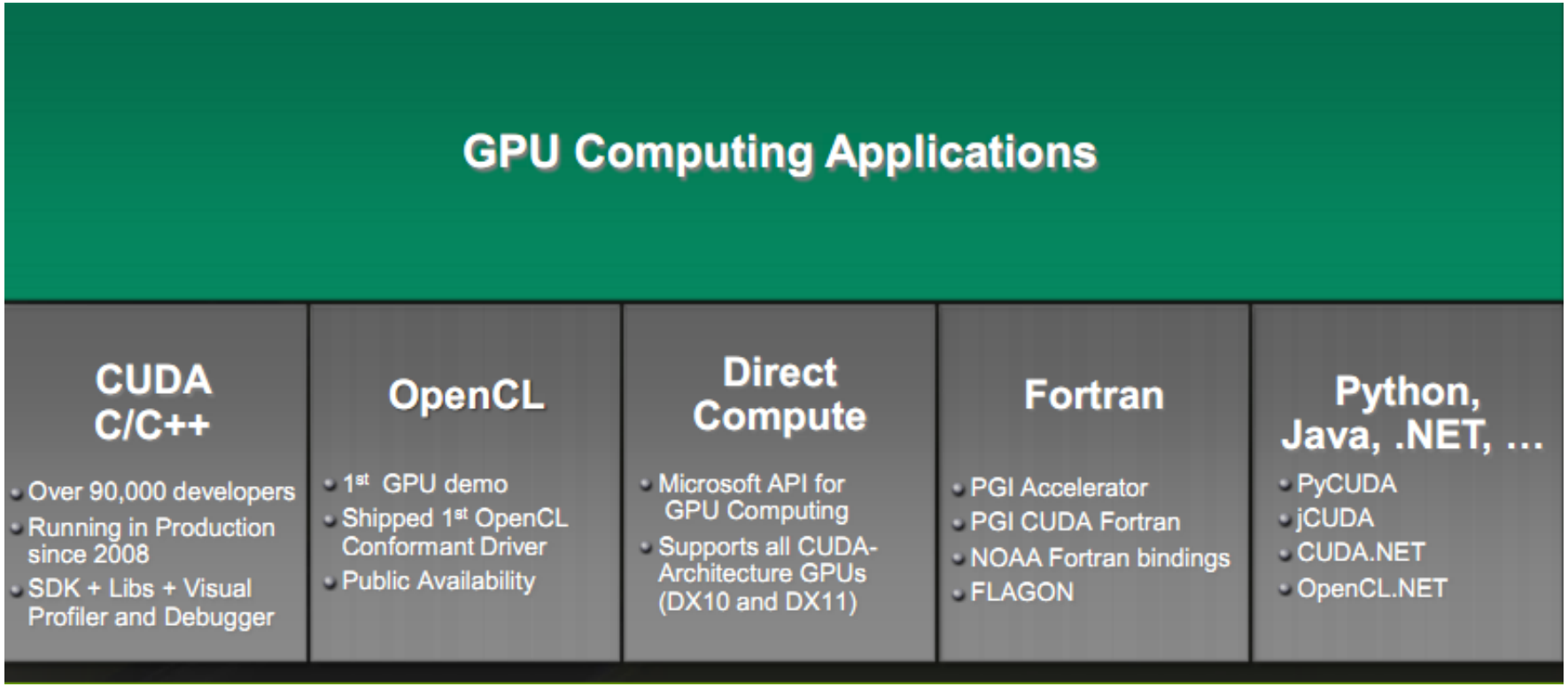
CUDA Programming

Sources

- http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf
- Slides by David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009, ECE498AL, University of Illinois, Urbana Champaign

GPU Language Options



Languages vs. Bindings

Solution	Approach
CUDA C / C++	Language Integration Device-Level API
PGI Accelerator PGI CUDA Fortran	Auto Parallelizing Compiler Language Integration
CAPS HMPP	Auto Parallelizing Compiler
OpenCL	Device-Level API
DirectCompute	Device-Level API
PyCUDA	API Bindings
CUDA.NET, OpenCL.NET, jCUDA	API Bindings
…	…

GPU vs. CPU

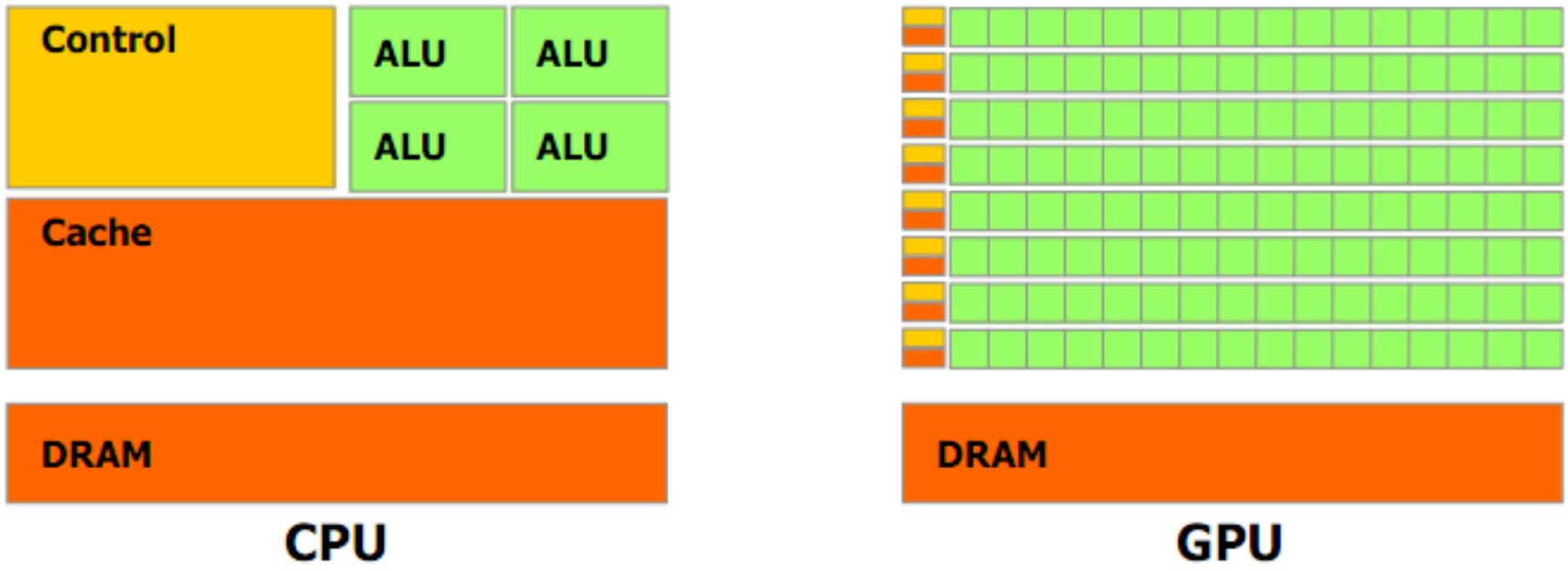
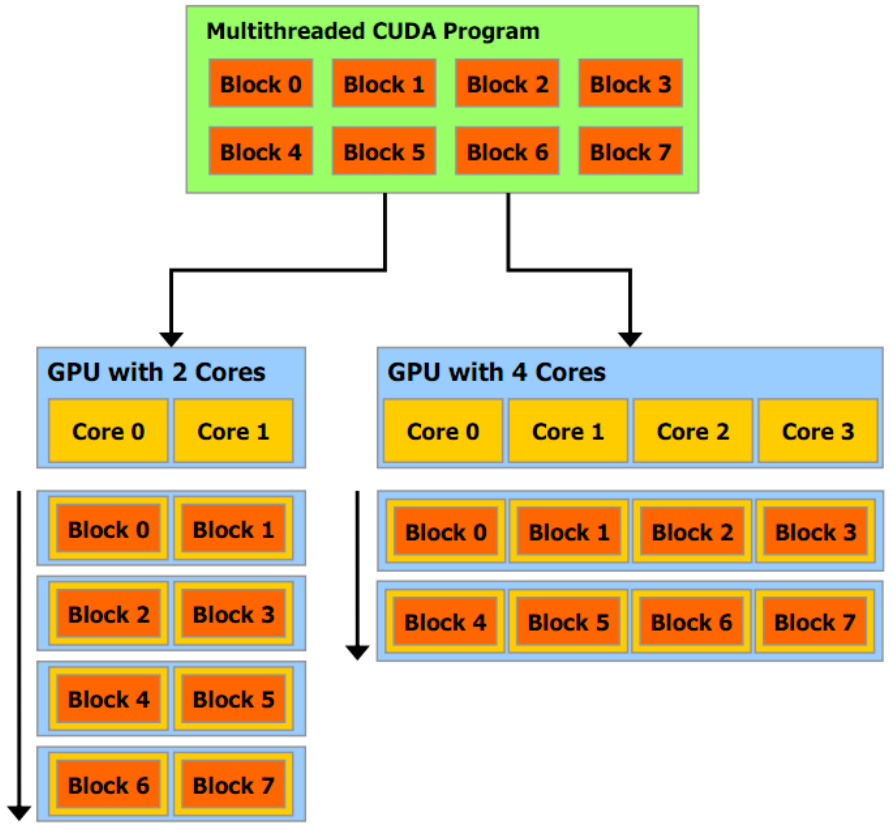


Figure 1-2. The GPU Devotes More Transistors to Data Processing

Automatic Scalability



A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4. Automatic Scalability

Kernel Procedures

- A kernel is defined using the

  __global__

  declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a

  new <<<…>>>

  execution configuration syntax.

- Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in variable:

  threadIdx

Simple Vector Addition Example

```
// 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 from CPU (1 is number of “blocks”)

VecAdd<<<1, N>>>(A, B, C);
}
```

Blocks vs. Threads

- A **block** consists of up to 1024 threads, organized in 1, 2, or 3 dimensions.
- A typical choice is 256 = 16x16 threads per block.
- Blocks themselves are organized as a 1 or 2 dimensional **grid**.
- Blocks are mapped onto the GPU SMs (Streaming Multiprocessors).
- Each SM has multiple cores (e.g. 32).

Warps

- Warps are occasionally mentioned.
- They are a *physical implementation* technique, used for scheduling groups of threads, not part of the CUDA specification proper.

- Threads in a **half-warp** execute SIMD.

- Code should not depend on any specific order of warps.

A 2D Grid of 2D Blocks of Threads

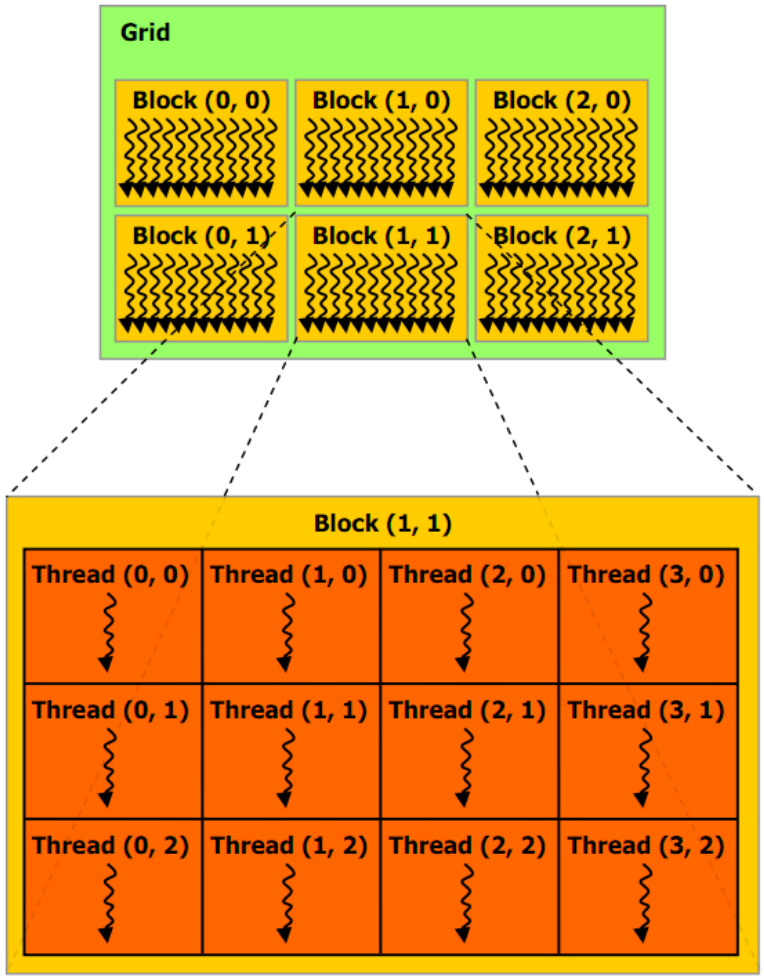
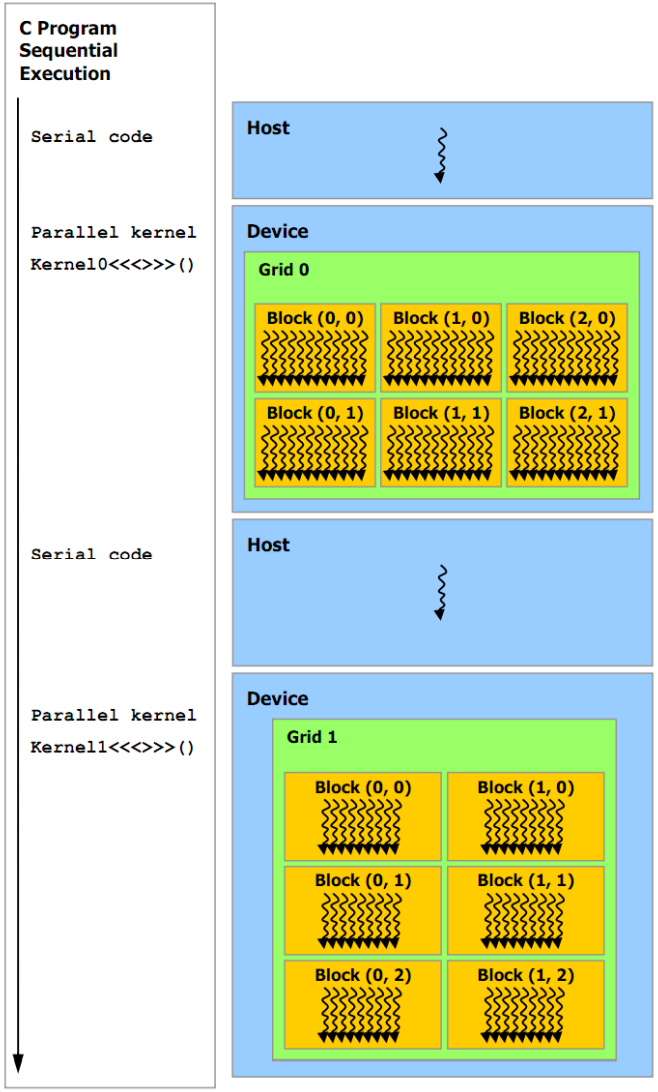


Figure 2-1. Grid of Thread Blocks

CPU vs. GPU Execution



Serial code executes on the host while parallel code executes on the device.

Thread Parallelism

- Threads are designed to execute in parallel or in any order.
- Threads can coordinate only **inside** a block.
- **__syncthreads()** is a built-in barrier for threads inside a block.

threadIdx Variable

- The index of a thread and its thread ID relate to each other in a straightforward way:
  - For a **one-dimensional block**, they are the same;
  - For a **two-dimensional block** of size (D_x, Dy), the thread ID of a thread of index (x, y) is $x + yD_x$;
  - For a **three dimensional block** of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is $x + yD_x + zD_xD_y$.

Example of 2D Thread Organization: Matrix Addition

```
// Kernel definition

__global__ void MatAdd(float A[N][N], float B[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);
}
```

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 (compiler generates both types of code)

Pointers

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:

```
__global__ void KernelFunc(float* ptr)
```

  - Obtained as the address of a global variable:

```
float* ptr = &GlobalVar;
```

dim3 and unit3 struct types

unit3 and dim3 – can be considered *essentially* as CUDA-defined structures of unsigned integers: x, y, z, i.e.

```
struct unit3 { x; y; z; };
struct dim3 { x; y; z; };
```

Unassigned structure components automatically set to 1.

Example:

```
dim3 grid(16, 16);                  // Grid -- 16 x 16 blocks

dim3 block(32, 32);                 // Block -- 32 x 32 threads

myKernel<<<grid, block>>>(...);
```

blockIdx and blockDim Variables

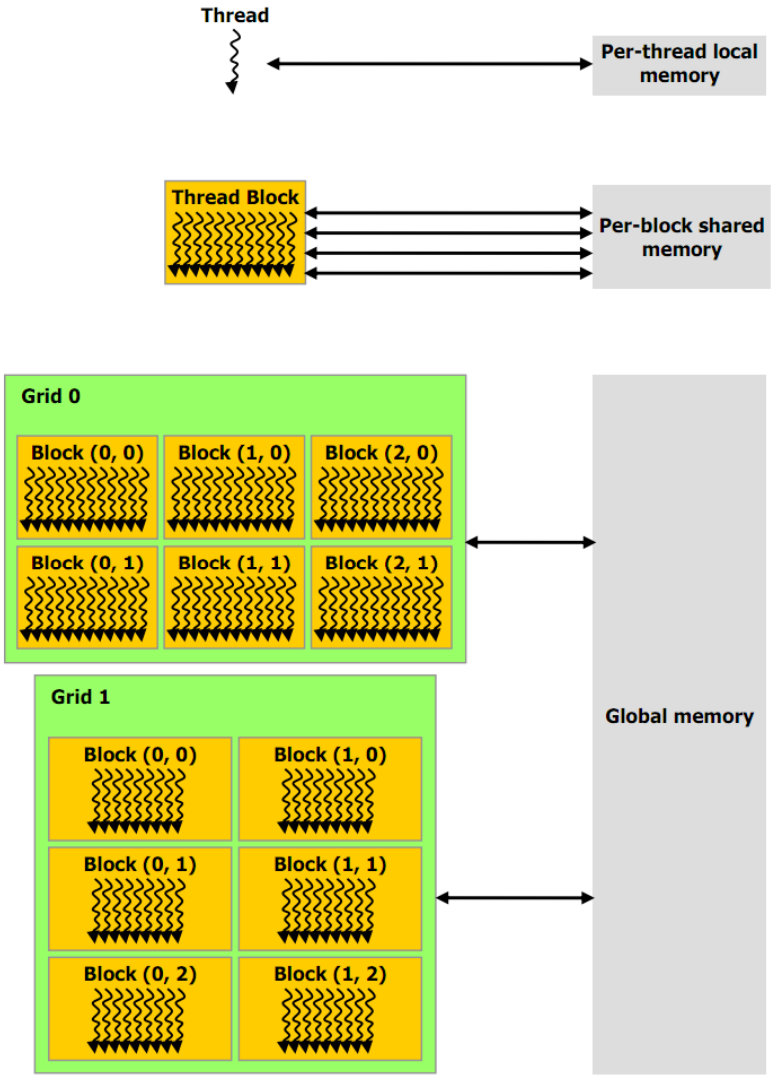
- blockIdx is similar to threadIdx, except that the number of dimensions is limited to 2.

