Lecture 7:

Caffe: GPU Optimization

boris. ginzburg@intel.com

Agenda

1. Practical intro to CUDA
   - Programming model
   - Memory model
   - Exercises
2. Caffe: CUDA part
   - SynchedMemory
   - Forward_gpu( );
3. Projects

CUDA: PRACTICAL INTRODUCTION

Based on David Kirk and Wen-mei W. Hwu tutorial
(2007, Univ. of Illinois, Urbana-Champaign)

Example1 : VecAdd

Based on NVIDIA_SAMPLES/SIMPLE/.vecAdd.cu

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main() {
   ...
  // Kernel invocation with N threads
  int numBlocks = 1;
  int threadsPerBlock = N;
  VecAdd<<< numBlocks, threadsPerBlock >>>(A, B, C);
   ...
}
```

Programming Model: Grid/Block/Thread

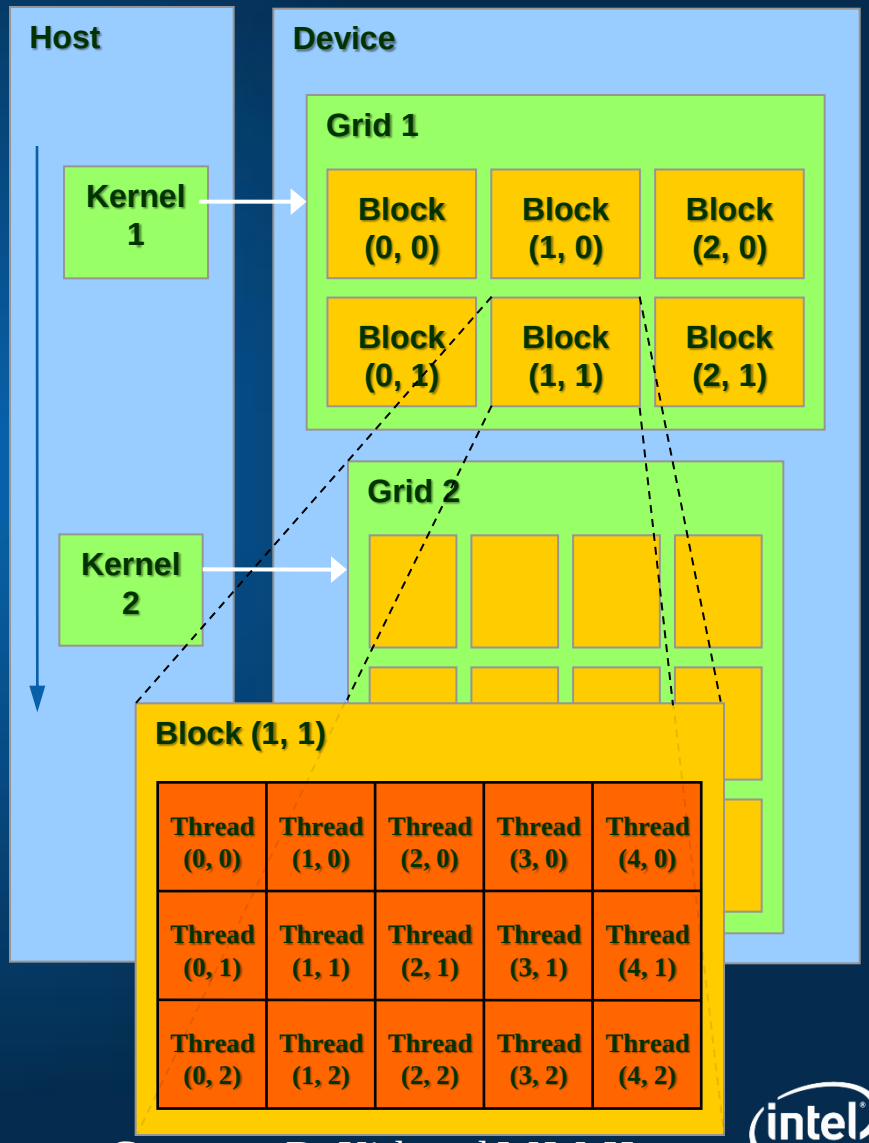
A kernel is executed as a grid of thread blocks

