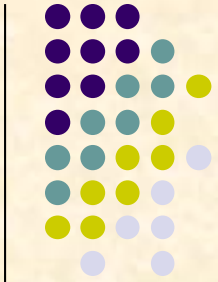


# 众核并行程序设计

北京大学信息学院

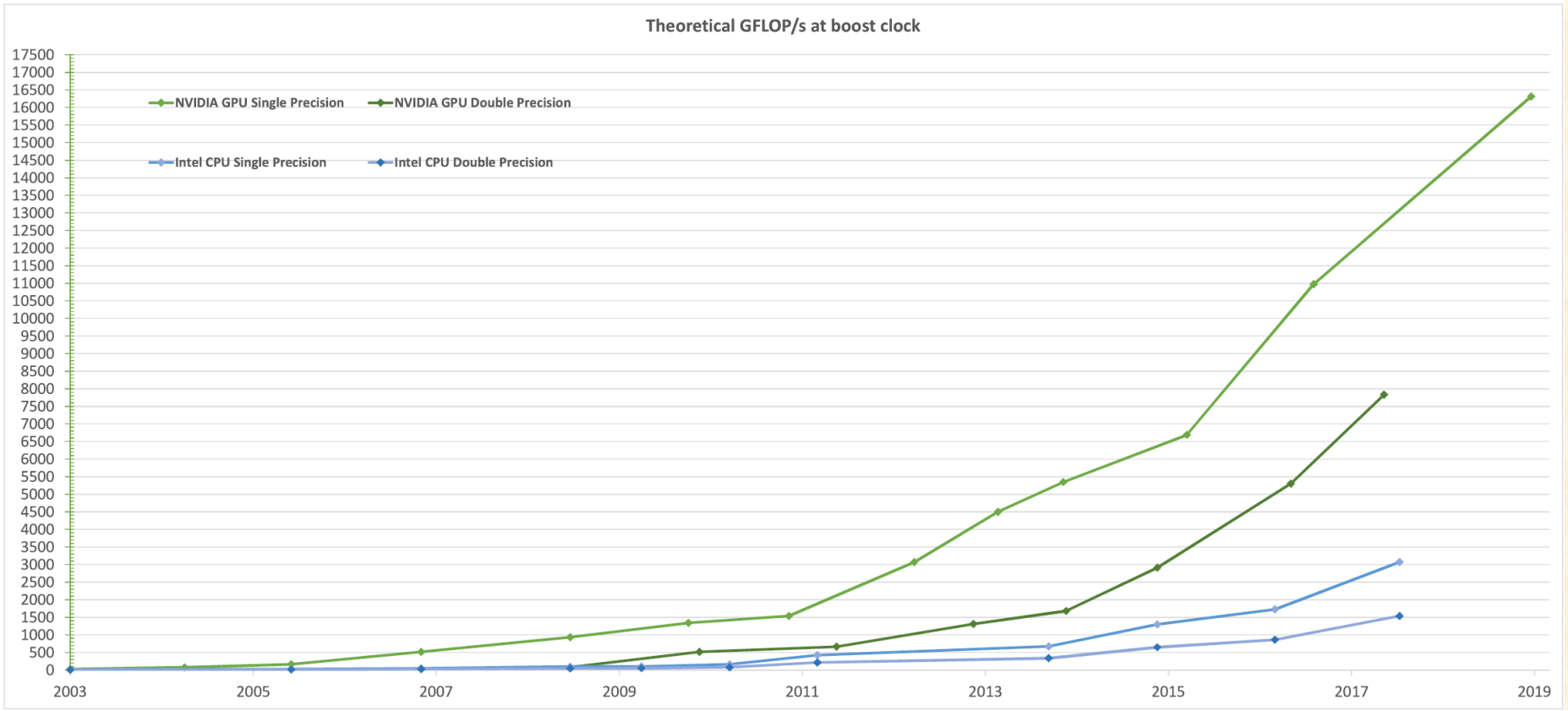



Figure 1 Floating-Point