- blockDim variable indicates the number of dimensions.

Matrix Add using Multiple Blocks

```
// Kernel definition

__global__ void MatAdd(float A[N][N], float B[N][N], float C[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]
}

// invocation
int main()
{
dim3 threadsPerBlock(16, 16);

dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);

   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Memory Hierarchy



cudaMalloc

- Allocates memory on device.

- Similar to C malloc, but unlike C malloc, result is passed by reference through a variable.

```
float* d_A;

cudaMalloc(&d_A, size);
```

cudaMemcpy

- Like C memcpy, except copy is between main memory and device memory.

```
// Copy vectors from host memory to device memory

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```

Variations

- cudaMallocPitch() [pitch = stride]
- cudaMalloc3D()
- cudaMemcpy2D()
- cudaMemcpy3D()

2D Dynamic Array Example

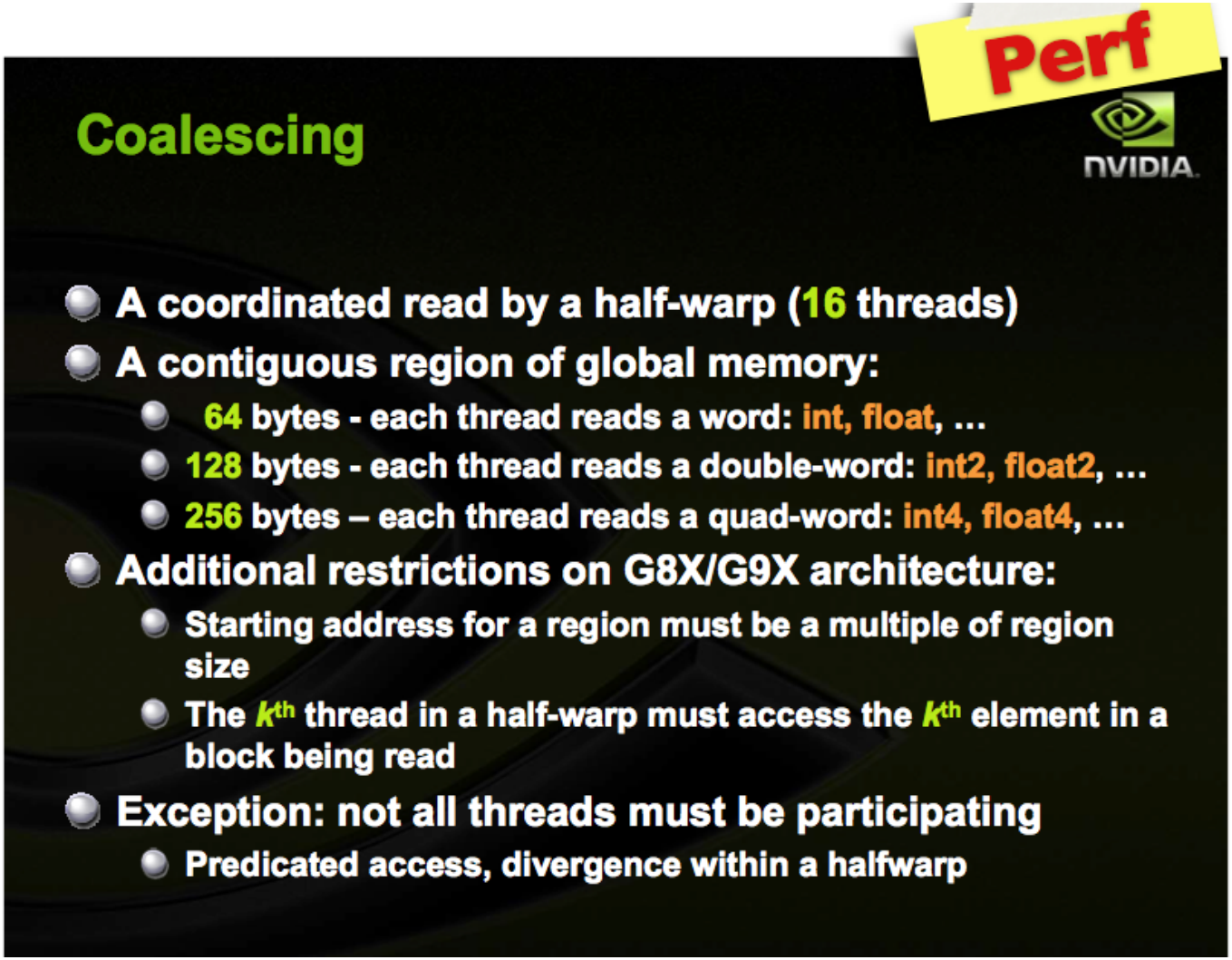
```
// Host code
int width = 64, height = 64;

float* devPtr; size_t pitch;

cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);

MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code

__global__ void MyKernel(float* devPtr, size_t pitch, int width, int height)
{
for (int r = 0; r < height; ++r) {
    float* row = (float*)((char*)devPtr + r * pitch);

    for (int c = 0; c < width; ++c) {
        float element = row[c];
    }
}
}
```

It’s All About Memory

- Without optimizing memory accesses, the maximum performance of a CUDA system will not be attained.

- Global memory access can take 400-600 clock cycles

“Coalescing” = Coordinated Global Access

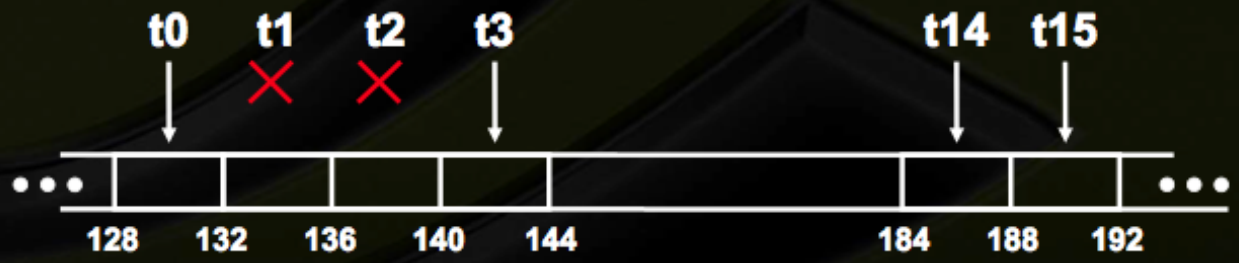


Coalesced Access: Reading floats

Perf

NVIDIA

All threads participate

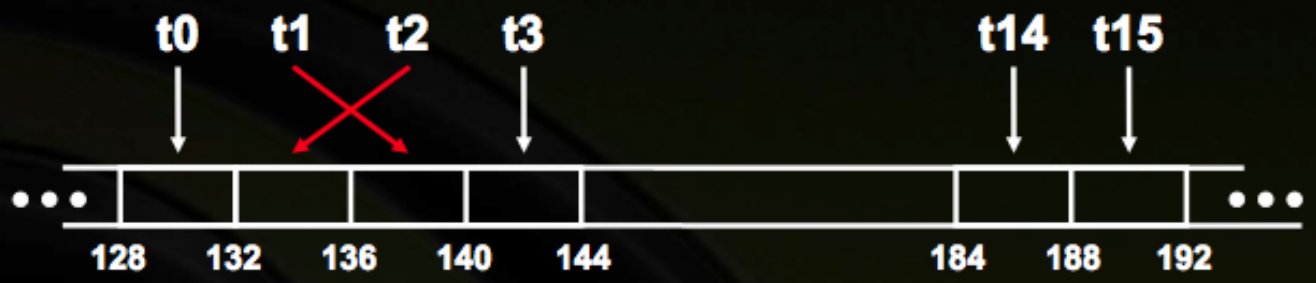


Some Threads Do Not Participate

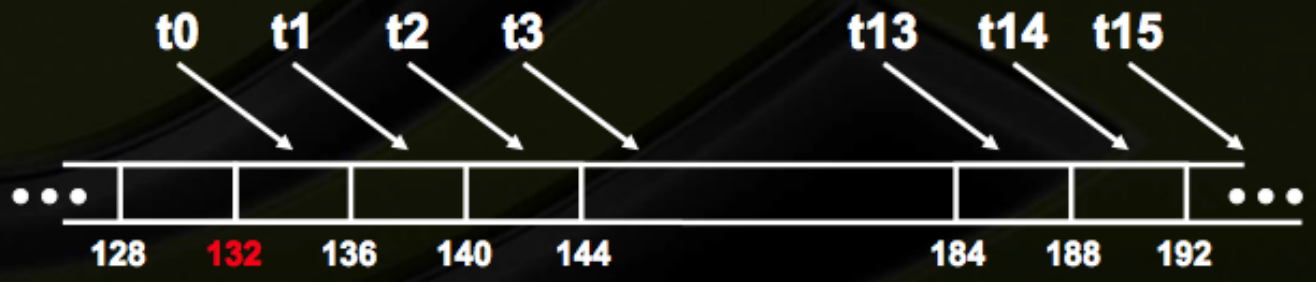
Uncoalesced Access: Reading floats

Perf

NVIDIA



Permuted Access by Threads



Misaligned Starting Address (not a multiple of 64)

