Introduction to CUDA Programming

15-418/618: Parallel Computer Architecture and Programming
Recitation 2, February 2, 2018,
Roadmap

- Review
- Possibly Helpful Tips
- Performance Optimization Example
Vocabulary

- **CPU**: A central processor unit, i.e. a normal processor
- **GPU**: A graphics processing unit, i.e. what we are learning about
- **Host**: The “normal computer” to which the GPU is connected
  - Of especial note are the CPU(s) and memory
- **Device**: The GPU and its memory
- **CUDA**: Compute Unified Device Architecture. nVidia’s framework for utilizing their GPUs for general purpose programming
  - **OpenCL**: Open Computing Language. The “generic version”
Vocabulary, *cont, cont*

- **Global memory**: Device memory shared across the various blocks
  - `CUDAMalloc()`, `CUDAMemcpy()`, `CUDAFree()`

- **Shared memory**: Memory shared only by threads within the associated block (not across blocks)
  - `__shared__`
Vocabulary, cont

- **Kernel**: The work, written as a function, to be parallelized across the GPU’s cores.
- **Thread**: An abstraction for the work associated with an instance of the kernel.
- **Thread Block**: A partition of threads and associated work that will be dispatched to a Streaming Media (SM) processor, basically a GPU.
- **Block**: See *Thread Block*
- **Grid**: Set of all blocks
Vocabulary, cont, cont, cont

- **CUDA Core:** A single graphics processor core. Within the CUDA architecture, these aren’t one-offs, but architected into *Streaming Multiprocessors (SMs).*

- **Streaming Multiprocessor (SM):** A collection of **CUDA Cores** architected together to form a single GPU. Threads within a *thread block* concurrently execute on an SM.

- **Warp:** A division of a block created within the SM to assign work to cores. Warps aren’t schedule until a core is available for each thread within the warp.
Syntax, Etc.

- **nvcc**: nVidia C compiler. Can compiler host and device code.
- **__shared__**: Qualifier to declare a variable in shared (per thread block) memory
- **__global__**: Qualifier to place a function into device memory, for execution onto the device, but enabling it to be callable from the host.

- **cudaMalloc()**, **cudaMemcpy()**, **cudaFree()**
  - Allocates, Frees, and copies to/from device memory.
  - `cudaMemcpyHostToDevice/cudaMemcpyDeviceToHost` flag sets direction of copy

- **__syncthreads()**
  - Barrier to ensure all threads get there before any continue.
Syntax, Etc, cont

- `someGlobalFunctionName<<<N,M>>>
  - “Kernel Launch”
  - N thread blocks
  - M threads per thread block

- `blockIdx`: block index within the (up to 3D) grid
  - threadIdx.x is 1D index

- `threadIdx`: thread index within the (up to 3D) thread block
  - threadIdx.x is 1D index

- `int index = threadIdx.x + blockIdx.x * M;`
  - Global thread index, given M threads per block

- `blockDim, gridDim`
  - 3D block and grid dimensions
  - E.g., blockDim.x, gridDim.x, etc
A Picture Worth Some Number of Words

- A kernel is launched as a grid of blocks of threads
  - blockIdx and threadIdx are 3D
  - We showed only one dimension (x)

- Built-in variables:
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim

CUDA C/C++ Basics
Cyril Zeller, NVIDIA Corporation
Supercomputing 2011 Tutorial
Roadmap

- Review
- Possibly Helpful Tips
- Performance Optimization Example
Did it fly? Wrap All CUDA Library Calls

Definitions in file reduce.cu:
// Support for CUDA error checking
// Wrapper for CUDA functions

#define CHK(ans) gpuAssert((ans), __FILE__, __LINE__);

// Checker
inline void gpuAssert(CUDAError_t code, const char *file, int line) {
    if (code != CUDASuccess) {
        fprintf(stderr, "GPUassert: %s %s %s\n", 
            CUDAGetErrorString(code), file, line);
    }
}

// Cannot wrap kernel launches. Instead, insert this after each
// kernel launch.
#define POSTKERNEL CHK(CUDAPeekAtLastError())
cuda-gdb: Can get data off of the device

- **break** `reduce.cu:90`
  - Set breakpoint corresponding to line 90 of file `reduce.cu`.

