, pix12, pix10, pix21, pix22, pix20, fScale );
……// read p01, p11, p21
out.z = ComputeSobel( pix02, pix00, pix01,  pix12, pix10, pix11, pix22, pix20, pix21, fScale );
……// read p02, p12, p22
out.w = ComputeSobel( pix00, pix01, pix02,  pix10, pix11, pix12, pix20, pix21, pix22, fScale );
   __syncthreads();
```

Loading data under current window, 9 reads

Sliding window right, reuse 6, update 3

Sliding window right, reuse 6, update 3

Sliding window right, reuse 6, update 3

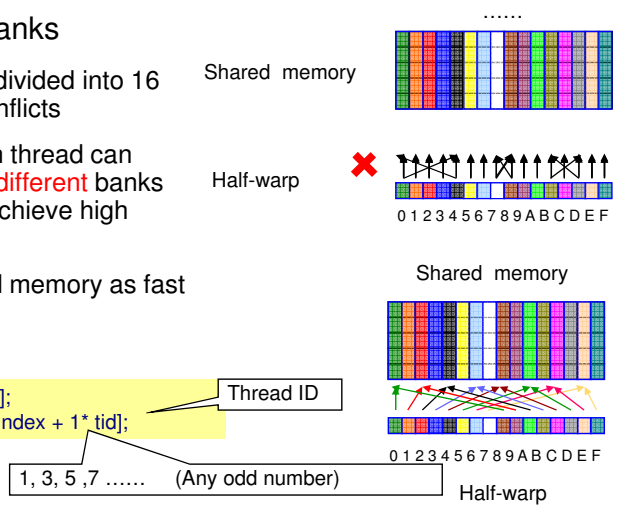
Shared Memory Bank Conflicts

Shared memory banks

- Shared memory is divided into 16 banks to reduce conflicts
- In a half-warp, each thread can access 32-bit from different banks simultaneously to achieve high memory bandwidth
- Conflict-free shared memory as fast as registers
- Linear

```
shared__ float shared[32];
float data = shared[BaseIndex + 1* tid];
```

- Random



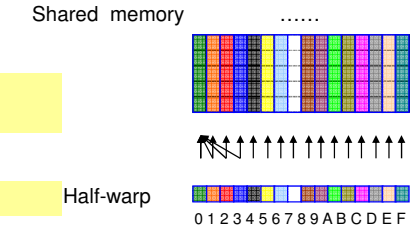
Shared Memory Bank Conflicts

4-way bank conflicts

```
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

No bank conflicts

```
char data = shared[BaseIndex + 4 * tid];
```

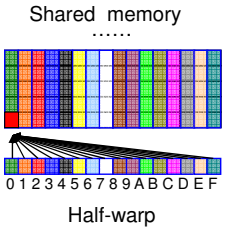


- In shared memory edge detection example, 4 pixels are chosen as a group (4 unsigned char = 32 bit)
- If data is larger than 32 bits, one way to avoid bank conflicts in this case is to split data. It might not always improve and will perform worse in future architectures
- Structure assignment can be used

Shared Memory Broadcasting

Shared memory read a 32-bit word and broadcast to several threads simultaneously

- Read
- Reduce or resolve bank conflicts if set to broadcasting
- Which word is selected as the broadcast word and which address is picked up for each bank at each cycle is unspecified



Streams

Help manage concurrency

- Allows overlap of memory copy with computations
- In some way similar to OpenCL command enqueueing

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

Best Programming Practices

Three basic strategies

- Maximize parallel execution
- Optimize memory usage → achieve maximum memory bandwidth
- Optimize instruction bandwidth → maximize instruction throughput

Maximized parallel execution

- Minimize number of synchronization barriers → let it flow
- Minimize divergent flows
- Better synchronize within a block than across → group threads

Optimize memory usage

- Map poor coalescing patterns in global memory to shared memory
- Load data in coalesced chunks before computation begins
  → #1: Get good coalescing in global memory
  → #2: Get conflict-free data access in shared memory