- All threads share global memory

A thread block is a batch of threads that can cooperate with each other by:

- Synchronizing their execution
- Efficiently sharing data through a low latency shared memory

Two threads from two different blocks cannot cooperate



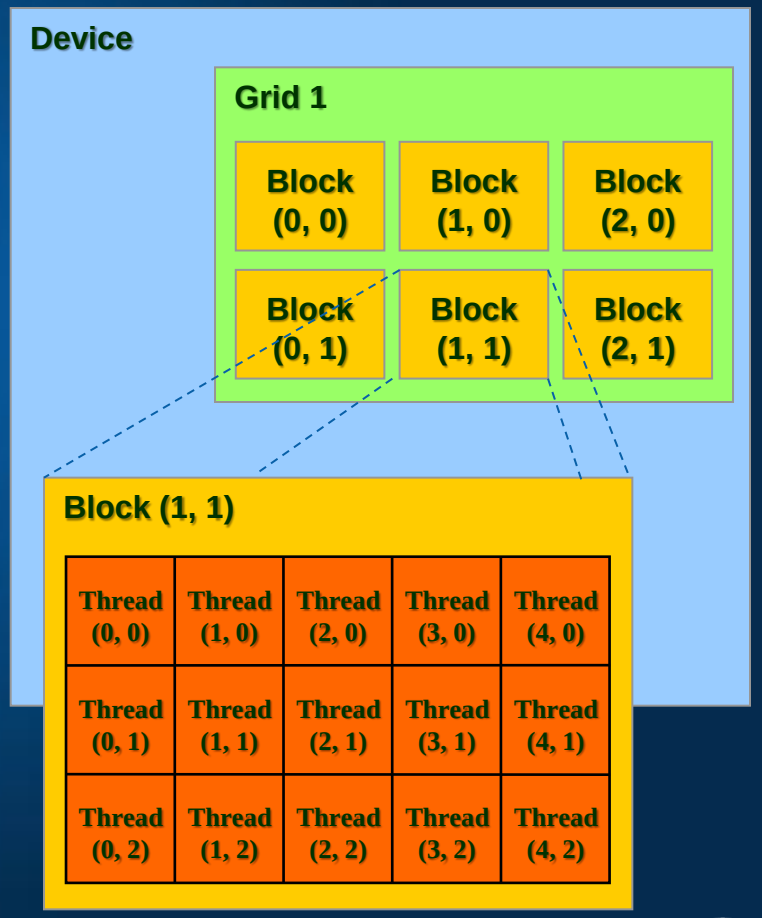
Courtesy: D. Kirk and W.W. Hwu

Programming Model: 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

- Max # of theads/block=1024



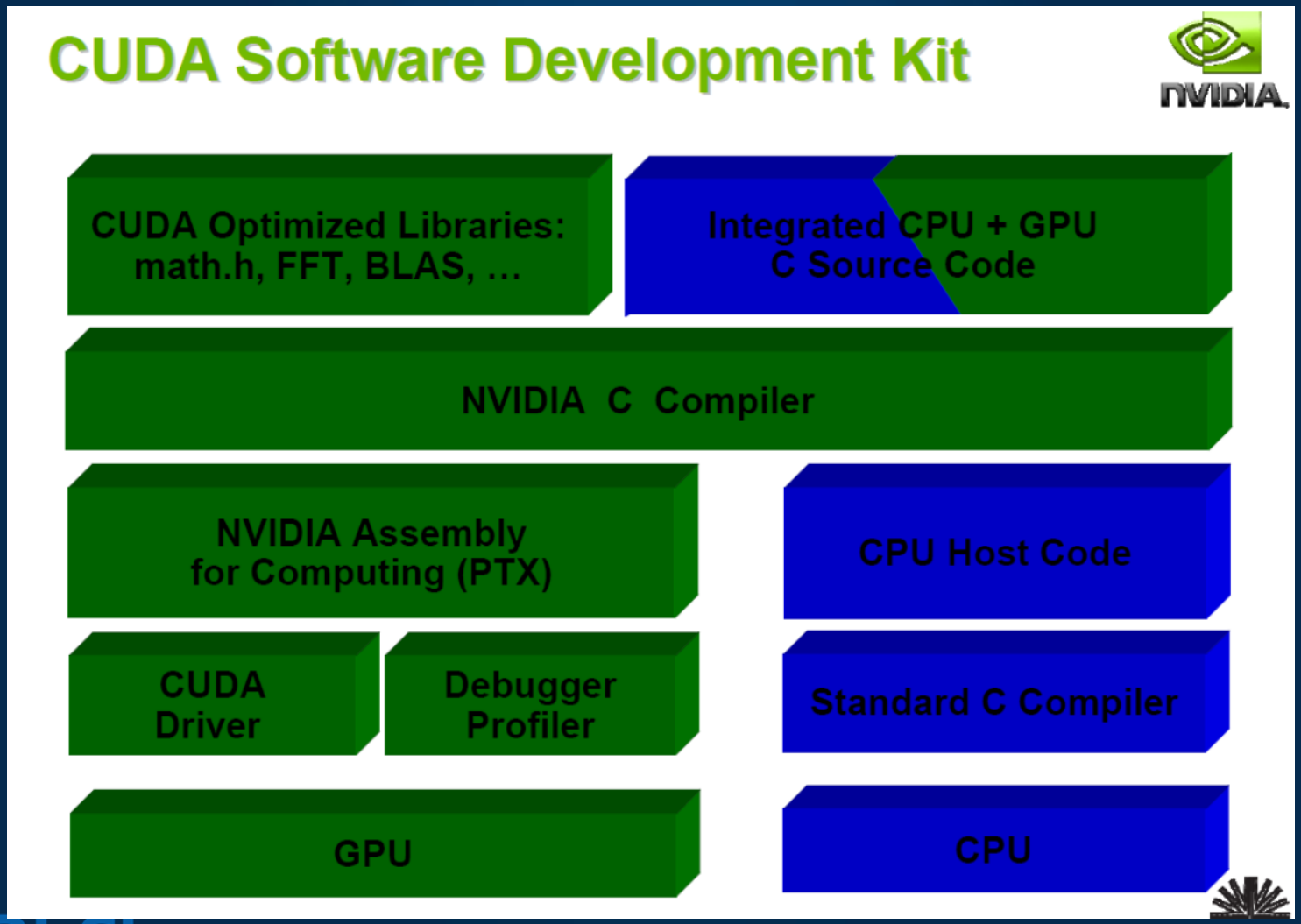
Courtesy: D. Kirk and W.W. Hwu

Calling a Kernel Function – Thread Creation

A kernel function is called with an execution configuration:

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


dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
KernelFunc<<<DimGrid,DimBlock>>>(...);
```

