Nvidia G80 Architecture and CUDA Programming

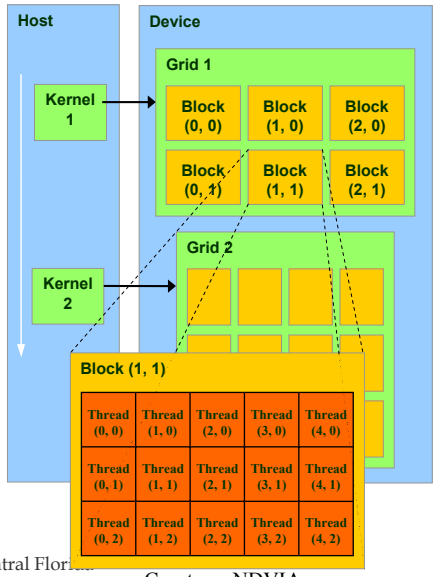
School of Electrical Engineering and Computer Science
University of Central Florida

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

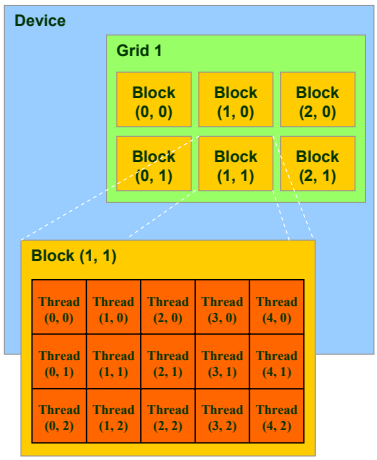
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

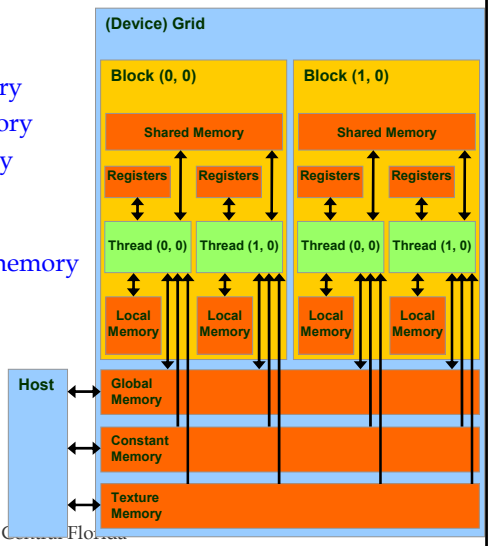
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

CUDA Device Memory Space Overview

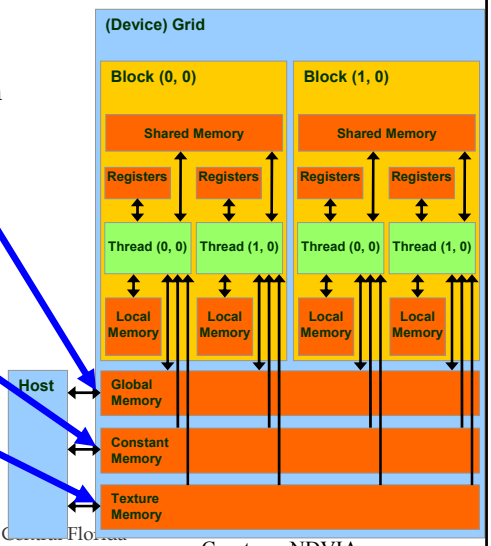
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu
University of Central Florida

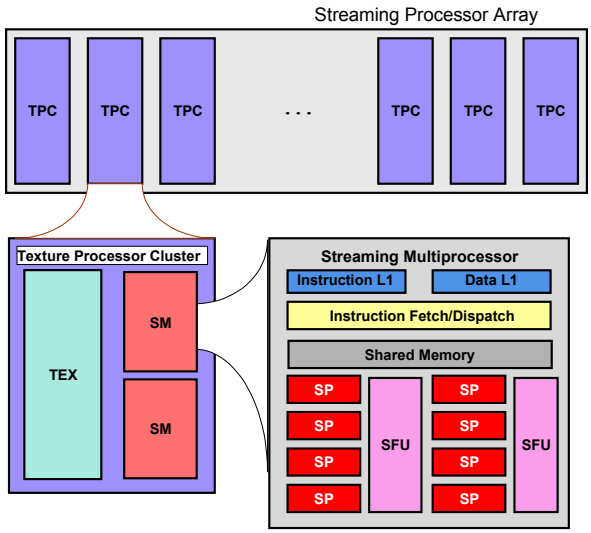
Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu
University of Central Florida
Courtesy: NDVIA