Best Programming Practices

Maximize instruction throughput

1. Instruction level

   →operator, branches and loops

2. Optimize execution configuration

Kernel will fail to launch if

- Number of threads per block >> max number of threads per block
- Requires too many registers or shared memory than available

Best Programming Practices

Block

- At least as many blocks as multiprocessors (SMs)
- More than one block per SM → have 2 or more active blocks

Threads

- Chose number of threads/block as a multiple of the warp size
- Typically 192 or 256 threads per block
- But watch out for required registers and shared memory
- Check **Visual profiler** or Occupancy Calculator

Multiprocessor occupancy

- Ratio of number of active warps per SM over max number of warps
- **Visual profiler** or Occupancy Calculator
  - Choose thread block size based on shared memory and registers

Returning Example : Sum of Numbers

Add up a large set of numbers

- Normalization factor:

$$S = \sum_{i=0}^{n-1} v[i]$$

- Mean square error:

$$MSE = \sum_{i=0}^{n-1} (a[i] - b[i])^2$$

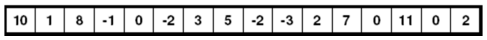
Number of addition operations:

$$v[0] + v[1] + v[2] + \cdots + v[n-1]$$

n-1 additions → How to optimize?

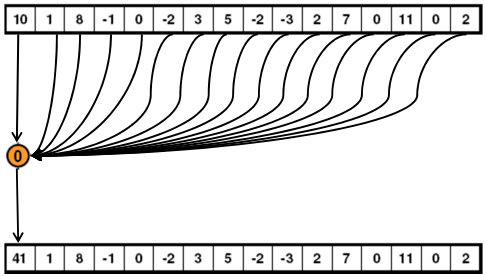
Non-parallel approach

Input numbers:



- Non-parallel approach:

Generate only one thread

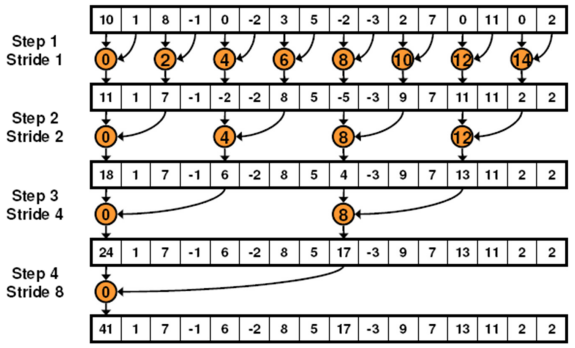


O(n) additions

Rule 1 : Maximized parallel execution

Parallel Approach: Kernel 1

- Interleaved addressing: Kernel 1



16 threads in a half wrap. Only 8 of them are active in the first loop

Uncoalesced global memory reading and writing pattern

O(logn) additions

Rule 2 : Optimize memory usage

Parallel Approach: Kernel 1

CUDA code:

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // 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();

  // 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();
  }

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

Rule 3: Maximize instruction throughput

inefficient statement, % operator is very slow

Writing with global memory coalescing

Parallel Approach: Kernel 2

Refinement strategy:

Just replace divergent branch in inner loop:

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

Initially each thread corresponds to 1 element

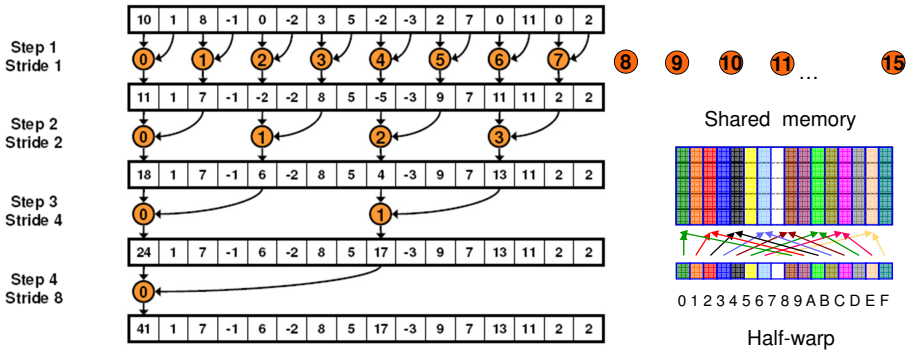
With strided index and non-divergent branch:

```
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();
}
```