CUDA SDK


CUDA Software Development Kit
NVIDIA
CUDA Optimized Libraries: math.h, FFT, BLAS, …
Integrated CPU + GPU C Source Code
NVIDIA C Compiler
NVIDIA Assembly for Computing (PTX)
CPU Host Code
CUDA Driver
Debugger Profiler
Standard C Compiler
GPU
CPU

CUDA Makefile Example

```
GCC := g++
NVCC := nvcc -ccbin $(GCC)
CCFLAGS   := -g
NVCCFLAGS   := -m64 -g -G
LDFLAGS   :=
ALL_CCFLAGS := $(NVCCFLAGS) $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_LDFLAGS := $(ALL_CCFLAGS) $(addprefix -Xlinker ,$(LDFLAGS))
# Common includes and paths for CUDA
INCLUDES  := -I../common/inc
LIBRARIES :=
# CUDA code generation flags
GENCODE_SM30   := -gencode arch=compute_30,code=sm_30
GENCODE_SM50   := -gencode arch=compute_50,code=sm_50
GENCODE_FLAGS   :=  $(GENCODE_SM30)  $(GENCODE_SM50)
```

CUDA Makefile Example – cont.

```
# Target rules
all:  vectorAdd

vectorAdd.o:vectorAdd.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

vectorAdd: vectorAdd.o
	$(NVCC) $(ALL_LDFLAGS) $(GENCODE_FLAGS) -o $@ $+ $(LIBRARIES)

run: all
	./vectorAdd

clean:
	rm -f vectorAdd vectorAdd.o
```