Operations per Second for the CPU and GPU

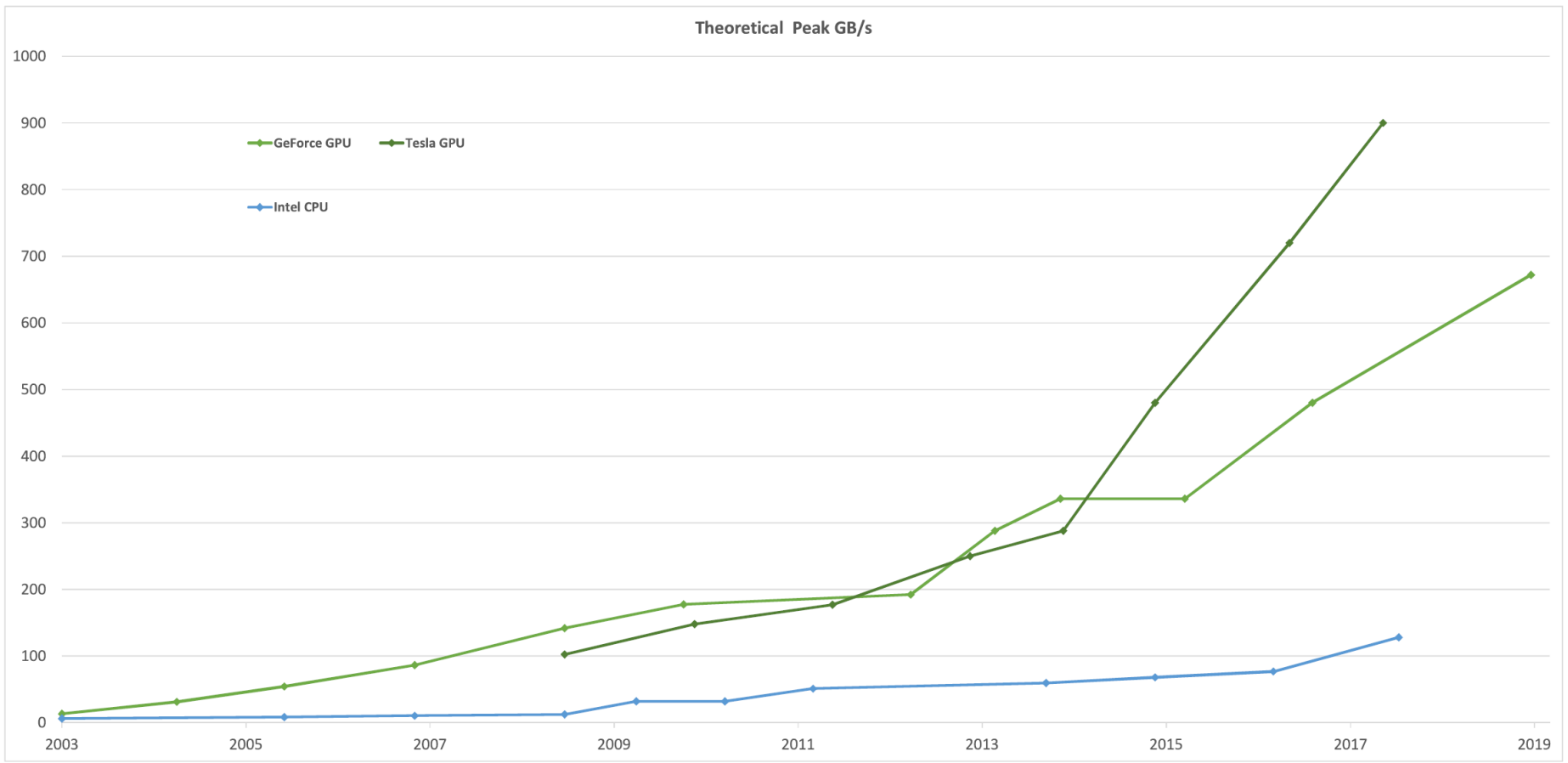
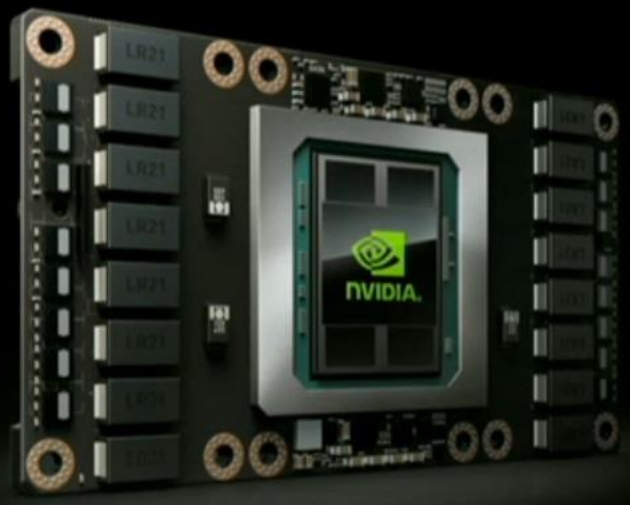