GeForce-8 Series HW Overview

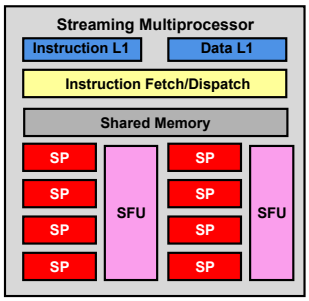


CUDA Processor Terminology

- SPA
  - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800, 30 in GTX280)
- TPC
  - Texture Processor Cluster (2 SM + TEX)
- SM
  - Streaming Multiprocessor (8 SP)
  - Multi-threaded processor core
  - Fundamental processing unit for CUDA thread block
- SP
  - Streaming Processor
  - Scalar ALU for a single CUDA thread

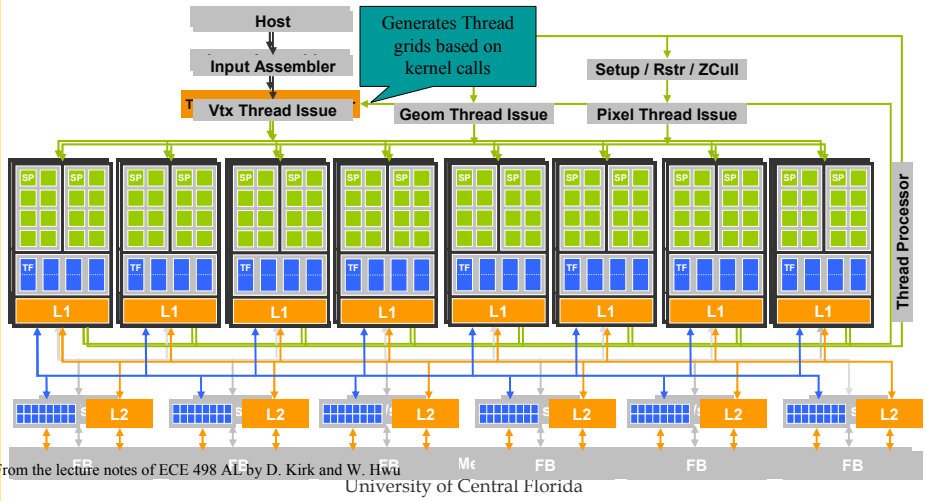
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 768 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- DRAM texture and memory access



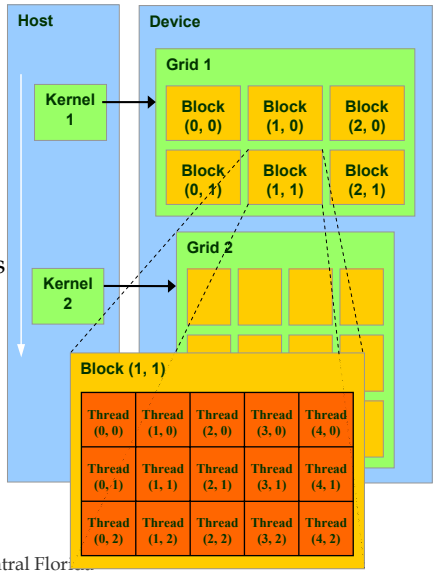
G80 Thread Computing Pipeline

- The future of GPUs is programmable processing
- So build the architecture around the processor

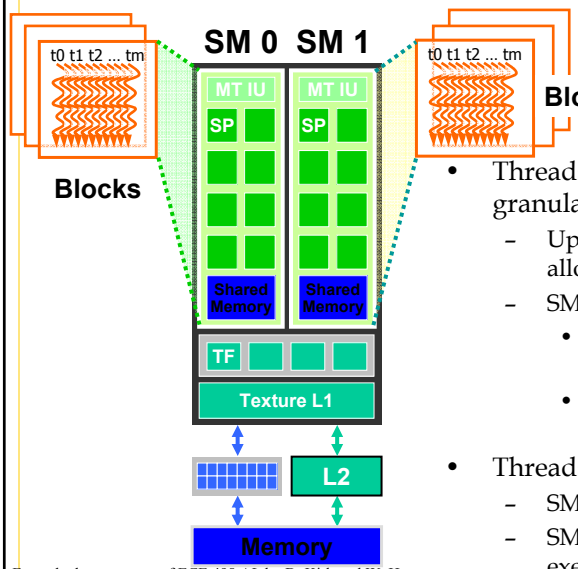


Thread Life Cycle in HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SM's
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks



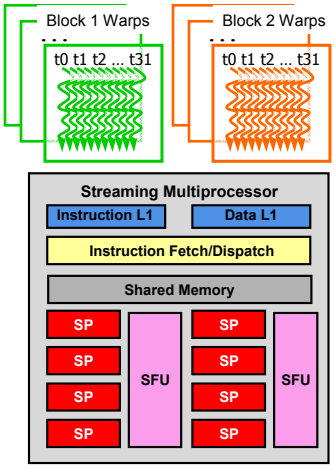
SM Executes Blocks



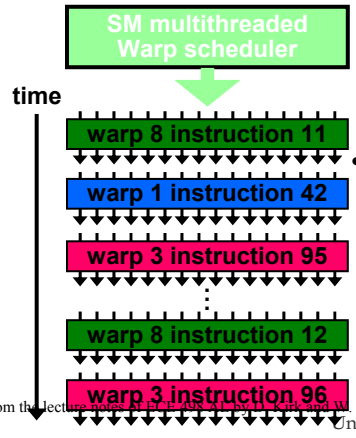
- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution

Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- ***Warps use the SIMD execution model***
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.



SM Warp Scheduling



- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

A Simple Running Example Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device

An Example: Matrix Multiplication P = M X N

- Simple code in C

```
void MatrixMulOnHost(const Matrix M, const
    Matrix N, Matrix P)
{
  for (int i = 0; i < M.height; ++i)
    for (int j = 0; j < N.width; ++j) {
      double sum = 0;
      for (int k = 0; k < M.width; ++k) {
        double a = M.elements[i * M.width + k];
        double b = N.elements[k * N.width + j];
        sum += a * b;
      }
      P.elements[i * N.width + j] = sum;
    }
}
```

Data Structure

```
typedef struct {
    int width;
    int height;
    int pitch;
    float* elements;
} Matrix;
```

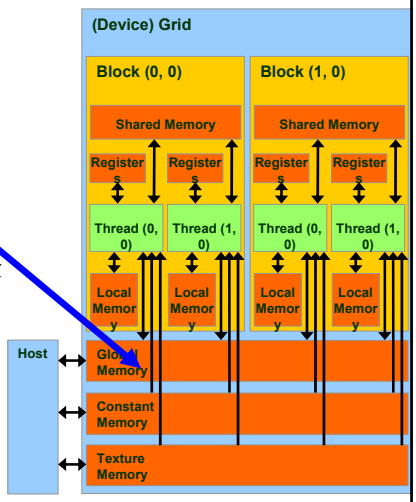
Optimizing the CPU code lays a solid foundation to optimize GPU code.

Analyzing the matrix multiplication (CPU) code

- # of instructions to be executed
  - # of memory access instructions (i.e., loads) to be executed
    - 2 * M.height * N.Width * M.width
    - Loading each element in M for N.Width times
    - Loading each element in N for M.Height times
- The ratio of computation over memory access instructions
  - For every two loads, one multiply and one add
- For CPU, cache locality (spatial and temporal) help to reduce the load latencies. For large M and N, temporal locality is low'
- Optimization?
  - Unroll and Jam.

CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu

CUDA Device Memory Allocation (cont.)

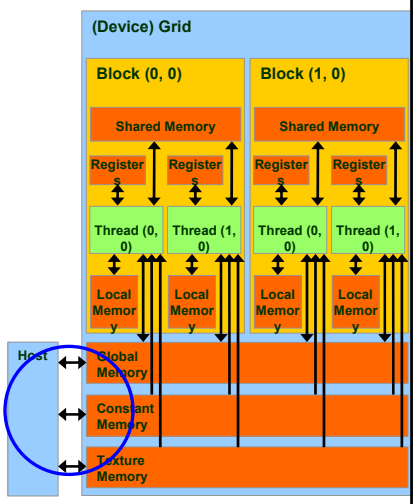
- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - "d" is often used to indicate a device data structure

```
BLOCK_SIZE = 64;
Matrix Md
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);
```

cudaMalloc((void)&Md.elements, size);**
cudaFree(Md.elements);

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous in CUDA 1.0



CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

CUDA Function Declarations (cont.)