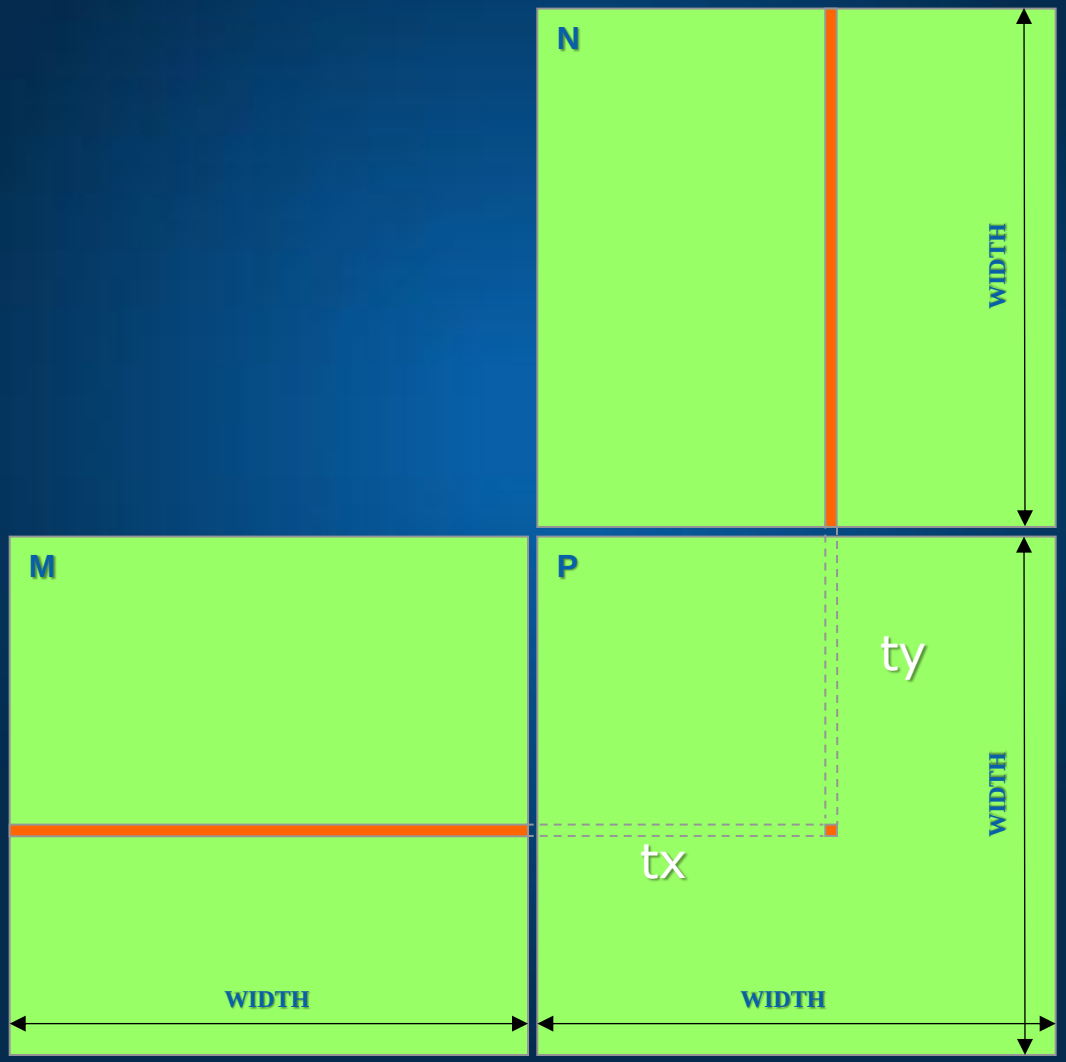
Example 2: Matrix Add

```
// Kernel definition
__global__ void MatAdd(floatA[N][N], floatB[N][N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main() {
   ...
  // Kernel invocation with one block of N * N * 1 threads
  int     numBlocks = 1;
  dim3 threadsPerBlock(N, N);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
}
```

Example 2: Matrix Add with Blocks

```
// Kernel definition
__global__ void  MatAdd(floatA[N][N], floatB[N][N], floatC[N][N]) {
    int  i = blockIdx.x * blockDim.x + threadIdx.x;
    int  j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i < N && j < N)   C[i][j] = A[i][j] + B[i][j];
}
int  main() {
   ...
   // Kernel invocation
  dim3  threadsPerBlock(16, 16);
  dim3  numBlocks( N / threadsPerBlock.x, N / threadsPerBlock.y);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
}
```