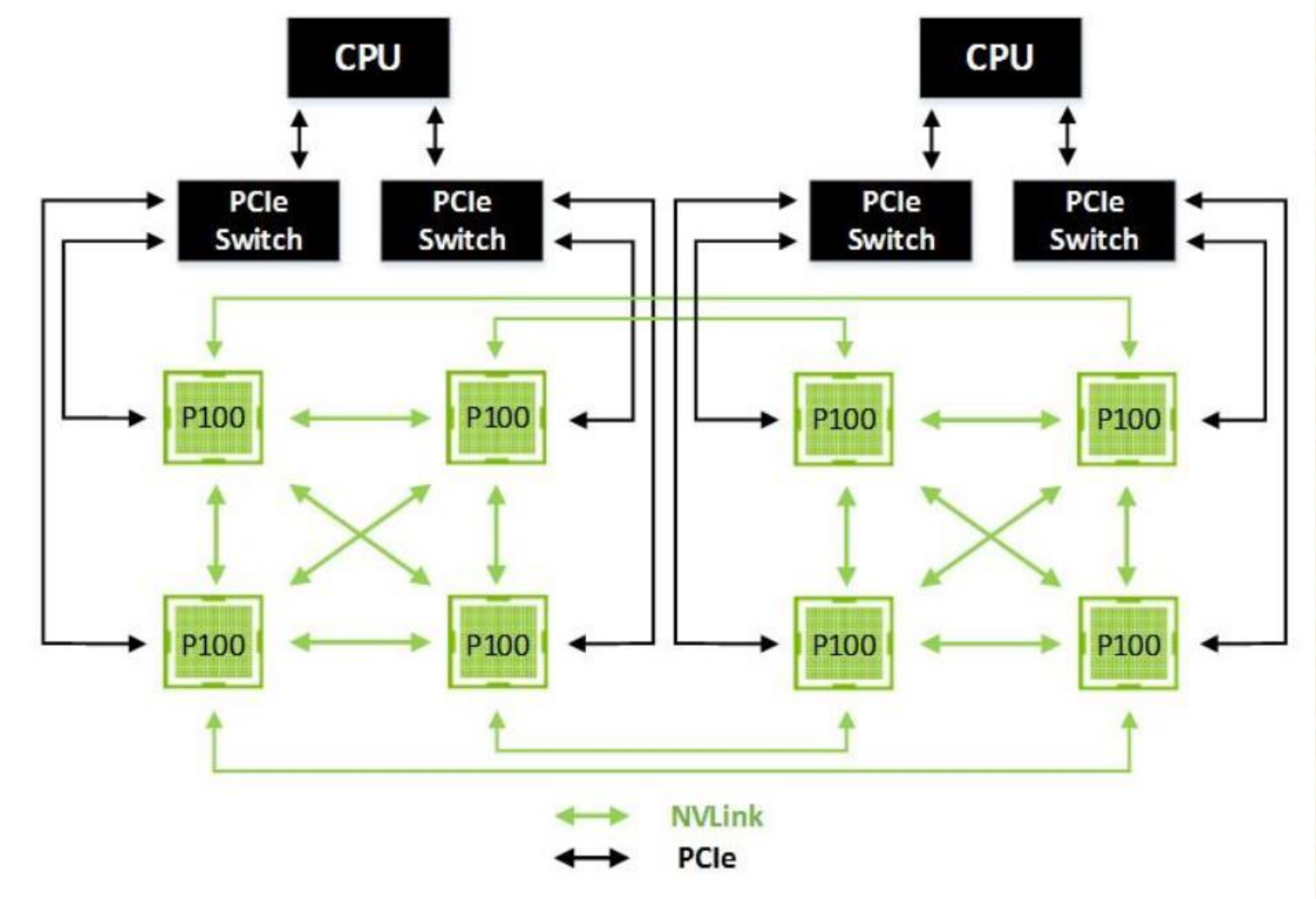

Figure 2 Memory Bandwidth for the CPU and GPU

# TESLA P100

## THE MOST ADVANCED HYPERSCALE DATACENTER GPU EVER BUILT

150B XTORS | 5.3TF FP64 | 10.6TF FP32 | 21.2TF FP16 | 14MB SM RF | 4MB L2 Cache




CPU
CPU
PCIe Switch
PCIe Switch
PCIe Switch
PCIe Switch
P100
P100
P100
P100
P100
P100
P100
P100
NVLink
PCIe


NVIDIA
T TAIWAN 1710A1
P6CA08 000
TR21
1704

# GPU PERFORMANCE COMPARISON

| | P100 | V100 | Ratio |
|---|---|---|---|
| DL Training | 10 TFLOPS | 120 TFLOPS | 12x |
| DL Inferencing | 21 TFLOPS | 120 TFLOPS | 6x |
| FP64/FP32 | 5/10 TFLOPS | 7.5/15 TFLOPS | 1.5x |
| HBM2 Bandwidth | 720 GB/s | 900 GB/s | 1.2x |
| STREAM Triad Perf | 557 GB/s | 855 GB/s | 1.5x |
| NVLink Bandwidth | 160 GB/s | 300 GB/s | 1.9x |
| L2 Cache | 4 MB | 6 MB | 1.5x |
| L1 Caches | 1.3 MB | 10 MB | 7.7x |

| | V100 PCIe | V100 SXM2 | V100S PCIe |
|---|---|---|---|
| GPU Architecture | NVIDIA Volta | | |
| NVIDIA Tensor Cores | 640 | | |
| NVIDIA CUDA® Cores | 5,120 | | |
| Double-Precision Performance | 7 TFLOPS | 7.8 TFLOPS | 8.2 TFLOPS |
| Single-Precision Performance | 14 TFLOPS | 15.7 TFLOPS | 16.4 TFLOPS |
| Tensor Performance | 112 TFLOPS | 125 TFLOPS | 130 TFLOPS |
| GPU Memory | 32 GB /16 GB HBM2 | | 32 GB HBM2 |
| Memory Bandwidth | 900 GB/sec | | 1134 GB/sec |
| ECC | Yes | | |
| Interconnect Bandwidth | 32 GB/sec | 300 GB/sec | 32 GB/sec |
| System Interface | PCIe Gen3 | NVIDIA NVLink™ | PCIe Gen3 |
| Form Factor | PCIe Full Height/Length | SXM2 | PCIe Full Height/Length |
| Max Power Comsumption | 250 W | 300 W | 250 W |
| Thermal Solution | Passive | | |
| Compute APIs | CUDA, DirectCompute, OpenCL™, OpenACC® | | |