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Calling a Kernel Function – Thread Creation

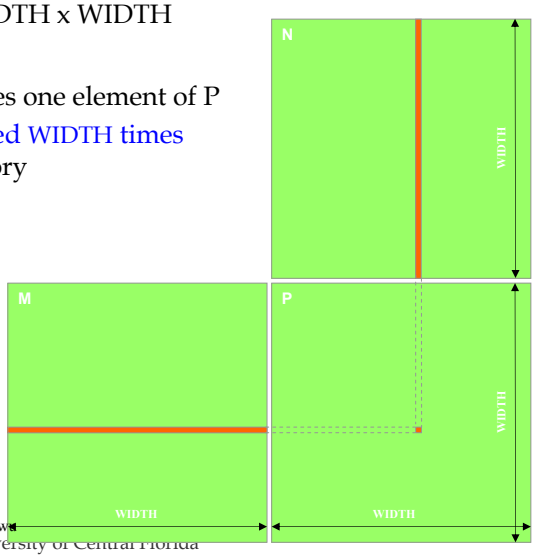
- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Programming Model: Parallelizing Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory



Step 1: Matrix Data Transfers

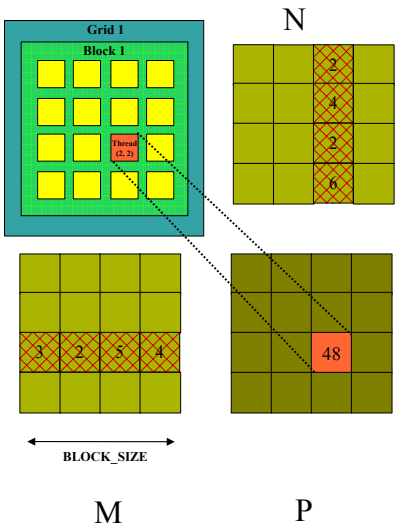
```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

Step 2: Matrix Multiplication
A Simple Host Code in C

```
// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
```


Multiply Using One Thread Block

- One Block of threads compute matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



Step 3: Matrix Multiplication Host-side Main Program Code

```
int main(void) {
// Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

// M * N on the device
    MatrixMulOnDevice(M, N, P);

// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```


Step 3: Matrix Multiplication Host-side code

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
```

Step 3: Matrix Multiplication Host-side Code (cont.)

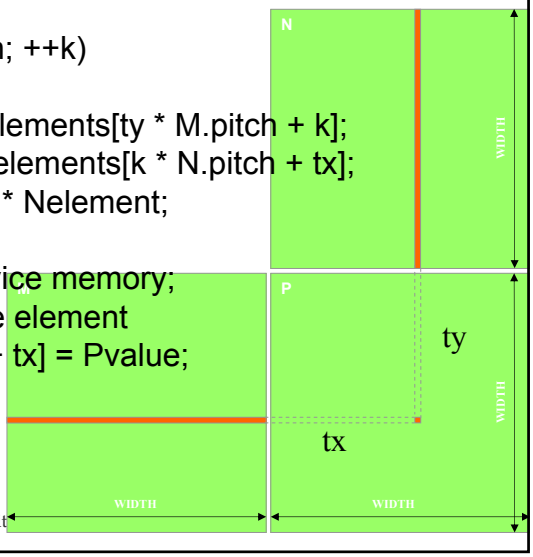
```
// Setup the execution configuration
   dim3 dimBlock(WIDTH, WIDTH);
   dim3 dimGrid(1, 1);

   // Launch the device computation threads!
   MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

   // Read P from the device
   CopyFromDeviceMatrix(P, Pd);

   // Free device matrices
   FreeDeviceMatrix(Md);
   FreeDeviceMatrix(Nd);
   FreeDeviceMatrix(Pd);
}
```


Step 4: Matrix Multiplication Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue is used to store the element of the matrix
   // that is computed by the thread
   float Pvalue = 0;
```

Step 4: Matrix Multiplication
Device-Side Kernel Function (cont.)

```
    for (int k = 0; k < M.width; ++k)
    {
        float Melement = M.elements[ty * M.pitch + k];
        float Nelement = Nd.elements[k * N.pitch + tx];
        Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P.pitch + tx] = Pvalue;
}
```