Example 3: Matrix Multiply

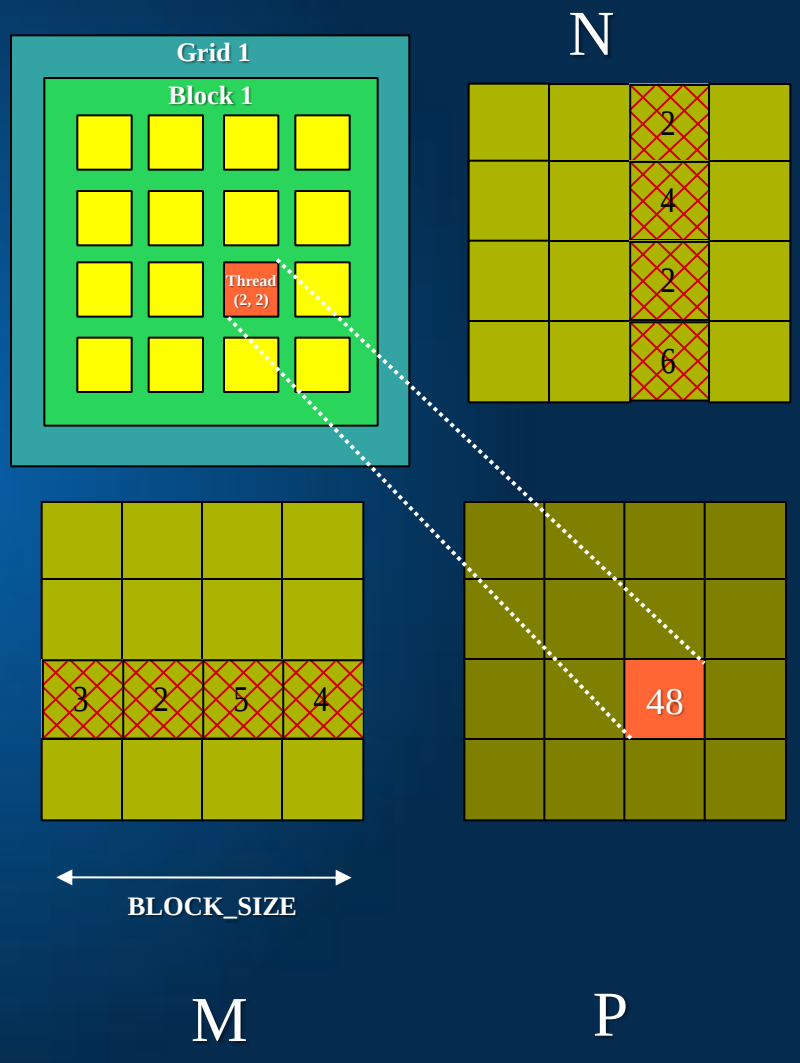


Example 3: Matrix Multiply

Block of threads compute matrix P:

- Each thread computes one element of P:
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply-add for each pair of M and N elements

Size of matrix limited by max # of threads per thread block: 1024



Example 3: Matrix Multiply – cont.

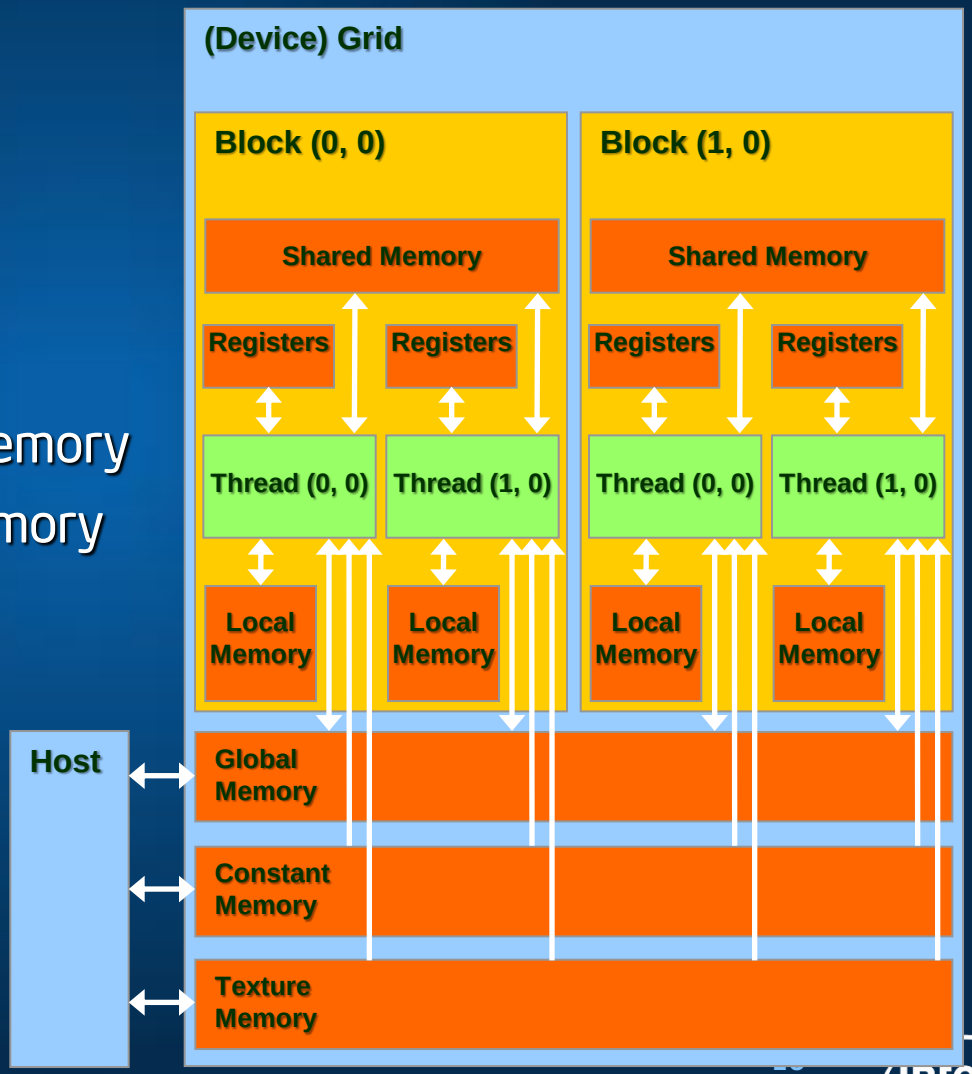
```
// Matrix multiplication kernel – thread specification

__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float z = 0; // accumulator for P
    for (int k = 0; k < W; ++k)    {
        z += M [ ty * W + k ]  *  N[ k * W + tx ];
    }

    // Write z to device memory;
    P [ ty * W + tx ] = z;
}
```