| Processor | SMs | CUDA Cores | Tensor Cores | Frequency | Cache | Max. Memory | Memory B/W |
|---|---|---|---|---|---|---|---|
| Nvidia P100 | 56 | 3,584 | N/A | 1,126 MHz | 4 MB L2 | 16 GB | 720 GB/s |
| Nvidia V100 | 80 | 5,120 | 640 | 1.53 GHz | 6 MB L2 | 16 GB | 900 GB/s |

| *Tesla Product* | *Tesla V100* | *Tesla P100* |
|---|---|---|
| Architecture | Volta | Pascal |
| Code name | GV100 | GP100 |
| Release Year | 2017 | 2016 |
| Cores / GPU | 5120 | 3584 |
| GPU Boost Clock | 1530 MHz | 1480 MHz |
| Tensor Cores / GPU | 640 | NA |
| Memory type | HBM2 | HBM2 |
| Maximum RAM amount | 32 GB | 16 GB |
| Memory clock speed | 1758 MHz | 1430 MHz |
| Memory bandwidth | 900.1 GB / s | 720.9 GB / s |
| CUDA Support | From 7.0 Version | From 6.0 Version |
| Floating-point performance | 14,029 gflops | 10,609 gflops |



## MULTICORE vs MANY-CORE

CPU

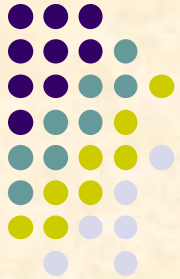
- Based on pipeline philosophy
- A lot of strucutres for cache and control
- More flexible
- MIMD – task parallelism
- Latency sensible

GPU

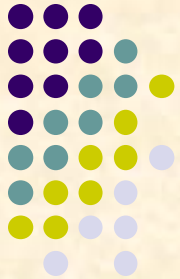
- Big amount of parallel data
- Less strucutres for cache and control
- Less flexibility
- SIMD – data parallelism
- Latency tolerant

# 其他并行语言/接口

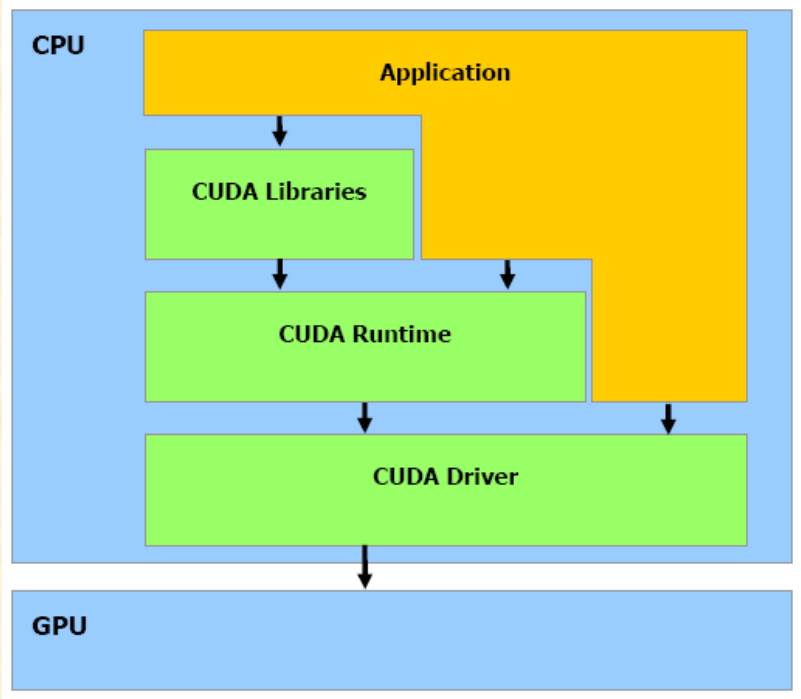
- OpenCL
- Brook++
- HMPP
- HPF
- Co-array Fortran
- PPL
- Op2
- Cilk
- TBB
- Chapel
- NESL
- X10
- ZPL
- MapReduce