- **print** `((@global float *) srcVecDevice)[1]`
  - Print contents of array in device memory

- **CUDA thread 2**
  - Shift focus to specified thread number

- **info locals**
  - Prints values of all currently-active local variables
  - **CUDA info threads**
  - Prints status of threads (in current block)
Some Advice

- Don’t wire down constants
- Don’t assume special properties of N
  - Multiple of block size, power of 2, ...
- Use function or macro to do rounding-up division
- Write checker code
  - Overall functionality
  - Individual steps on device
    - Must transfer data back to host to check
- Avoid printf within kernel functions
  - Only use on small examples or too much unordered output.
- Get the algorithm & abstract implementation and benchmark right before attempting low-level optimizations
  - Exploiting the various memory categories on device
  - Exploiting properties specific to block level
Some More Advice

- Even more so than with C programs, out-of-bounds memory writes in CUDA lead to bizarre and erratic behavior.
  - Write bounds checking code that gets invoked when program is run in DEBUG mode

- It’s possible to put printf’s in kernel code, but don’t rely on them
  - Often nothing gets printed, or values printed are incorrect.
  - Just because nothing prints, it doesn’t mean that part of the code wasn’t reached.

- Write host code that duplicates functionality of different parts of CUDA code
  - In debug mode, transfer results back to host memory and check values against this code
Why Is printf() weird?

- printf() output is stored in a circular buffer of a fixed size.
  - If the buffer fills, old output will be overwritten. The buffer's size defaults to 1MB and can be configured with CUDADeviceSetLimit(CUDALimitPrintfFifoSize, size_t size).

- This buffer is flushed only for
  - the start of a kernel launch
  - synchronization (e.g. CUDADeviceSynchronize())
  - blocking memory copies (e.g. CUDAMemcopy(...))
  - module load/unload
  - context destruction
  - Note: The list above does not include program exit.
    - If the call to CUDADeviceSynchronize() was removed from the example program above, the we would see no output

- Concurrency serialized upon output

Credit: Steven Fackler, Former 418/618 TA, SCS’13
Application Example: $N \times N$ Matrix Multiplication

Row $i$ \hspace{1cm} \times \hspace{1cm} \text{Column } j \hspace{1cm} = \hspace{1cm} \text{Element } i, j

$C_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j}$

\textbf{Complexity}
- $N^3$ multiplications
- $N^3$ additions

\textbf{Assume row-major access}

```
#define RM(r, c, width) (((r) * (width)) + (c))
```
Matrix Multiplication: Simple CPU Implementation

```c
void multMatrixSimple(int N, float *matA, float *matB, float *matC) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * matB[RM(k,j,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```
CPU Simple Performance

- Measured in GFLOPS
- Drops off for large values of N
- B has bad access pattern
Optimization #1: Pretranspose

- Transposed version of B has better access pattern
  - Transpose once
  - Use each element N times

\[
C_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b^T_{j,k}
\]
Transposing a Matrix

- Column-major ordering of elements

```c
#define CM(r, c, height) ((c) * (height) + (r))
```

- Transposing converts from row-major to column-major order

```c
void transposeMatrix(int N, float *matS, float *matD) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            matD[CM(i, j, N)] = matS[RM(i, j, N)];
}
```
Matrix Multiplication: Pretranspose Implementation

```c
void multMatrixTransposed(int N, float *matA, float *matB, float *matC) {
    float *tranB = scratchMatrix(N);
    transposeMatrix(N, matB, tranB);
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * tranB[RM(j,k,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```

\[ c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b^T_{j,k} \]
Pretranspose Performance

- Scales to large matrices
- “Cache-friendly” code

Graph showing performance in GFLOPS against matrix size.

Legend:
- Simple
- Transpose
Abstract Single Program Multiple Data (SPMD) Model

- M Processors, all executing same code
  - Called “kernels”
  - M based on problem size

- Share common global memory
  - And also have private memory for local variables
  - Make no assumptions about effect of memory access conflicts

- No synchronization primitives

- Called *threads*, but not at all like pthreads
  - Very simple & lightweight
  - All execute the same program
Interacting with SPMD Machine: Control

- Overall execution managed by code executing on host
- Launch set of kernels
  - Number & kernel function can vary with each launch
- Wait until all completed
  - Explicit or implicit synchronization
- Repeat as necessary
Structure of SPMD Program

- **Concept**
  - Partition computation into sequence of tasks
  - Perform each task over all data with single operation

- **Performance Limitations**
  - Synchronization requires waiting for slowest task
  - No locality of data
  - No locality of synchronization
Block/Thread Notation

- Idea (One-dimensional version)
  - Executing threads grouped into *blocks*
    - Each contains same number of threads
      - Host program specifies block size (blockDim.x)
  - Host program makes sure there are enough blocks to generate N threads
  - Some threads in last block should not get used

```c
__global__ void
inplaceReduceKernel(int length, int nlength, float *data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < nlength) {
        
    }
}
```
Interacting with SPMD Machine: Data

- Host acts as controller
- Does not have direct access to device memory

![Diagram of interacting with SPMD machine showing data exchange between host, device, and shared memory.]
CUDA Program

- **CUDA file (.cu) contains mix of device code & host code**
  - It’s up to you to understand which is which!