Coalescing: Timing Results

Perf

NVIDIA

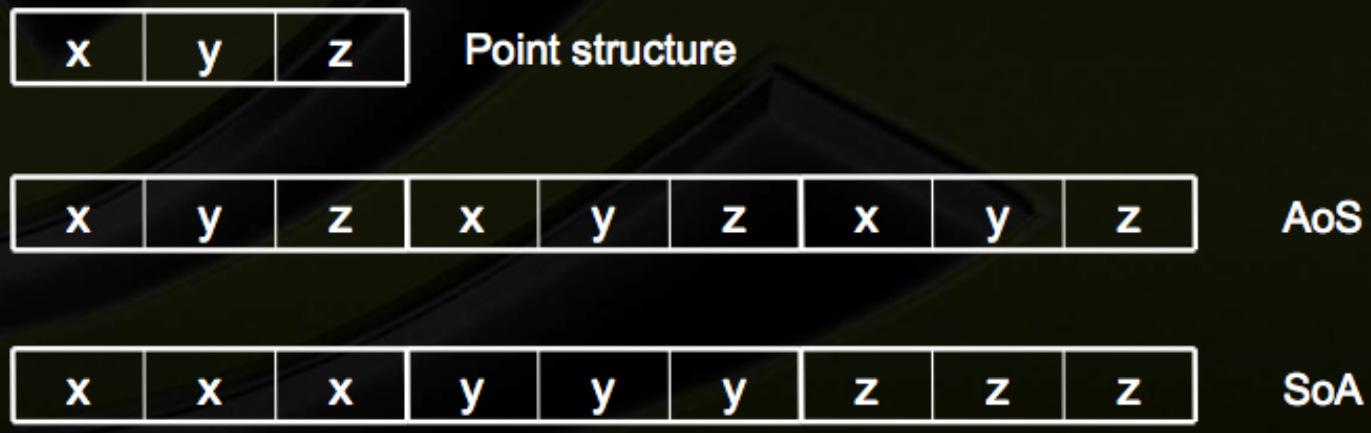
- Experiment on G80:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads:
  - 356µs – coalesced
  - 357µs – coalesced, some threads don't participate
  - 3,494µs – permuted/misaligned thread access

Perf

Coalescing:
Structures of size ≠ 4, 8, 16 Bytes

- Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)

- If SoA is not viable:
  - Force structure alignment: __align(X), where X = 4, 8, or 16
  - Use SMEM to achieve coalescing



Shared Memory

- “Shared” means among threads in a block.
- Much faster than global memory
- From the device’s viewpoint, global memory is like I/O compared to shared memory.

Matrix Multiply using Shared Memory

- Want to “cache” data in shared memory so that it can be reused.
- Several elements in a product rely on common elements in rows and columns.
- But can’t generally store all of a row or column in shared memory at one time.
- Idea: Use a **block-oriented** multiply.
- Threads copy blocks from global into shared, then use the data.

Narrative Description

Each thread block is responsible for computing one square sub-matrix Csub of C and each thread within the block is responsible for computing one element of Csub.

Csub is equal to the product of two rectangular matrices: the sub-matrix of A of dimension (A.width, BLOCK_SIZE) that has the same line indices as Csub, and the submatrix of B of dimension (BLOCK_SIZE , A.width) that has the same column indices as Csub.

In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension BLOCK_SIZE as necessary and Csub is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product.

Each thread accumulates the result of each of these products into a register and once done writes the result to global memory