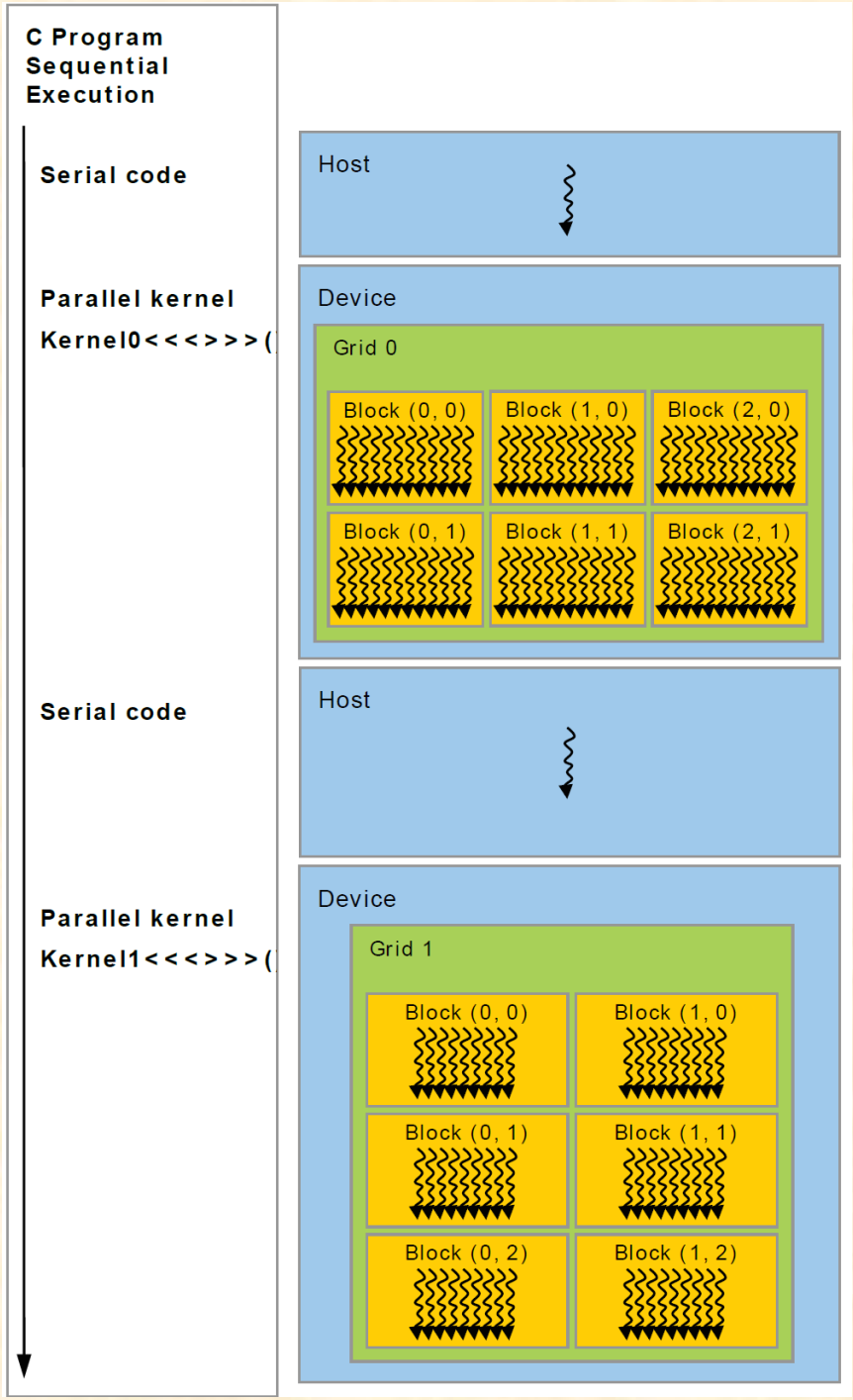
# Bottlenecks of parallel algorithms



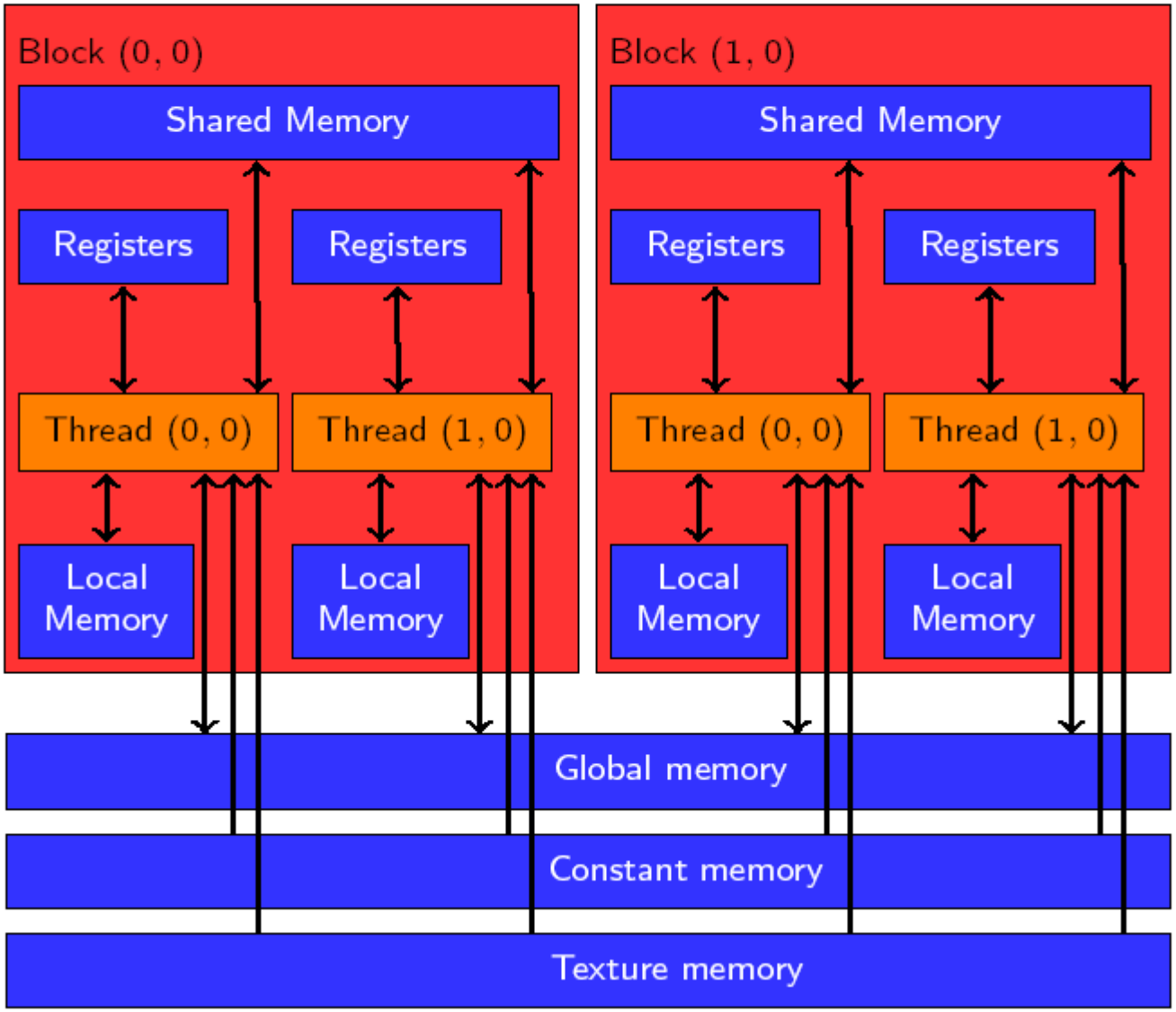
- SGEMM O(N^3/2)
  - Computation-bound
  - number of nodes, total flops of GPUs per node
- DGEMM O(N^3/2) and LINPACK
  - Computation-bound
  - GPU + CPU
- FFT O(NlogN)
  - Communication-bound: PCI and Infiniband, number of nodes related to amount of communication
  - GPU + CPU
- NBody  Brute force O(N^2)
  - Computation-bound
- Barnes-Hut tree O(NlogN)
  - Communication-bound: memory access pattern