Memory model

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 memory

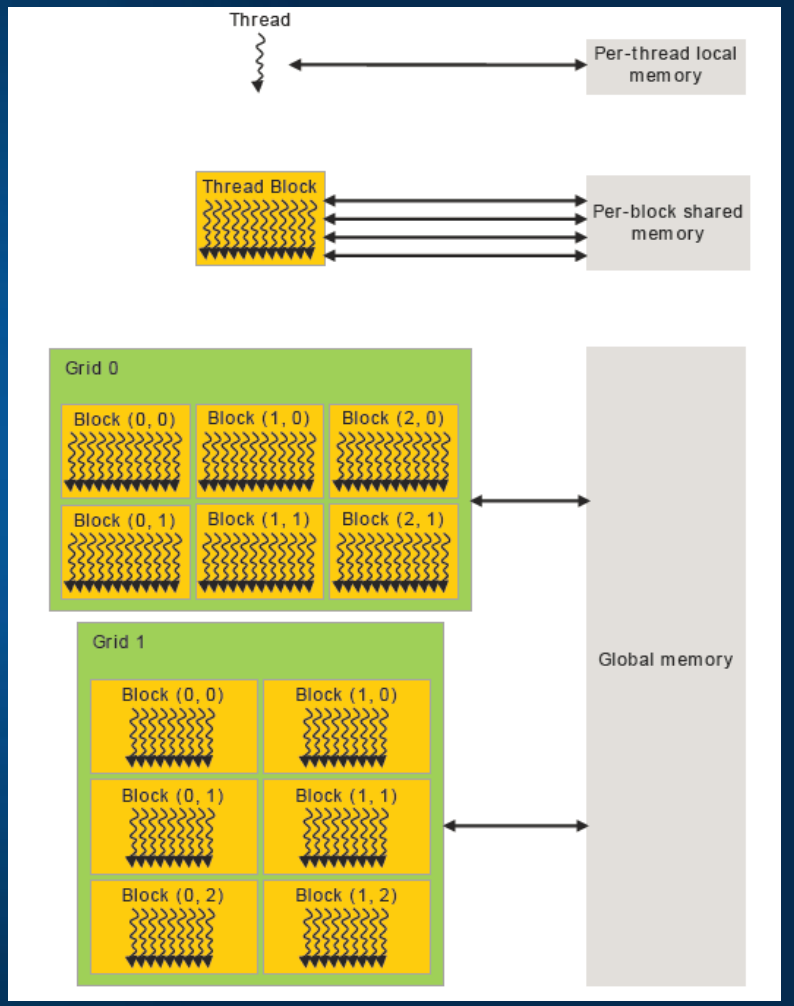


CUDA Shared Memory

Threads inside Thread Block can share some block of local memory.

Access to Shared Memory is much faster than to global memory.