Block-Oriented Matrix Multiply

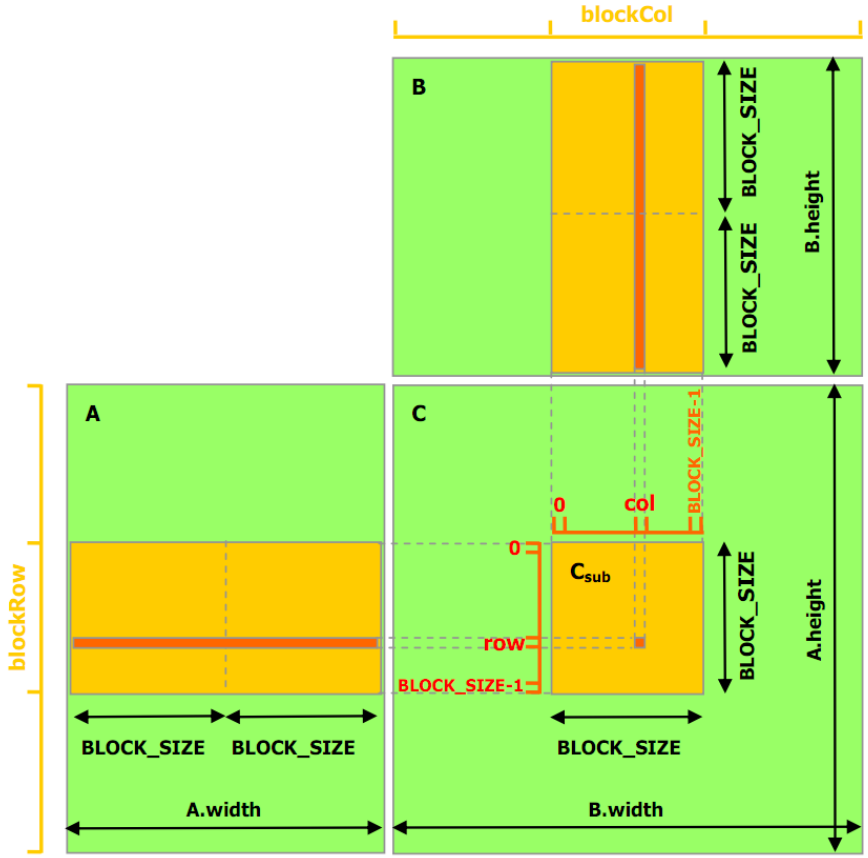


Figure 3-2. Matrix Multiplication with Shared Memory

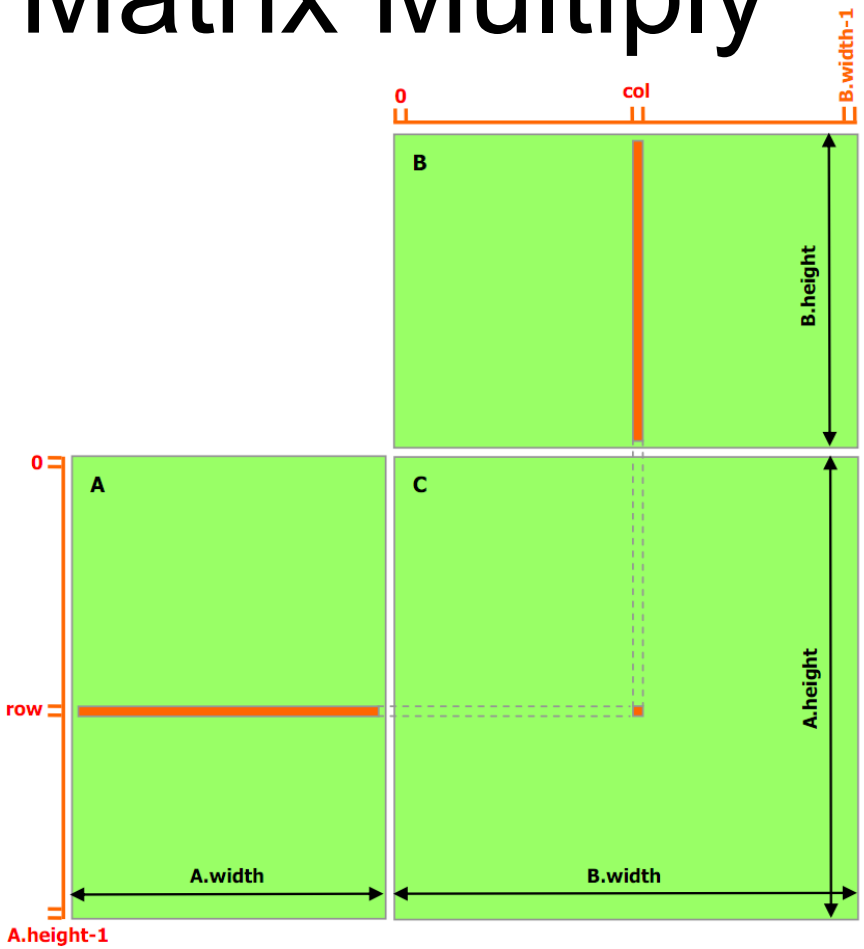
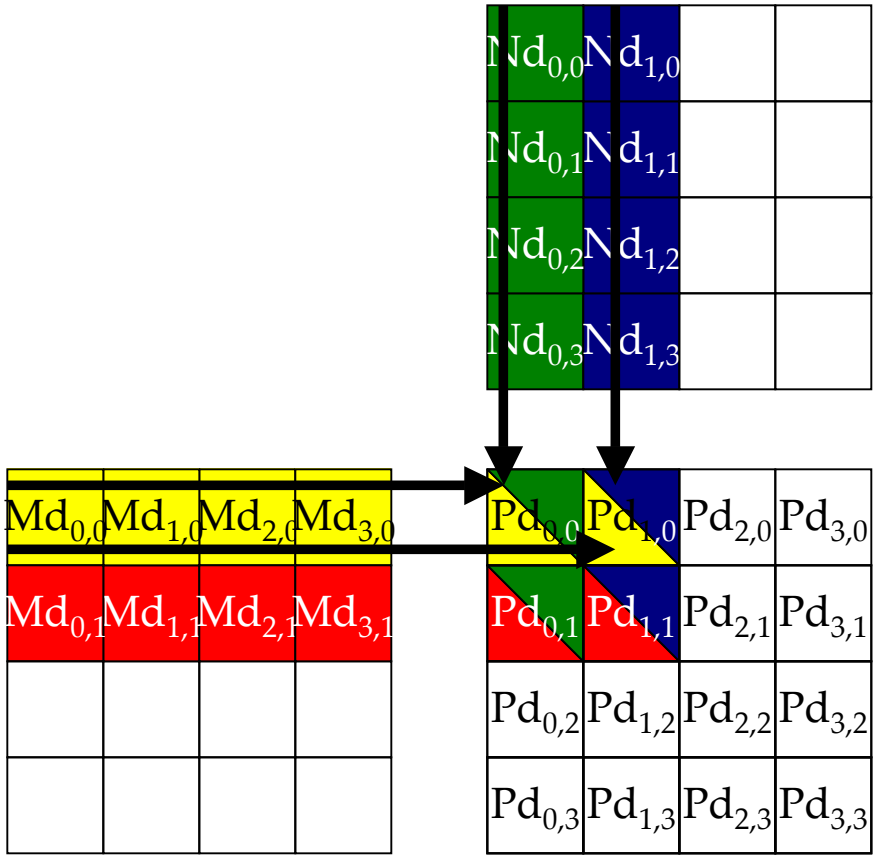
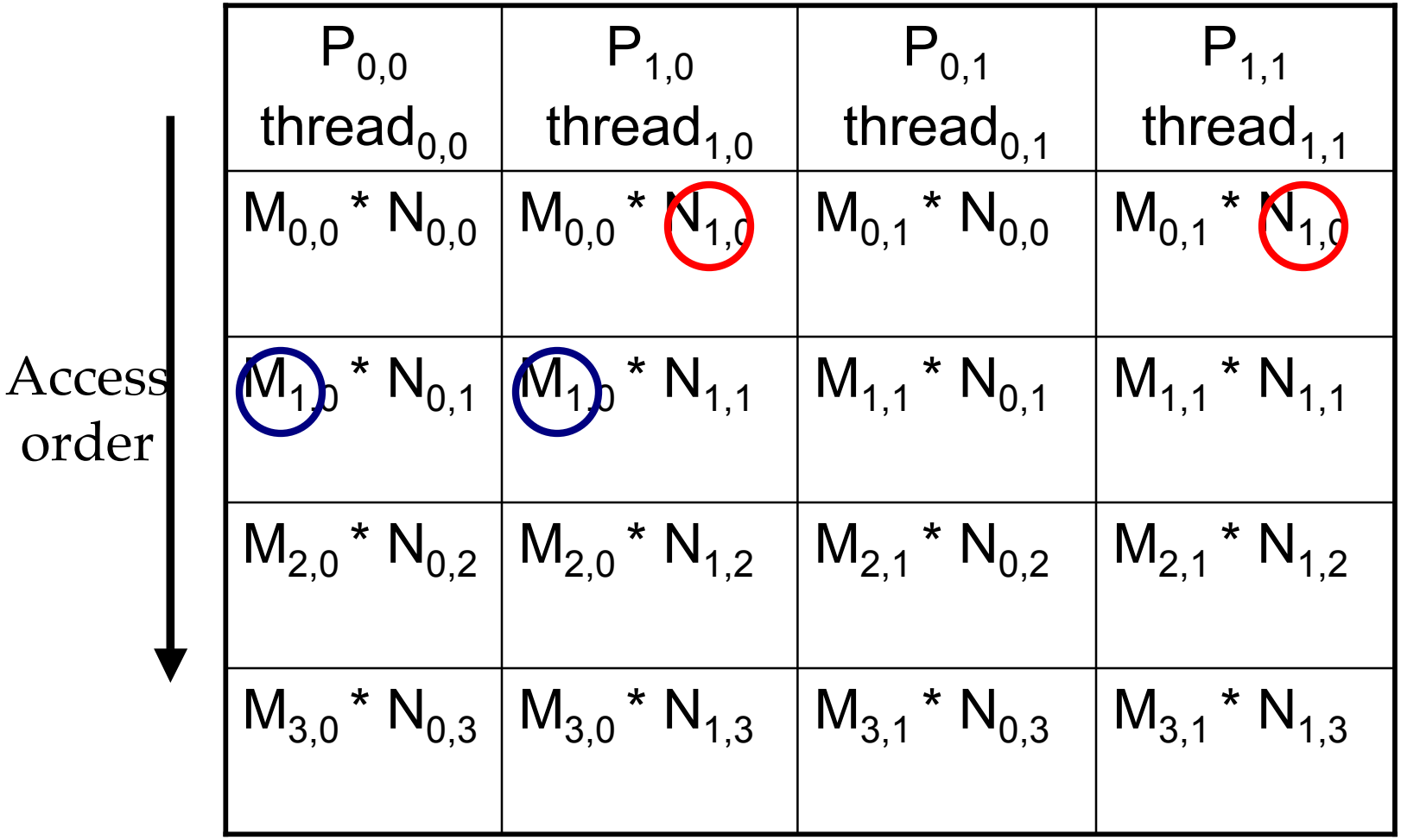


Figure 3-1. Matrix Multiplication without Shared Memory

A Small Example

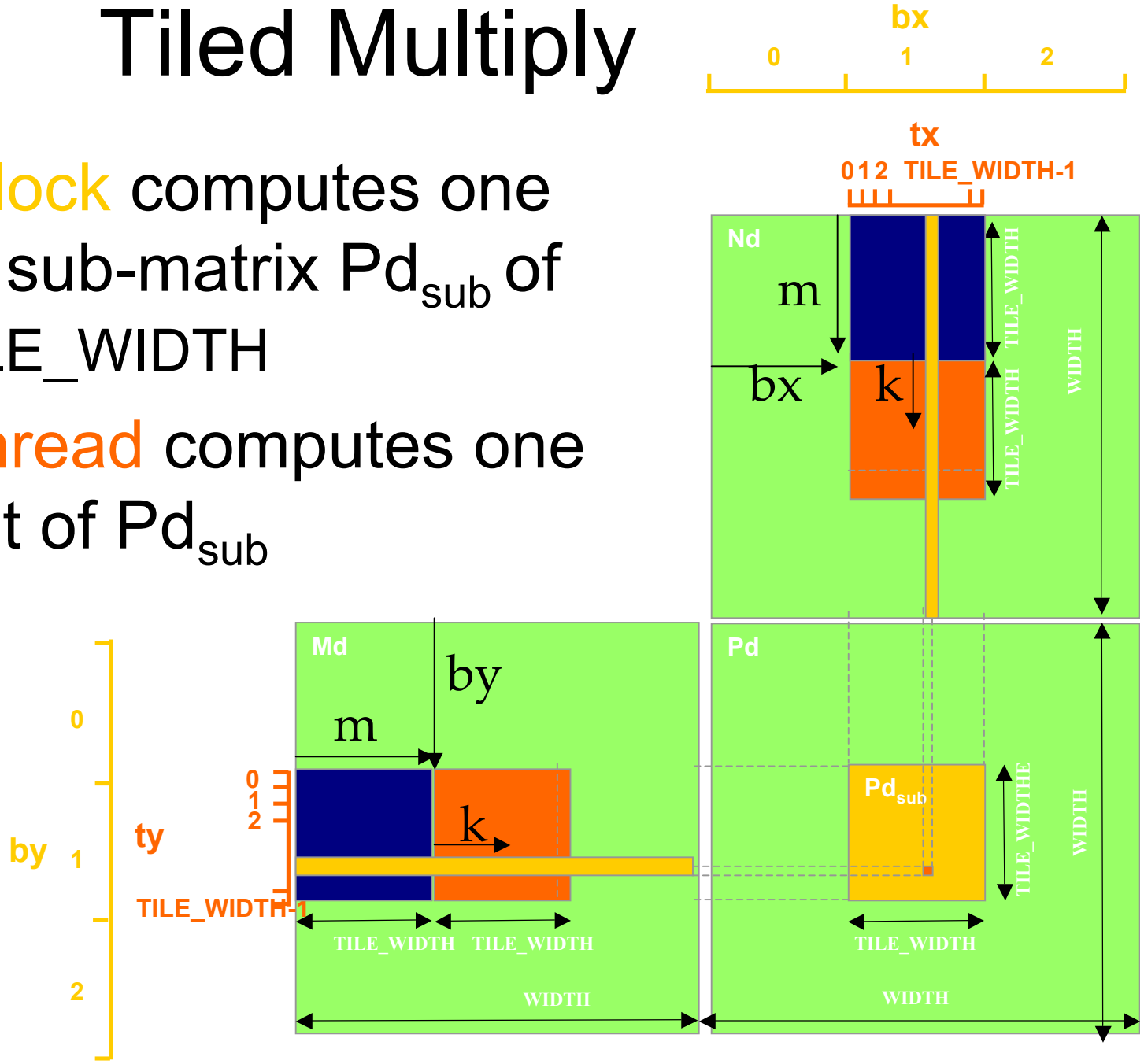


Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P



Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



Two Versions

- There are two versions of the solution:
  - A simplified solution by Kirk and Hwu
  - One from the CUDA manual

Kirk and Hwu version

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.    float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of Md and Nd tiles into shared memory
9.     Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
11.    __syncthreads();

11.    for (int k = 0; k < TILE_WIDTH; ++k)
12.   Pvalue += Mds[ty][k] * Nds[k][tx];
13.   Synchthreads();
14.  }
13.    Pd[Row*Width+Col] = Pvalue;
}
```

CUDA Manual Version

- A matrix will be represented as a struct containing a pointer to the first element in a flat array, and other parameters.

- This is so that sub-matrices can be represented **uniformly** as matrices are.

Matrix Struct Declaration

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)

typedef struct {

int width;
int height;

int stride;

float* elements;

} Matrix;
```