- **Device Code**
  - Kernels (**__global__**)  
    - Code for threads  
    - Must only reference device memory
  - Device functions (**__device__**)  
    - Called by kernels  
    - Only reference device memory  
    - Do not generate new threads

```c
__global__ void inplaceReduceKernel(int length, int nlength, float *data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < nlength) {
        // ...
    }
}

__device__ void deviceMult(float x, float y, float *dest) {
    *dest = x * y;
}
```
CUDA Program (cont.)

- **CUDA file (.cu) contains mix of device code & host code**
  - It’s up to you to understand which is which!

- **Host Code**
  - Conventional C/C++
  - Can only reference host memory
    - But, can have pointers to device memory
  - Manages the launching of threads
  - Manages movement of data between host & device memories
Matrix Multiplication: Simple CUDA Implementation

Each thread computes element $i, j$ of product

```c
__global__ void CUDASimpleKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N || j >= N)
        return;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i, k, N)] * dmatB[RM(k, j, N)];
    }
    dmatC[RM(i, j, N)] = sum;
}
```
**Host Code Example**

- Launch kernels to perform vector product

```c
void CUDAMultMatrixSimple(int N, float *dmatA, float *dmatB, float *dmatC)
{
    dim3 threadsPerBlock(LBLK, LBLK);
    dim3 blocks(updiv(N, LBLK), updiv(N, LBLK));
    CUDASimpleKernel<<<blocks, threadsPerBlock>>>(N, dmatA, dmatB, dmatC);
}
```

**Useful stuff**

- Compute  \( n / d \)

```c
// Integer division, rounding up
static inline int updiv(int n, int d) {
    return (n+d-1)/d;
}
```

- Setting number of threads per block:
  - Should be multiple of 32
  - Max value = 1024
Host Code Example (cont).

```c
void CUDAMultiply(int N, float *aData, float *bData, float *cData) {
    float *aDevData, *bDevData, *cDevData
    CUDAMalloc((void **) &aDevData, N*N * sizeof(float));
    CUDAMalloc((void **) &bDevData, N*N * sizeof(float));
    CUDAMalloc((void **) &cDevData, N*N * sizeof(float));
    CUDAMemcpy(aDevData, aData, N*N * sizeof(float),
                CUDAMemcpyHostToDevice);
    CUDAMemcpy(bDevData, bData, N*N * sizeof(float),
                CUDAMemcpyHostToDevice);

    CUDAMultMatrixSimple(N, aDevData, bDevData, tDevData);

    CUDAMemcpy(cData, cDevData, N*N * sizeof(float),
                CUDAMemcpyDeviceToHost);
    CUDAFree(aDevData); CUDAFree(bDevData); CUDAFree(cDevData);
}
```

**Observe:** Host can hold pointers to device memory, but cannot read or write device memory locations
Simple CUDA Performance

- Transpose
- Cuda Simple
Inverted Indexing Accessing Pattern

**Regular**

```c
__global__ void CUDASimpleKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    ...
}
```

**Inverted**

```c
__global__ void CUDASimpleKernelOld(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    ...
}
```
CUDA Inverted Indexing Performance

~10x worse
Why?!

GFLOPS

8 16 32 64 128 256 512 1024

Transpose
Cuda Simple Inverted
Cuda Simple

Lightly edited from Prof. Bryant, 15-418/618, Spring 2017
What’s the Difference?

CUDA threads numbered within block in row-major order
- \( X = \) column number, \( Y = \) row number

Threads with same value of \( Y \) map to single warp.

Threads with same value of \( Y \) and consecutive values of \( X \) map to consecutive positions in single warp

When single warp accesses consecutive memory locations, do block read or write

When single warp accesses separated memory locations, requires gather (read) or scatter (write)
Impact on Memory Referencing: Regular

Threads within warp have:
- same value of $k$
- same value of $i$
- consecutive values of $j$

Warp reads & writes match memory organization

```c
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

- Read A: $\mathit{dmatA}[\mathit{RM}(i,k,N)] = \mathit{Read A}$
- Threads in warp reference single location
- Read B: $\mathit{dmatB}[\mathit{RM}(k,j,N)] = \mathit{Read B}$
- Threads in warp do block read
- Write B: $\mathit{dmatC}[\mathit{RM}(i,j,N)] = \mathit{Write B}$
- Threads in block do block write
Impact on Memory Referencing: Inverted

---