Step 5: Some Loose Ends

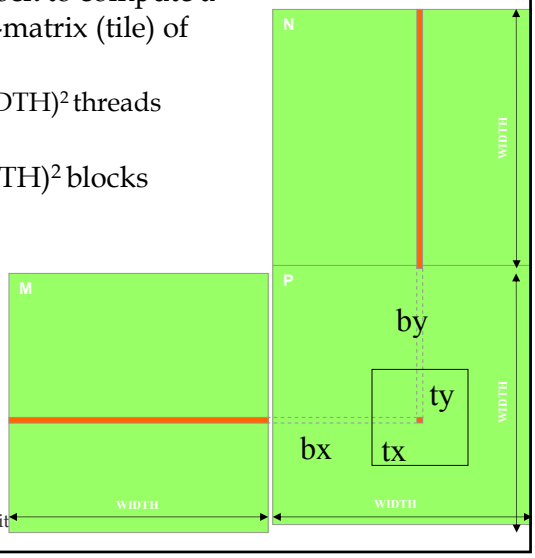
```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Step 5: Some Loose Ends (cont.)

```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
   int size = Mhost.width * Mhost.height * sizeof(float);
   cudaMemcpy(Mdevice.elements, Mhost.elements, size,
            cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
   int size = Mdevice.width * Mdevice.height * sizeof(float);
   cudaMemcpy(Mhost.elements, Mdevice.elements, size,
            cudaMemcpyDeviceToHost);
}
```


Step 6: Handling Arbitrary Sized Square Matrices

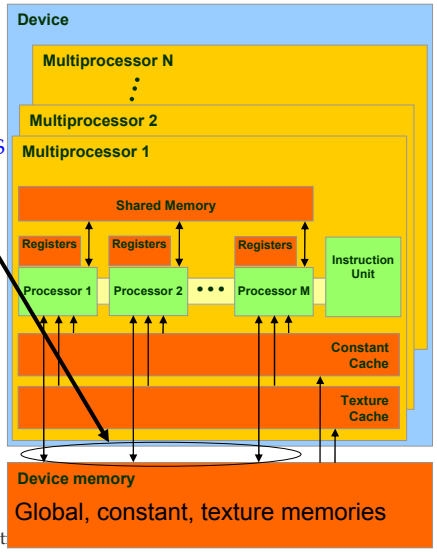
- Have each 2D thread block to compute a $(BLOCK\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(BLOCK\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/BLOCK\_WIDTH)^2$ blocks



You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!

How about performance?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code should run at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu

Developing High Performance Multithreaded Programs

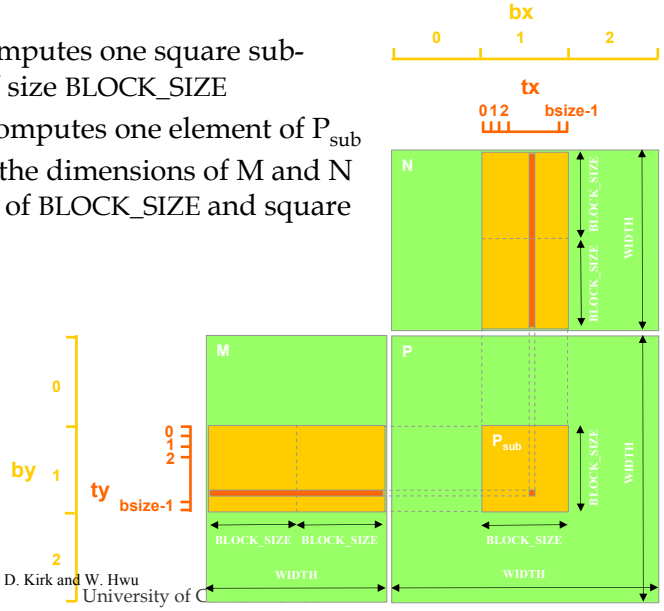
- Can be very complex, application dependent
- General Guidelines
  - Improving Parallelism (thread level).
    - #of thread blocks, #of threads in a block
  - Optimizing memory usage to achieve high memory bandwidth
    - Memory-level parallelism
    - Memory coalescing
    - Reduce memory accesses
  - Improving the instruction throughput
- Those goals may conflict.
  - E.g., increase number of insns to get higher parallelism
  - Additional hardware constraints due to registers, memory sizes, etc.

Idea # 1: Use Shared Memory to reuse global memory data

- Each input element is read by WIDTH threads.
- If we load each element into Shared Memory and have several threads use the local version, we can drastically reduce the memory bandwidth
  - Tiled algorithms

Tiled Multiply Using Thread Blocks

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape



Shared Memory Usage

- Each SMP has 16KB shared memory
  - Each Thread Block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
  - For BLOCK_SIZE = 16, this allows up to 8*512 = 4,096 pending loads
    - In practice, there will probably be up to half of this due to scheduling to make use of SPs.
  - The next BLOCK_SIZE 32 would lead to 2*32*32*4B= 8KB shared memory usage per Thread Block, allowing only up to two Thread Blocks active at the same time

First-order Size Considerations

- Each Thread Block should have a minimal of 192 threads
  - BLOCK_SIZE of 16 gives 16*16 = 256 threads
- A minimal of 32 Thread Blocks
  - A 1024*1024 P Matrix gives 64*64 = 4096 Thread Blocks
- Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - Memory bandwidth no longer a limiting factor

CUDA Code – Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N.width  / dimBlock.x,
             M.height / dimBlock.y);
```