Matrix Multiply Host Code 1/3

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE

void MatMul(const Matrix A, const Matrix B, Matrix C)
{
// Load A and B to device memory

Matrix d_A;

d_A.width = d_A.stride = A.width; d_A.height = A.height;

size_t size = A.width * A.height * sizeof(float);

cudaMalloc(&d_A.elements, size);

cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
```

Matrix Multiply Host Code 2/3

```
Matrix d_B;

d_B.width = d_B.stride = B.width; d_B.height = B.height;

size = B.width * B.height * sizeof(float);

cudaMalloc(&d_B.elements, size);

cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);


// Allocate C in device memory

Matrix d_C;

d_C.width = d_C.stride = C.width; d_C.height = C.height;

size = C.width * C.height * sizeof(float);
```

Matrix Multiply Host Code 3/3

```
// Invoke kernel

dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);

MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

// Read C from device memory

cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

// Free device memory

cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}
```

Matrix Multiply *Device* Code 1/5

```
// Matrix multiplication kernel called by MatMul()

__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
// Block row and column

int blockRow = blockIdx.y;
int blockCol = blockIdx.x;

// Each block computes one sub-matrix Csub of C

Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

// Each thread computes one element of the sub-matrix Csub.

float Cvalue = 0;

// Thread row and column within Csub

int row = threadIdx.y;
int col = threadIdx.x;
```

Matrix Multiply Device Code 2/5

```
// Loop over all the sub-matrices of A and B that are required to compute Csub

for (int m = 0; m < (A.width / BLOCK_SIZE); ++m)
{
// Get sub-matrices Asub, Bsub of A, B

Matrix Asub = GetSubMatrix(A, blockRow, m);
Matrix Bsub = GetSubMatrix(B, m, blockCol);

// Shared memory used to store Asub and Bsub respectively

__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

// Load Asub and Bsub from device memory to shared memory
// Each thread loads one element of each sub-matrix

As[row][col] = GetElement(Asub, row, col);

Bs[row][col] = GetElement(Bsub, row, col);

// Synchronize to make sure the sub-matrices are loaded before starting the computation

__syncthreads();
```

Matrix Multiply Device Code 3/5

```
// Inner loop: Multiply Asub and Bsub together

for (int e = 0; e < BLOCK_SIZE; ++e)

        Cvalue += As[row][e] * Bs[e][col];

// Synchronize to make sure that the preceding

// computation is done before loading two new

// sub-matrices of A and B in the next iteration

__syncthreads();


// Write Csub to device memory

// Each thread writes one element

SetElement(Csub, row, col, Cvalue);
}
```

Matrix Element Getter and Setter 4/5

```
// Get a matrix element

__device__ float GetElement(const Matrix A, int row, int col)
{
return A.elements[row * A.stride + col];
}


// Set a matrix element

__device__ void SetElement(Matrix A, int row, int col, float value)
{
A.elements[row * A.stride + col] = value;
}
```

Block Getter 5/5

```
// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A

__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
Matrix Asub;

Asub.width    = BLOCK_SIZE;
Asub.height   = BLOCK_SIZE;

Asub.stride   = A.stride;

Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
return Asub;
}
```

Summary- Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__... __global__, __constant__, __texture__
- Function prototypes
  - __global__ void kernelOne(…)
  - float handyFunction(…)
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution
- Kernel – void kernelOne(type args,…)
  - variables declaration - __local__, __shared__
    - automatic variables transparently assigned to registers or local memory
  - syncthreads()…
- Other functions
  - float handyFunction(int inVar…);

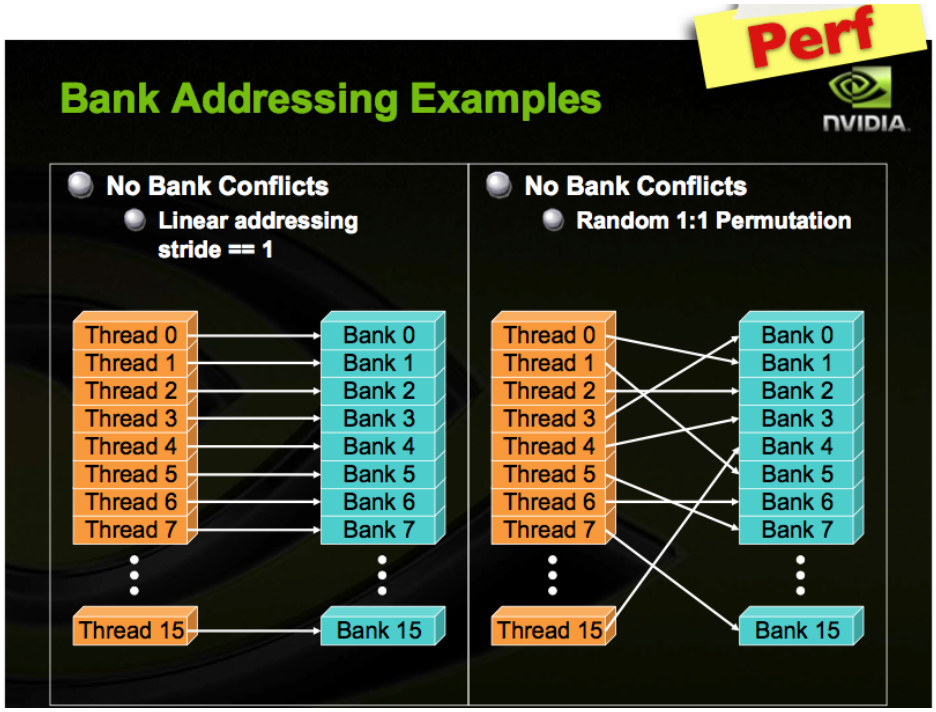
Shared memory bank conflicts

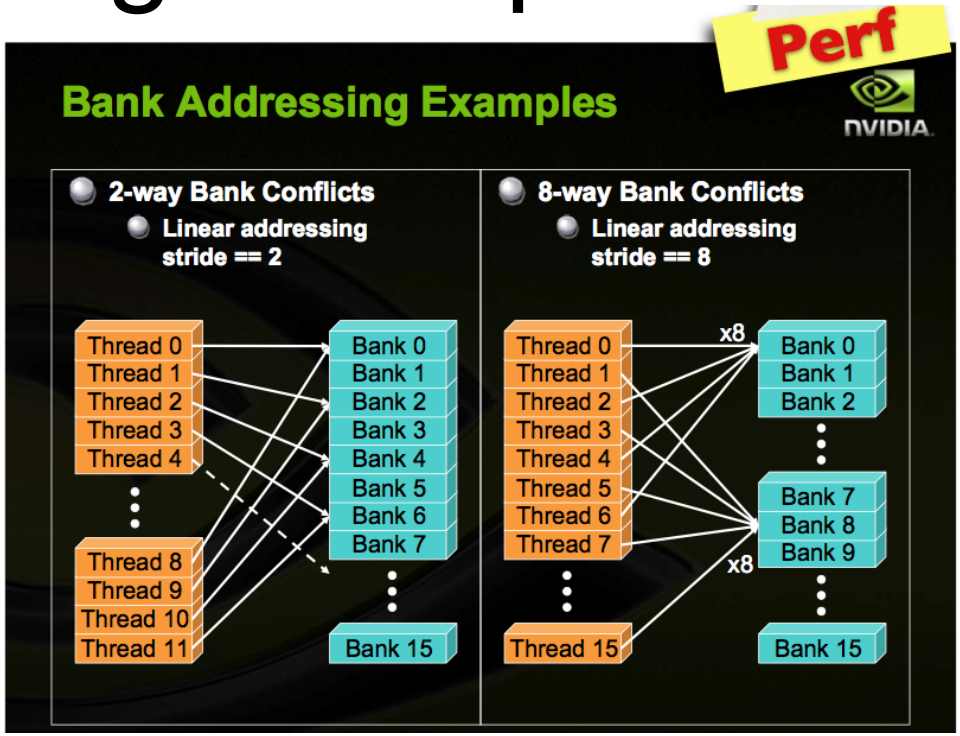
Perf

NVIDIA

- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

Bank Addressing Examples





Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`

Host-Device Asynchrony

Some function calls are asynchronous: Control is returned to the host thread before the device has completed the requested task. These are:

- Kernel launches;
- Device - device memory copies;
- Host - device memory copies of a memory block of 64 KB or less;
- Memory copies performed by functions that are suffixed with Async;
- Memory set function calls.

Programmers can globally disable asynchronous kernel launches for all CUDA applications running on a system by setting the CUDA_LAUNCH_BLOCKING environment variable to 1. This feature is provided for **debugging** purposes only and should never be used as a way to make production software run reliably.

GPU Atomic Integer Operations

Atomic operations on integers in global memory:

- Associative operations on signed/unsigned ints
- add, sub, min, max, ...
- and, or, xor
- Increment, decrement
- Exchange, compare and swap

Concurrent Kernels

Some devices of compute capability 2.x can execute multiple kernels concurrently.

Applications may query this capability by calling **cudaGetDeviceProperties**()

and checking the concurrentKernels property.

The **maximum number of kernel launches** that a device can execute concurrently is sixteen.

A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context.

Compiling under Linux

Command line. CUDA provides nvcc (a NVIDIA “compiler-driver”. Use instead c

**nvcc –O3 –o <exe> <input> -I/usr/local/cuda/include **
–L/usr/local/cuda/lib –lcudart

Separates compiled code for CPU and for GPU and compiles code.
Need regular C compiler installed for CPU.

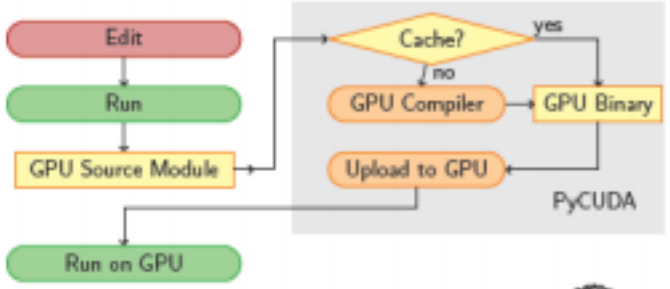
CUDA Data Parallel Primitives Library: CUDPP

- http://code.google.com/p/cudpp/

Python + CUDA = PyCUDA

- All of CUDA in a modern scripting language
- Full Documentation
- Free, open source (MIT)
- Also: PyOpenCL

- CUDA C Code = Strings
- Generate Code Easily
  - Automated Tuning
- Batteries included: GPU Arrays, RNG, ...
- Integration: `numpy` arrays, Plotting, Optimization, ...



BROWN

jCuda

Using jCUDA you can create cross-platform CUDA solutions, that can run on any operating system supported by CUDA without changing your code.
Either select between Windows XP or Vista by Microsoft or even Linux/MacOS/Solaris systems.
Current support is for both 32 and 64 bits of every platform.

Features

- Double precision
- Object model for CUDA programming
- CUDA 2.1 Driver API
- CUDA 2.1 Runtime API
- CUFFT routines
- OpenGL interoperability
- * Support for CUBLAS routines will be added in the future

Operating System Support

- Microsoft Windows
- Linux
- * Support for Mac OSX will be added in the future
- * Support for Solaris 10 (x86) will be added in the future

http://www.hoopoe-cloud.com/Solutions/jCUDA/Default.aspx

OpenCL

2.3.6, 14/9/2009
Updates to native wrappers, added *SizeT* structure to handle 32/64 systems compatibility for functions taking *size_t* as parameter.
Support for CUDA 2.3 through .NET bindings to CUDA functions.
Currently supported on Windows, Linux and MacOS platforms.

Features

- Double precision
- Object model for CUDA programming
- CUDA 2.2 Driver API
- CUDA 2.2 Runtime API
- CUFFT routines
- CUBLAS routines
- Direct3D interoperability
- OpenGL interoperability

Operating System Support

- Microsoft Windows
- Linux 32/64 bit (using Mono)
- Mac OSX (using Mono)

http://www.hoopoe-cloud.com/Solutions/CUDA.NET/Default.aspx

Cuda.NET

2.3.6, 14/9/2009
Updates to native wrappers, added *SizeT* structure to handle 32/64 systems compatibility for functions taking *size_t* as parameter.
Support for CUDA 2.3 through .NET bindings to CUDA functions.
Currently supported on Windows, Linux and MacOS platforms.

Features

- Double precision
- Object model for CUDA programming
- CUDA 2.2 Driver API
- CUDA 2.2 Runtime API
- CUFFT routines
- CUBLAS routines
- Direct3D interoperability
- OpenGL interoperability

Operating System Support

- Microsoft Windows
- Linux 32/64 bit (using Mono)
- Mac OSX (using Mono)

http://www.hoopoe-cloud.com/Solutions/CUDA.NET/Default.aspx

Matrix Transpose Example

Perf

Matrix Transpose

- SDK Sample ("transpose")
- Illustrates:
  - Coalescing
  - Avoiding SMEM bank conflicts
  - Speedups for even small matrices

Uncoalesced Transpose

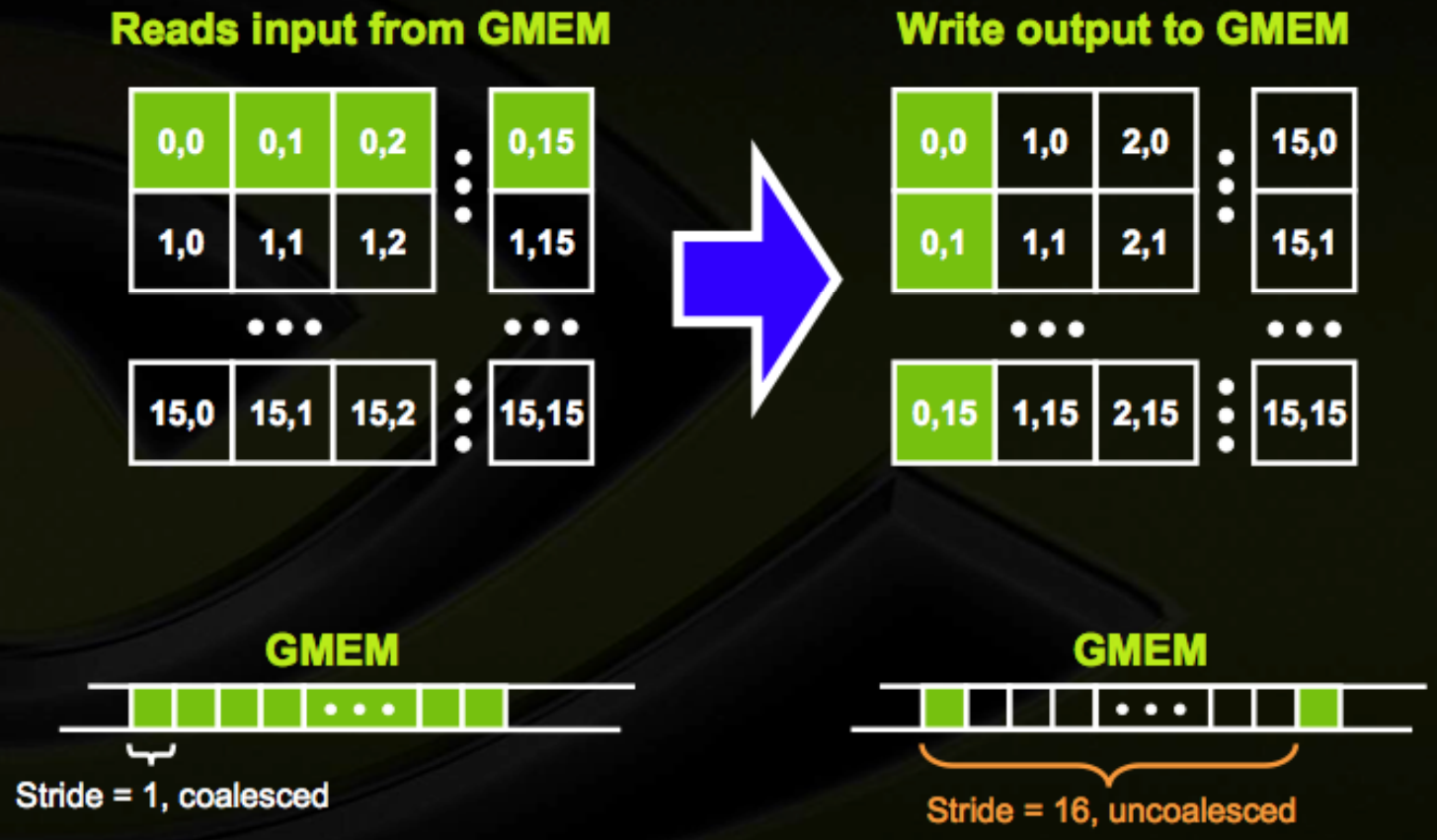
Perf

NVIDIA

```
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
1.  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
2.  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