```c
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Read A  \( = \text{dmatA[RM(i,k,N)]} \)

Read B  \( = \text{dmatB[RM(k,j,N)]} \)

Write B  \( \text{dmatC[RM(i,j,N)]} = \)

- Threads in warp do gather
- Threads in warp reference single location
- Threads in block do scatter

- Threads within warp have:
  - same value of \( k \)
  - consecutive values of \( i \)
  - same value of \( j \)

- Warp reads/writes does not match memory organization
Relation to Hardware

- **Optimizing memory instruction performance**
  - Load faster than gather
  - Store faster than scatter

- **Avoiding memory conflicts**
  - Inverted code has multiple warps competing for same block of memory
Pretransposing with CUDA

/* Transpose matrix */
__global__ void
CUDATransposeKernel(int N, const float *dmatS, float *dmatD) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N || j >= N)
        return;
    dmatD[CM(i,j,N)] = dmatS[RM(i,j,N)];
}

__global__ void
CUDATransposedKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N || j >= N)
        return;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i,k,N)] * dmatB[RM(j,k,N)];
    }
    dmatC[RM(i,j,N)] = sum;
}
CUDA Pretranspose Implementations

Transpose operation must do either scatter or gather

GFLOPS

8 16 32 64 128 256 512 1024

Transverse
Cuda Simple Inverted
Cuda Simple
Cuda Transpose
Thinking About CUDA
GPU Hierarchy

■ Block Level
  ▪ Programmer partitions problem into blocks of K threads each
    ▪ 32 ≤ K ≤ 1024
    ▪ Multiple of 32
  ▪ Within block, have access to fast shared memory
  ▪ Within block, can synchronize with __syncthreads()  

■ Warp Level
  ▪ Each block implemented as set of warps
    ▪ 32 threads each
  ▪ Implemented using “SIMT” processor
    ▪ Single-instruction, multiple threads
    ▪ Guarantees stay synchronized
Programming with Blocks

- Localize computation within blocks
- Each performs sequence of tasks
- Each uses shared memory and local synchronization
**MM Optimization #2: Partitioning into Blocks**

- Generate results on block-by-block basis
- Localizes access to A and B
- N need not be multiple of block size

\[
C_{I,J} = \sum_{K=0}^{N_b-1} A_{I,K} \times B_{K,J}
\]
**CPU-based Blocked Implementation**

- **Use pretranspose**
  - Required for performance

- **Structure**
  - Outer loops index over blocks
  - Inner loops compute product for single block

- **Block size** $S_{BLK} = 8$

\[
C_{I,J} = \sum_{K=0}^{N_b-1} A_{I,K} \times B_{K,J}^{T}
\]
Blocked Multiplication
Implementation: Outer Loops

- Look at actual code to see how it handles cases where N is not multiple of block size

\[ C_{I,J} = \sum_{K=0}^{N_b-1} A_{I,K} \times B_{K,J} \]

```
void multMatrixTransposeBlocked(int N,
    float *matA, float *matB, float *matC) {
    float *tranB = scratchMatrix(N);
    transposeMatrix(N, matB, tranB);
    /* Zero out C */
    memset(matC, 0, N * N * sizeof(float));
    int i, j, k;
    for (i = 0; i <= N-SBLK; i+= SBLK)
        for (j = 0; j <= N-SBLK; j+= SBLK)
            for (k = 0; k <= N-SBLK; k+=SBLK)
                Compute contribution to C[i..i+SBLK-1][j..j+SBLK-1]
}
```
Blocked Multiplication
Implementation: Inner Loops

\[ C_{i+bi, j+bj} = a_{i+bi, k+bk} \times b_{j+bj, k+bk} \]

- \( i, j, k \) provide starting indices of blocks
- \( bi, bj, bk \) provide offsets within blocks

```c
for (int bi = 0; bi < SBLK; bi++)
    for (int bj = 0; bj < SBLK; bj++) {
        float sum = 0.0;
        for (int bk = 0; bk < SBLK; bk++)
            sum += matA[RM(i+bi, k+bk, N)] * tranB[RM(j+bj, k+bk, N)];
        matC[RM(i+bi, j+bj, N)] += sum;
    }
```
**CPU Implementations**

Fast blocked: Pretranspose + Unroll inner loop 8x and reassociate

- **Simple**
- **Transpose**
- **Blocked**
- **Transpose+Blocked**
- **Fast Blocked**

---

15-418/618: Lightly edited from Prof. Bryant, 15-418/618, Spring 2017
Blocking with CUDA