The max amount of Shared Memory = 48K



Example 4: Matrix Multiply with Shared Memory

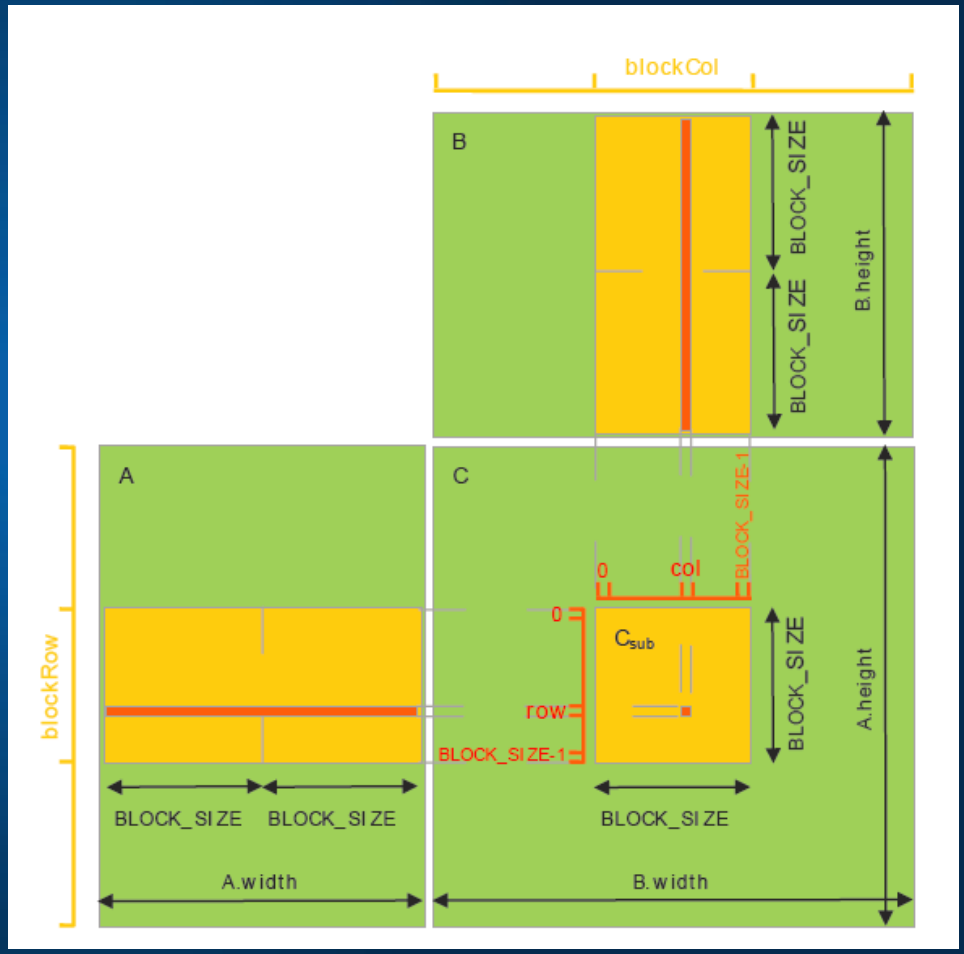
Thread block computes one square sub-matrix C_{sub} (BxB)

- each thread computes one element of C_{sub}.

For this rectangular bands (WxB) are divided into sub-blocks (BxB).

- Each sub-block is copied from Global memory into Shared Memory.
- Each Thread [x,y] computes partial product, and adds it to C[x,y].

This saves a lot of READs from global memory



Example 4: Matrix Multiply with Shared Memory

```
__global__ void matrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE]; // shared memory array As
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE]; // shared memory array Bs

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

    int aBegin = wA * BLOCK_SIZE * by; // first index of A
    int aEnd   = aBegin + wA - 1;      // last ndex of A
    int aStep  = BLOCK_SIZE;           // step to iterate through A
    int bBegin = BLOCK_SIZE * bx;      // first index of B
    int bStep  = BLOCK_SIZE * wB;      // step to iterate through  B
```

Example 4: Matrix Multiply with Shared Memory –cont.

```
float Csub = 0;                    // Csub to store the element of the result
// Loop over all the sub-matrices of A and B
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)  {
    // Load the sub-matrices from device memory to shared memory;
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads(); // Synchronize to make sure the matrices are loaded
    // Multiply the two sub-matrices together;
    for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();  // Synchronize to make sure that computation is done
}
// Write the result to device memory;
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}
```

Example 4: Matrix Multiply with Shared Memory –cont.

```
float *d_A, *d_B, *d_C;
cudaMalloc((void **) &d_A, mem_size_A);
cudaMalloc((void **) &d_B, mem_size_B);
cudaMalloc((void **) &d_C, mem_size_C);
 // Copy host memory to device
cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);
int block_size = 32;
dim3 threads(block_size, block_size);
dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);
matrixMulCUDA<32><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
cudaDeviceSynchronize();
// Copy result from device to host
cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);
cudaFree(d_A);
…
```

CUDA 6.0 – Unified Memory+cublasXT

Unified CPU-GPU memory:

- Before: Data that is shared between the CPU and GPU must be allocated in both memories, and explicitly copied between them by the programmer
- Now: a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. Data allocated in Unified Memory automatically *migrates* between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.

cublasXT

- new BLAS multi-GPU library that automatically scales performance across up to 8 GPUs /node; supporting workloads up to 512GB).
- The re-designed FFT library scales up to 2 GPUs/node

Exercises

1. Implement 2D_correlation_CUDA
2. Implement ConvLayer_forward_CUDA

Caffe: GPU

GPU support in Caffe is based on:

1. *SynchedMemory*
   - "transparent" memory moving between CPU and GPU
2. GPU implementation for each layer
   - ConvolutionLayer::Forward_gpu( )
   - ConvolutionLayer::Backward_gpu( )

Caffe: SynchedMemory

```
class Blob {
  public:
    Blob()
    ....
  protected:
   shared_ptr<SyncedMemory> data_;
   shared_ptr<SyncedMemory> diff_;

}
```

Caffe: SynchedMemory

```
class SyncedMemory {
  public:
    SyncedMemory()
        : cpu_ptr_(NULL), gpu_ptr_(NULL), size_(0), head_(UNINITIALIZED) {}
    explicit SyncedMemory(size_t size)
        : cpu_ptr_(NULL), gpu_ptr_(NULL), size_(size), head_(UNINITIALIZED) { }
    ~SyncedMemory();
    const void* cpu_data();
    const void* gpu_data();
    void* mutable_cpu_data();
    void* mutable_gpu_data();
    enum SyncedHead { UNINITIALIZED, HEAD_AT_CPU, HEAD_AT_GPU, SYNCED };
    SyncedHead head() { return head_; }
    size_t size() { return size_; }
```

Caffe: SynchedMemory

```
class SyncedMemory {
  ...
  private:
    void to_cpu();
    void to_gpu();
    void* cpu_ptr_;
    void* gpu_ptr_;
    size_t size_;
    SyncedHead head_;

};
```

ConvolutionalLayer:: Forward_gpu()

```
void ConvolutionLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>&
bottom,  vector<Blob<Dtype>*>* top) {
  const Dtype* bottom_data = bottom[0]->gpu_data();
  Dtype* top_data = (*top)[0]->mutable_gpu_data();
  Dtype* col_data = col_buffer_.mutable_gpu_data();
  const Dtype* weight = this->blobs_[0]->gpu_data();
  int weight_offset = M_ * K_;
  int col_offset = K_ * N_;
  int top_offset = M_ * N_;
  for (int n = 0; n < NUM_; ++n) {
        im2col_gpu( ...);
        for (int g = 0; g < GROUP_; ++g)
              caffe_gpu_gemm<Dtype>(..);
  }
  if (biasterm_)
        caffe_gpu_gemm<Dtype>();
}
```

CAFFE under the hood: BLAS

```
void caffe_cpu_gemm<float>(const CBLAS_TRANSPOSE TransA,  const CBLAS_TRANSPOSE TransB, const int
M, const int N, const int K,   const float alpha, const float* A, const float* B, const float beta, float* C) {
  int lda = (TransA == CblasNoTrans) ? K : M;
  int ldb = (TransB == CblasNoTrans) ? N : K;
  cblas_sgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha, A, lda, B,
      ldb, beta, C, N);
}
```

BLAS:

- ATLAS
- openBLAS
- MKL

CAFFE under the hood: BLAS

```
void caffe_gpu_gemm<float>(const CBLAS_TRANSPOSE TransA,  const CBLAS_TRANSPOSE TransB, const
int M, const int N, const int K,     const float alpha, const float* A, const float* B, const float beta,     float* C) {
  // Note that cublas follows fortran order.
  int lda = (TransA == CblasNoTrans) ? K : M;
  int ldb = (TransB == CblasNoTrans) ? N : K;
  cublasOperation_t   cuTransA =
                (TransA == CblasNoTrans) ? CUBLAS_OP_N : CUBLAS_OP_T;
  cublasOperation_t   cuTransB =
                (TransB == CblasNoTrans) ? CUBLAS_OP_N : CUBLAS_OP_T;

  CUBLAS_CHECK(  cublasSgemm (Caffe::cublas_handle(), cuTransB, cuTransA,
                 N, M, K, &alpha, B, ldb, A, lda, &beta, C, N));
}
```

Projects

1. Re-implement caffe_gpu based on CuBLASXT (CUDA 6.0).
2. Direct implementation of caffe convolutional layer using CUDA 6.0 (check cuda-convnet2!)