3.  if (xIndex < width && yIndex < height)
    {
4.      unsigned int index_in  = xIndex + width * yIndex;
5.      unsigned int index_out = yIndex + height * xIndex;
6.      odata[index_out] = idata[index_in];
    }
}
```

Uncoalesced Transpose

Perf



Coalesced Transpose

Perf

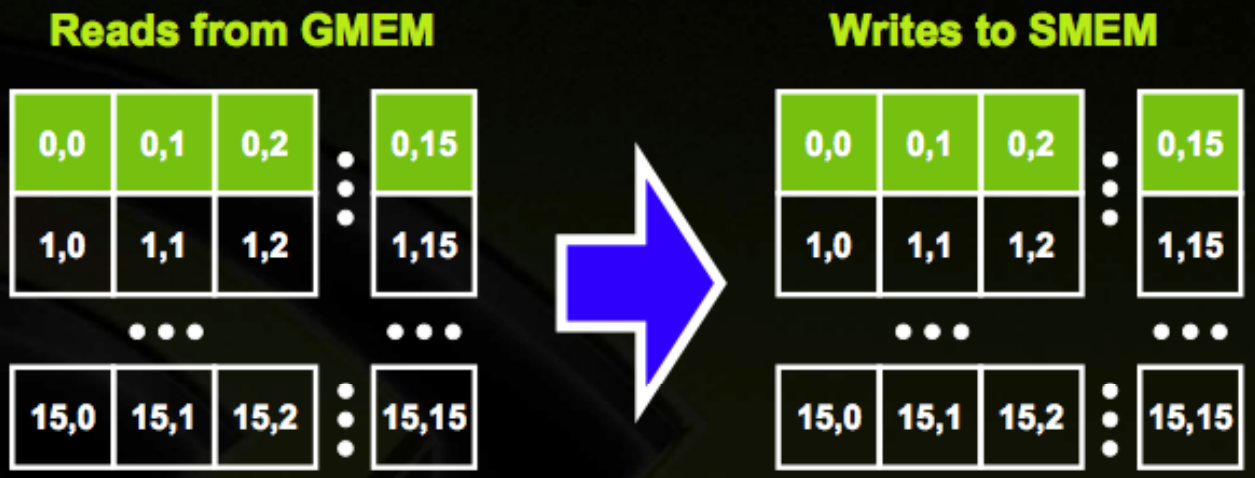
NVIDIA

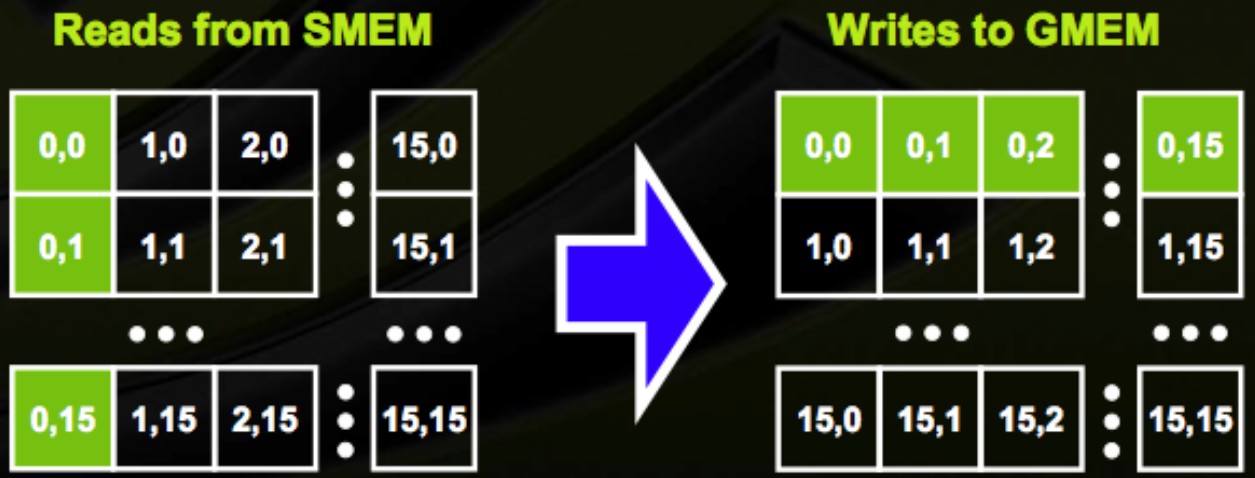
- Assumption: matrix is partitioned into square tiles
- Threadblock (bx, by):
  - Read the (bx,by) input tile, store into SMEM
  - Write the SMEM data to (by,bx) output tile
    - Transpose the indexing into SMEM
- Thread (tx,ty):
  - Reads element (tx,ty) from input tile
  - Writes element (tx,ty) into output tile
- Coalescing is achieved if:
  - Block/tile dimensions are multiples of 16

Coalesced Transpose

Perf

NVIDIA



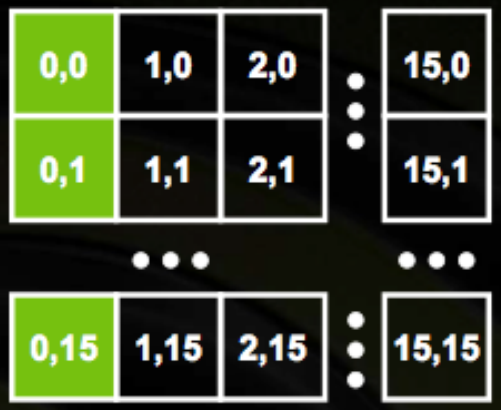


SMEM Optimization

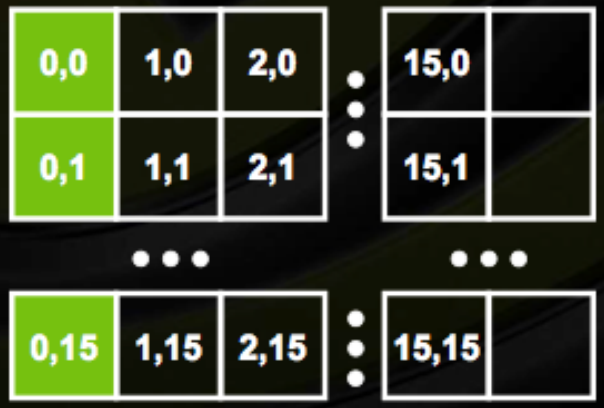
Pert

NVIDIA

Reads from SMEM



- Threads read SMEM with stride = 16
  - Bank conflicts



- Solution
  - Allocate an “extra” column
  - Read stride = 17
  - Threads read from consecutive banks

Coalesced Transpose

Perf

NVIDIA

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
1.  __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];

2.  unsigned int xBlock = blockDim.x * blockIdx.x;
3.  unsigned int yBlock = blockDim.y * blockIdx.y;
4.  unsigned int xIndex = xBlock + threadIdx.x;
5.  unsigned int yIndex = yBlock + threadIdx.y;
6.  unsigned int index_out, index_transpose;

7.  if (xIndex < width && yIndex < height)
    {
8.      unsigned int index_in = width * yIndex + xIndex;
9.      unsigned int index_block = threadIdx.y * (BLOCK_DIM+1) + threadIdx.x;
10.     block[index_block] = idata[index_in];
11.     index_transpose = threadIdx.x * (BLOCK_DIM+1) + threadIdx.y;
12.     index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
13. __syncthreads();

14. if (xIndex < width && yIndex < height)
15.     odata[index_out] = block[index_transpose];
}
```

Transpose Timings

Perf

NVIDIA

- Speedups with coalescing and SMEM optimization:
  - 128x128: 0.011ms vs. 0.022ms (2.0X speedup)
  - 512x512: 0.07ms vs. 0.33ms (4.5X speedup)
  - 1024x1024: 0.30ms vs. 1.92ms (6.4X speedup)
  - 1024x2048: 0.79ms vs. 6.6ms (8.4X speedup)
- Coalescing without SMEM optimization:
  - 128x128: 0.014ms
  - 512x512: 0.101ms
  - 1024x1024: 0.412ms
  - 1024x2048: 0.869ms

October 28, 2010

New Chinese GPGPU Super Outruns Jaguar

The end of the US dominance at the top of the TOP500 appears to be at hand. Tianhe-1A, a new Chinese supercomputer powered by over 7,000 NVIDIA Tesla GPUs has recorded a Linpack score of 2.507 petaflops. That would beat out Oak Ridge National Lab's 1.759 petaflop Jaguar machine, the current TOP500 title holder, by a wide margin.

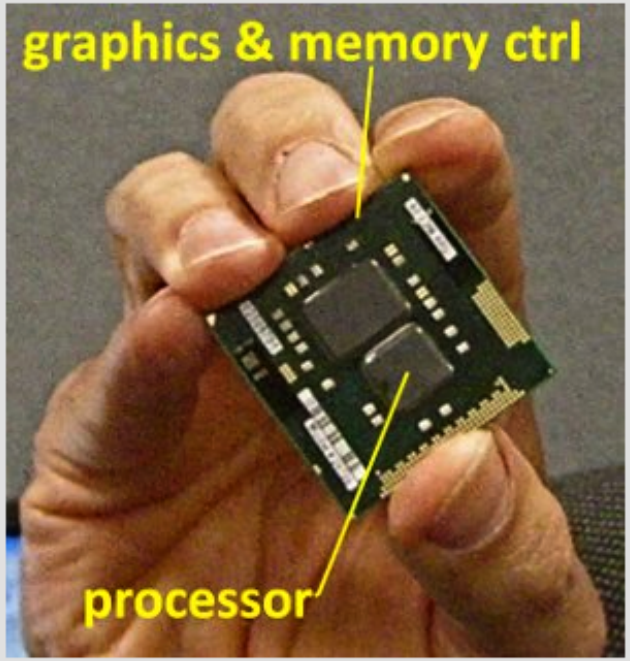
It would also be the first time a non-US supercomputer held the number one spot in six years. From June 2002 to June 2004, Japan's Earth Simulator was the fastest supercomputer in the world. In September 2004, it yielded its title to the new kid on the block -- IBM's Blue Gene/L. The US has never looked back.

Until now.

The concentrated performance available in high end discrete GPUs has opened up the petaflop club for a lot more players. But to get to the top of the heap, you need thousands of GPUs. Tianhe-1A, for example, sports 7,168 of them, in this case, NVIDIA Tesla M2050 (Fermi) GPUs. These represent the lion's share of FLOPS in the system, despite the presence of accompanying 14,336 CPUs. Tianhe-1A also comes with 262 TB of memory and 2 PB of Lustre-based storage.

Feature - GPU-based cheap supercomputing coming to an end

Nvidia's CUDA has been hailed as "Supercomputing for the Masses," and with good reason – amazing speedups ranging from 10x through hundreds have been reported on scientific / technical code. CUDA has become a darling of academic computing and a major player in DARPA's Exascale program, but performance alone does not account for that popularity: price clinches the deal. For all that computing power, they're incredibly cheap. As Sharon Glotzer of UMich noted, "Today you can get two gigaflops for $500. That is ridiculous." It is indeed. And it's only possible because CUDA is subsidized by sinking the fixed costs of its development into the high volumes of Nvidia's mass market low-end GPUs.



Intel's Sandy Bridge architecture places the processor and GPU on the same chip.

Image courtesy Greg Pfister.

Unfortunately, that subsidy won't last forever; its end is now visible. Intel has now started pounding the marketing drums on something long predicted: integration of Intel's graphics onto the same die as its next generation "Sandy Bridge" processor chip, due out in mid-2011.

Probably not coincidentally, mid-2011 is when AMD's Llano processor will see daylight. It incorporates enough graphics-related processing to be an apparently decent DX11 GPU, although to my knowledge the architecture hasn't been disclosed in detail.

Just prior to this Fall's IDF (Intel Developer Forum), Anandtech received an early demo part of Sandy Bridge and checked out the graphics, among other things. Their net is that