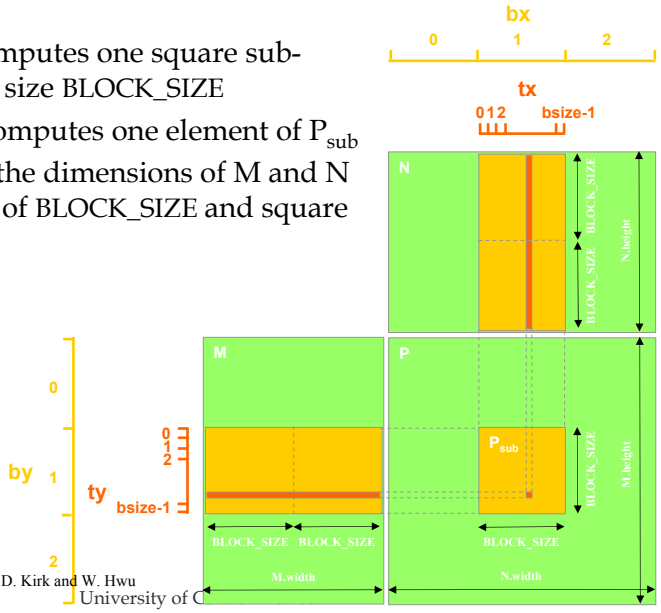
For very large N and M dimensions, one will need to add another level of blocking and execute the second-level blocks sequentially.

CUDA Code – Kernel Overview

```
// Block index
int bx = blockIdx.x;
int by = blockIdx.y;
// Thread index
int tx = threadIdx.x;
int ty = threadIdx.y;

// Pvalue stores the element of the block sub-matrix
// that is computed by the thread
float Pvalue = 0;

// Loop over all the sub-matrices of M and N
// required to compute the block sub-matrix
for (int m = 0; m < M.width/BLOCK_SIZE; ++m) {
     code from the next few slides };
```

Multiply Using Several Blocks

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape

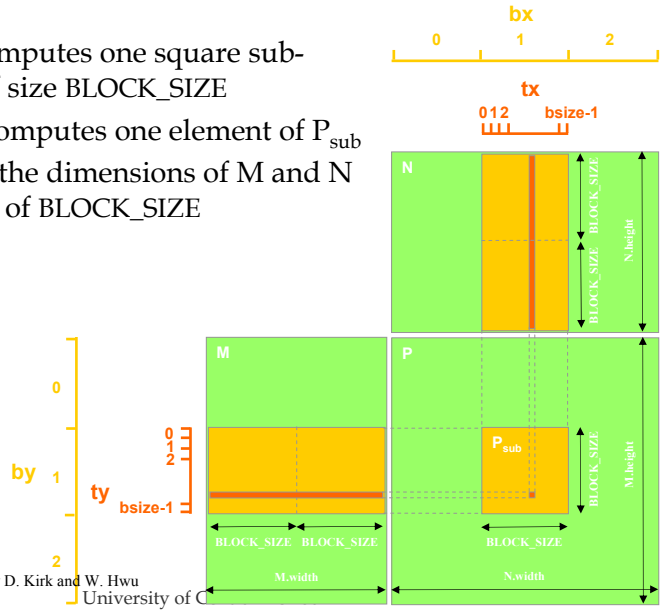


CUDA Code - Load Data to Shared Memory

```
// Get a pointer to the current sub-matrix Msub of M
Matrix Msub = GetSubMatrix(M, m, by);

// Get a pointer to the current sub-matrix Nsub of N
Matrix Nsub = GetSubMatrix(N, bx, m);

__shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];

// each thread loads one element of the sub-matrix
Ms[ty][tx] = GetMatrixElement(Msub, tx, ty);

// each thread loads one element of the sub-matrix
Ns[ty][tx] = GetMatrixElement(Nsub, tx, ty);
```