Initially each thread corresponds to 2 element

Parallel Approach: Kernel 2

- Interleaved addressing: Kernel 2

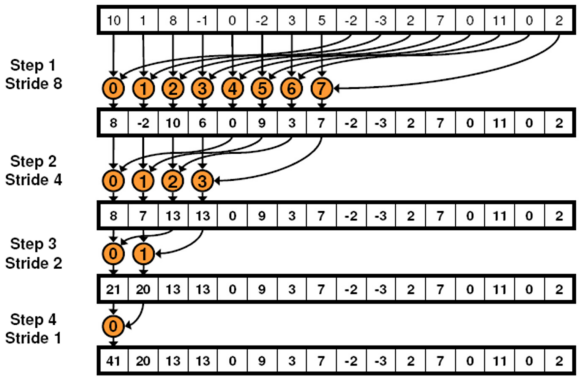


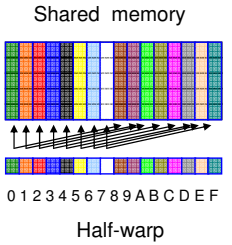
Shared memory conflict !

Law 2 : Optimize memory usage

Parallel Approach: Kernel 3

- Sequential addressing: Kernel 3





Conflict-free

Parallel Approach: Kernel 3

CUDA code:

Just replace strided indexing in inner loop:

```
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();
}
```

Every thread is active in first loop interation

With reversed loop and threadID-based indexing:

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

Half of the threads are idle in first loop iteration! Wasteful!

Law 3: Maximize instruction throughput

Toward Final Optimized Kernel

Performance for 4M numbers:

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Further Optimizations

Kernel 4 Rule 3: Maximize instruction throughput

- Halve the number of blocks, with two loads

Kernel 5

- Unrolling last loop

Kernel 6

- Completely unrolling loops

Kernel 7

- Multiply element per thread
- See details changes in

  M. Harris, Optimizing parallel reduction with CUDA

Towards Final Optimized Kernel

Performance for 4M numbers:

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

Final optimized kernel:

Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

CUBLAS Example

Compute a vector's L2 norm $\|\mathbf{x}\| := \sqrt{x_1^2 + \cdots + x_n^2}$

- Single precision
  float cublasSnrm2 (int n, const float *x, int incx)
- Double precision
  double cublasDnrm2 (int n, const double *x, int incx)

```
cublasInit();                                                    initialize libray
 float *h_A;
h_A = (float*)malloc(n * sizeof(h_A[0]));                        initialize vector
…
cublasAlloc(n, sizeof(d_A[0]), (void**)&d_A);                    data transfer
cublasSetVector(n, sizeof(h_A[0]), h_A, 1, d_A, 1);
float norm2result=cublasSnrm2 (n, const float *x, 1);            compute norm
cublasFree(d_A); free(h_A);
cublasShutdown();                                                Wrap-up
```

Linear Algebra

Dense Linear Algebra

- CUBLAS (Nvidia)
  Basic BLAS functions
- CULA
- MAGMA

Sparse Linear Algebra

- SPMV (Nvidia)
  General sparse matrix
- CNC (Concurrent Number Cruncher)
  Symmetric sparse matrix with linear solver

Others

CUFFT (Nvidia)

Jasper for DWT (Discrete wavelet transform)

OpenViDIA for computer vision

CUDPP for radix sort

…

To Probe Further

NVIDIA CUDA Zone:

- http://www.nvidia.com/object/cuda_home.html
- Lots of information and code examples
- NVIDIA CUDA Programming Guide

GPGPU community:

- http://www.gpgpu.org
- User forums, tutorials, papers
- Good source: conference tutorials
  http://www.gpgpu.org/developer/index.shtml#conference-tutorial

Conclusions

- CUDA Introduction

Threads cooperate using shared memory

- CUDA Programming API

Launch parallel kernels

- CUDA Graphics API

Visualize the result

- CUDA Performance

Memory is complex but important

- CUDA Utilities

Improve programming productivity