- Block size $\text{LBLK} = 32$
- Use one CUDA block for each block of destination matrix
- Enough CUDA blocks to cover $C$
- Each thread in block accumulates single destination value

- Fetch blocks $A$ & $B$ for $k = 0$
- Compute block product
- Fetch blocks $A$ & $B$ for $k = 32$
- Compute block product
- Fetch blocks $A$ & $B$ for $k = N-32$
- Compute block product
- Store values at destination
CUDA Block Kernel Structure

```c
__global__ void CUDABlockKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int bi = threadIdx.x;
    int bj = threadIdx.y;
    float sum = 0.0; // Accumulate result for C[i][j]
    // Shared space for two submatrices of A and B
    __shared__ float subA[LBLK*LBLK];
    __shared__ float subB[LBLK*LBLK];

    Loop over values of k

    if (i < N && j < N)
        dmatC[RM(i,j,N)] = sum;
}
```

- **Block size LBLK = 32**
  - blockDim.x = blockDim.y = 32
- **i, j index into source and destination arrays**
- **bi, bj index local arrays**
CUDA Block Loop Structure

```
for (int k = 0; k < N; k+= LBLK) {
    Fetch elements bi, bj for local arrays subA and subB
    // Wait until entire block gets filled
    __syncthreads();
    Compute contribution to element i, j of output
    // Wait until all products computed
    syncthreads();
}
```

- **Within loop, each thread plays two distinct roles**
  - Fetch elements from source arrays into shared memory
  - Compute one element of subblock product

- Fetch blocks A & B for next value of k
- Compute block product
Fetching Blocks

\[
\text{if } (i < N \land k+bi < N) \{ \\
\quad \text{subA[RM(bi,bj,LBLK)]} = \text{dmatA[RM(i,k+bi,N)]}; \\
\} \quad \text{else} \{ \\
\quad \text{subA[RM(bi,bj,LBLK)]} = 0.0; \\
\}\]

\[
\text{if } (j < N \land k+bi < N) \{ \\
\quad \text{subB[RM(bi,bj,LBLK)]} = \text{dmatB[RM(k+bi,j,N)]}; \\
\} \quad \text{else} \{ \\
\quad \text{subB[RM(bi,bj,LBLK)]} = 0.0; \\
\}\]

- \text{k is multiple of LBLK}
  - Coarse-grained
- \text{Fetch element } i, k+bi \text{ from A to get } \text{subA[bi,bj]}
- \text{Fetch element } k+bi, j \text{ from B to get } \text{subB[bi,bj]}
- \text{Set to 0 if out of range}
Computing Block Product

- Each thread in block accumulates single destination value

```
for (int bk = 0; bk < LBLK; bk++)
    sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
```

\[
C_{bi,bj} = \sum_{bk=0}^{b-1} a_{bi,bk} \times b_{bk,bj}
\]
CUDA Blocked Implementations

- Fast Blocked
- Cuda Simple
- Cuda Block
CUDA Inverted Indexing

Blocked version has similar indexing properties as unblocked

<table>
<thead>
<tr>
<th>GFLOPS</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast Blocked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cuda Simple Inverted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cuda Simple</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cuda Block Inverted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cuda Block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Warning!

```c
for (int k = 0; k < N; k+= LBLK) {
    if (i >= N || j >= N)
        continue;  // Skip if out of bounds

    Computation when in-bounds

    // Wait until everyone finished
    __syncthreads();

    Compute contribution to element i, j of output
    // Wait until all products computed
    __syncthreads();
}
```

- What’s wrong with this code?
Observations

- Making use of CUDA hierarchy can help
  - Lighter weight synchronization
  - Shared access to fast memory
  - Different blocks can proceed at different rates
    - (Not shown in this example)

- Advice
  - Implement pure data-parallel version first
  - Only exploit hierarchy for performance critical parts
  - Watch out for synchronization bugs
  - Proper memory referencing more important than these low-level optimizations
But Wait... THERE'S MORE!
Reading Memory with Float4’s

Idea suggested by Kayvon Fatahalian

- Fast Blocked
- Cuda Simple
- Cuda Block
- Cuda Quad Block
Idea

- Thread blocks compute products of 64x64 submatrices
- 1024 threads
- Organize as 64 rows X 16 columns
- Threads read & write memory in chunks of 16 bytes
  - 4 float’s each

16 columns of float4’s
Added Inner Step of Computation

- Each thread loops 16 times
  - Within loop, compute product:
    - 1x4 portion of A
    - 4x4 of B
    - Add sum to 1x4 portion of C
    - 16 multiplies, 16 adds

- Why so fast?
  - Makes maximum use of memory bus capability