Multiply Using Several Blocks

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE

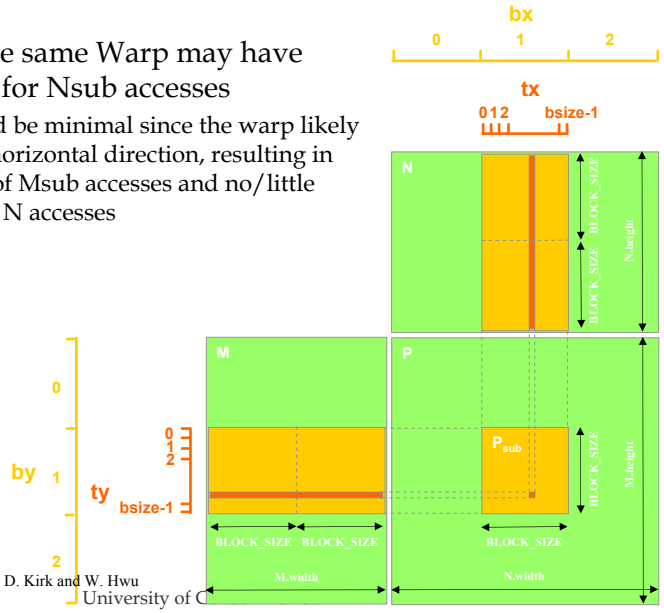


CUDA Code - Compute Result

```
// Synchronize to make sure the sub-matrices are loaded
// before starting the computation
__syncthreads();

// each thread computes one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
     Pvalue += Ms[ty][k] * Ns[k][tx];

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of M and N in the next iteration
__syncthreads();
```

Shared Memory Bank Conflicts

- Threads in the same Warp may have bank conflict for Nsub accesses
  - This should be minimal since the warp likely spans the horizontal direction, resulting in broadcast of Msub accesses and no/little conflict for N accesses



CUDA Code - Save Result

```
// Get a pointer to the block sub-matrix of P
Matrix Psub = GetSubMatrix(P, bx, by);

// Write the block sub-matrix to device memory;
// each thread writes one element
SetMatrixElement(Psub, tx, ty, Pvalue);
```

This code should run at about 45 GFLOPS

Idea # 2: Use unrolling & jam to reuse global memory data

- Each thread processes more than 1 element in P
- Multiple elements end with reusing the global memory data
- Which loop to unroll?
  - Which one does the CPU code favor?
  - Which one does the GPU code favor?
    - Can we take advantage of the cache for const memory?

Kernel Code for Unroll and Jam (with a unroll factor of 2 in outer loop)

```
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
   for (int i = 0; i < M.height; i += 2)
      for (int j = 0; j < N.width; ++j) {
         double sum1 = 0;
         double sum2 = 0;
         for (int k = 0; k < M.width; ++k) {
            double a1 = M.elements[i * M.width + k];
            double b = N.elements[k * N.width + j];
            double a2 = M.elements[(i + 1)* M.width + k];
            sum1 += a1 * b;
            sum2 += a2 * b;
         }
         P.elements[i * N.width + j] = sum1;
         P.elements[(i+1) * N.width + j] = sum2;
      }
}
```

Kernel Code for Unroll and Jam (with a unroll factor of 2 in inner loop)

```
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
   for (int i = 0; i < M.height; ++i)
      for (int j = 0; j < N.width; j+=2) {
         double sum1 = 0;
         double sum2 = 0;
         for (int k = 0; k < M.width; ++k) {
            double a = M.elements[i * M.width + k];
            double b1 = N.elements[k * N.width + j];
            double b2 = N.elements[k * N.width + j +1];
            sum1 += a * b1;
            sum2 += a * b2;
         }
         P.elements[i * N.width + j] = sum1;
         P.elements[i * N.width + j + 1] = sum2;
      }
}
```


Tradeoff

- Reduced loads
- High register usage (saving those in shared memory?)
- Reduced parallelism (i.e., number of thread blocks)