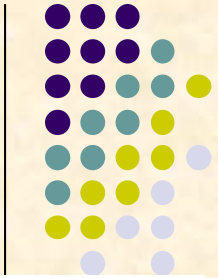

CPU
Application
CUDA Libraries
CUDA Runtime
CUDA Driver
GPU

NVIDIA


C Program
Sequential
Execution
Serial code
Parallel kernel
Kernel0<<<>>>(
Host
Device
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Serial code
Host
Parallel kernel
Kernel1<<<>>>(
Device
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)

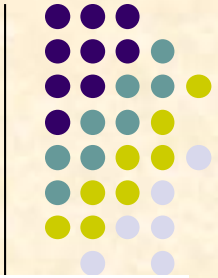
# CUDA exposes all the different types of memory on the GPU:





## To summarize

| | | | |
|---|---|---|---|
| Registers | Per thread | Read-Write | |
| Local memory | Per thread | Read-Write | |
| Shared memory | Per block | Read-Write | For sharing data within a block |
| Global memory | Per grid | Read-Write | Not cached |
| Constant memory | Per grid | Read-only | Cached |
| Texture memory | Per grid | Read-only | Spatially cached |

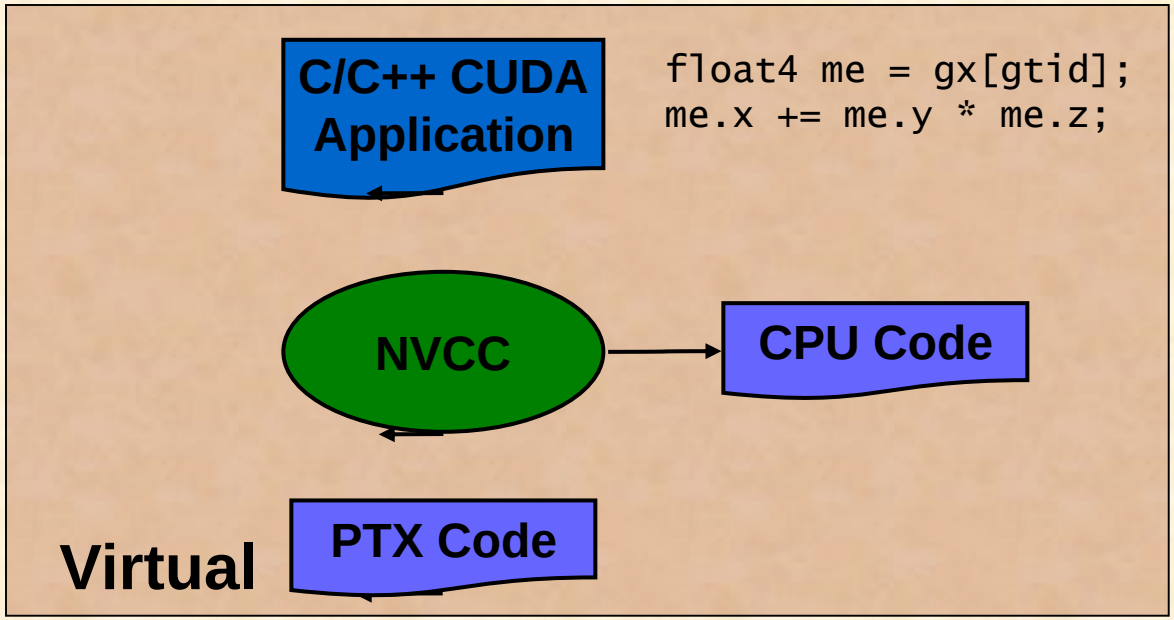


- 4 cycles to issue on memory fetch
- but 400-600 cycles of latency
  - The equivalent of 100 MADs
- Likely to be a performance bottleneck
- Order of magnitude speedups possible
  - Coalesce memory access
- Use shared memory to re-order non-coalesced addressing

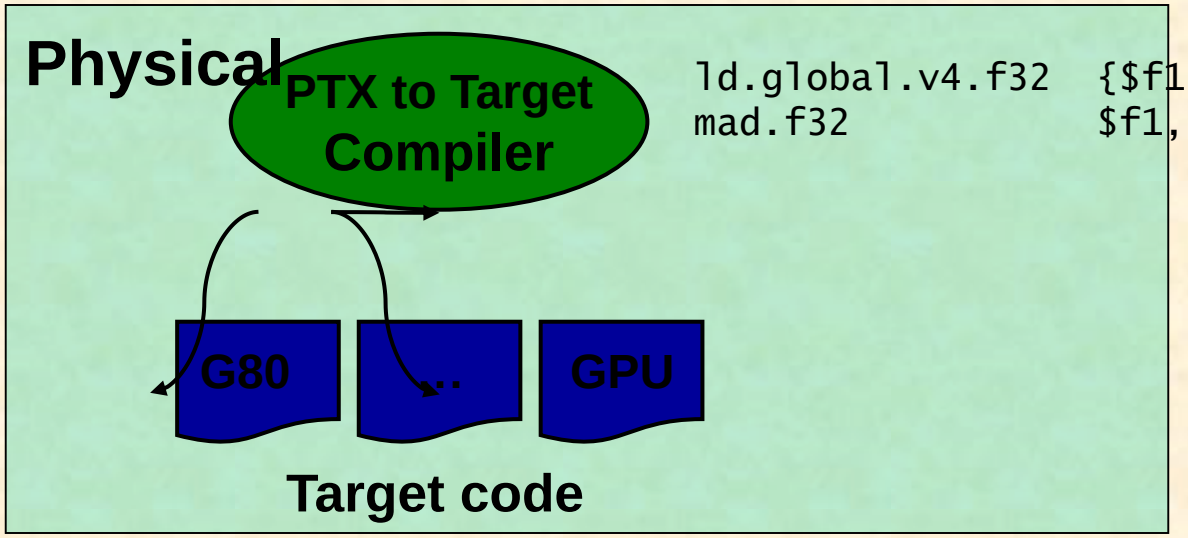
For best performance, global memory accesses should be coalesced:

- A memory access coordinated within a warp
- A contiguous, aligned, region of global memory
  - 128 bytes – each thread reads a `float` or `int`
  - 256 bytes – each thread reads a `float2` or `int2`
  - 512 bytes – each thread reads a `float4` or `int4`
  - float3s are not aligned!
- Warp base address (`WBA`) must be a multiple of `16*sizeof(type)`
- The $k$th thread should access the element at `WBA` $+ k$
- Not all threads need to participate
- These restrictions apply to both reading and writing
  - Use shared memory to achieve this

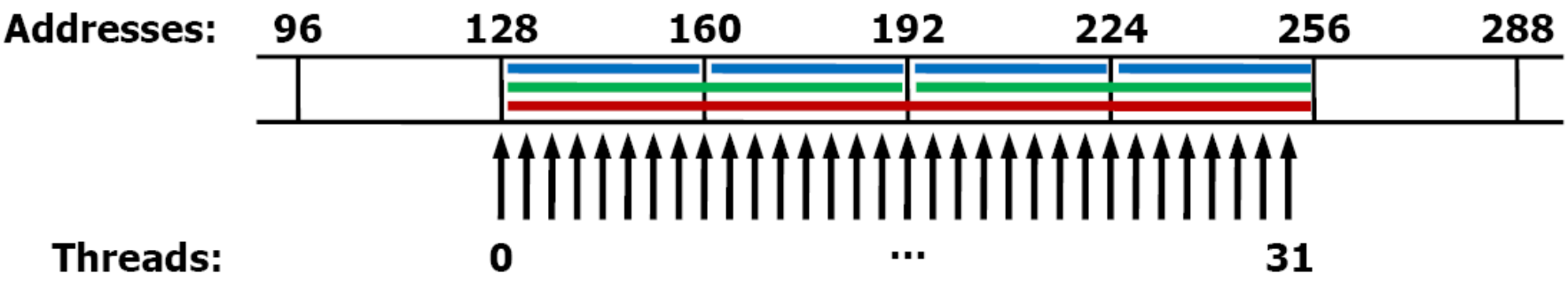
# Compiling a CUDA Program




- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

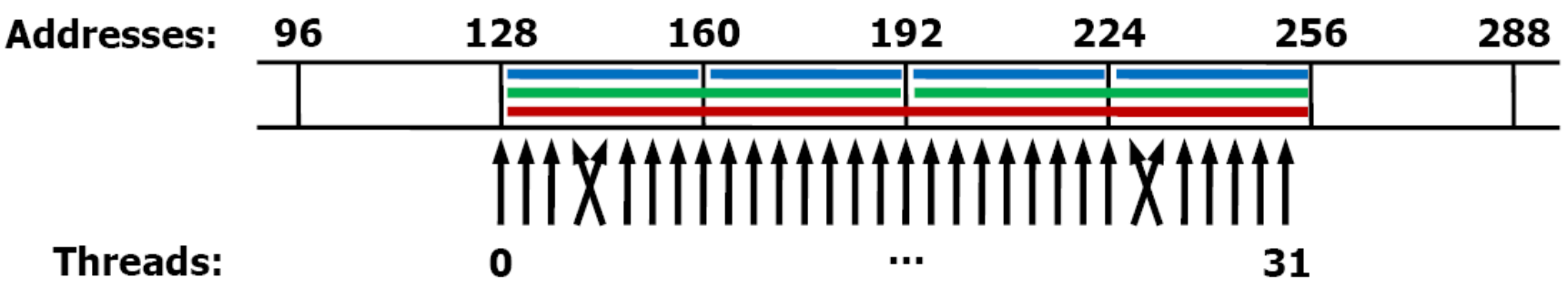


## Aligned and sequential



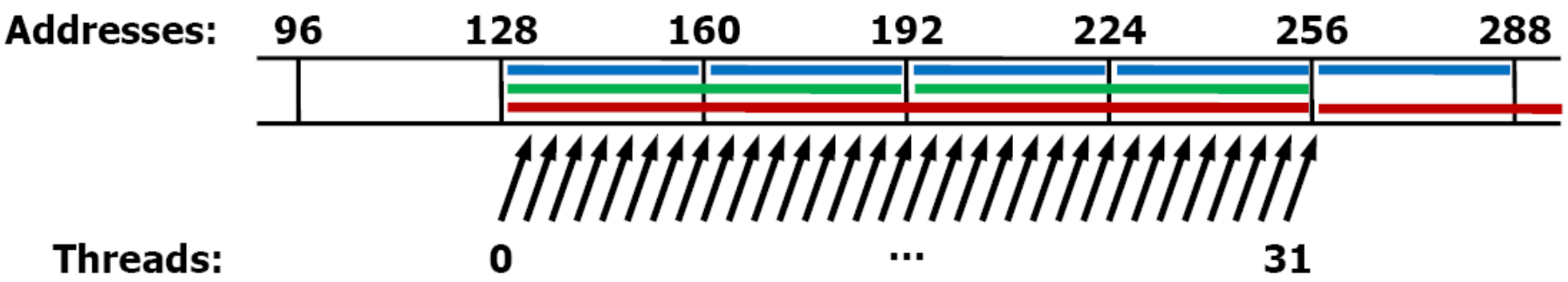

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.0 |
|---|---|---|---|
| Memory transactions: | Uncached | | Cached |
| | 1 x 64B at 128<br>1 x 64B at 192 | 1 x 64B at 128<br>1 x 64B at 192 | 1 x 128B at 128 |

## Aligned and non-sequential



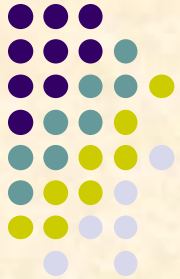

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.0 |
|---|---|---|---|
| Memory transactions: | Uncached | | Cached |
| | 8 x 32B at 128<br>8 x 32B at 160<br>8 x 32B at 192<br>8 x 32B at 224 | 1 x 64B at 128<br>1 x 64B at 192 | 1 x 128B at 128 |

## Misaligned and sequential

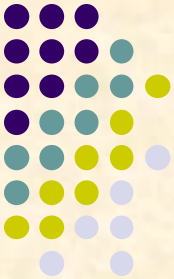



| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.0 |
|---|---|---|---|
| Memory transactions: | Uncached | | Cached |
| | 8 x 32B at 128<br>8 x 32B at 160<br>8 x 32B at 192<br>8 x 32B at 224 | 1 x 128B at 128<br>1 x 64B at 192<br>1 x 32B at 256 | 1 x 128B at 128<br>1 x 128B at 256 |

# Floating Point



- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host



# CUDA C

NVIDIA

## Standard C Code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{

  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

## Parallel C Code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096, 256>>>(n,2.0,x,y);
```

http://developer.nvidia.com/cuda-toolkit

# Example: Increment Array Elements


## CPU code

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, 16);
}
```

## CUDA code

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_gpu<<< 4, 4 >>>(a, b, 16);
}
```

## CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

## CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

The nested `for`-loops are replaced with an implicit grid

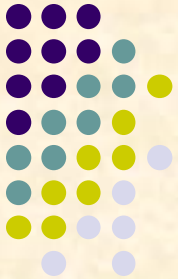
```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

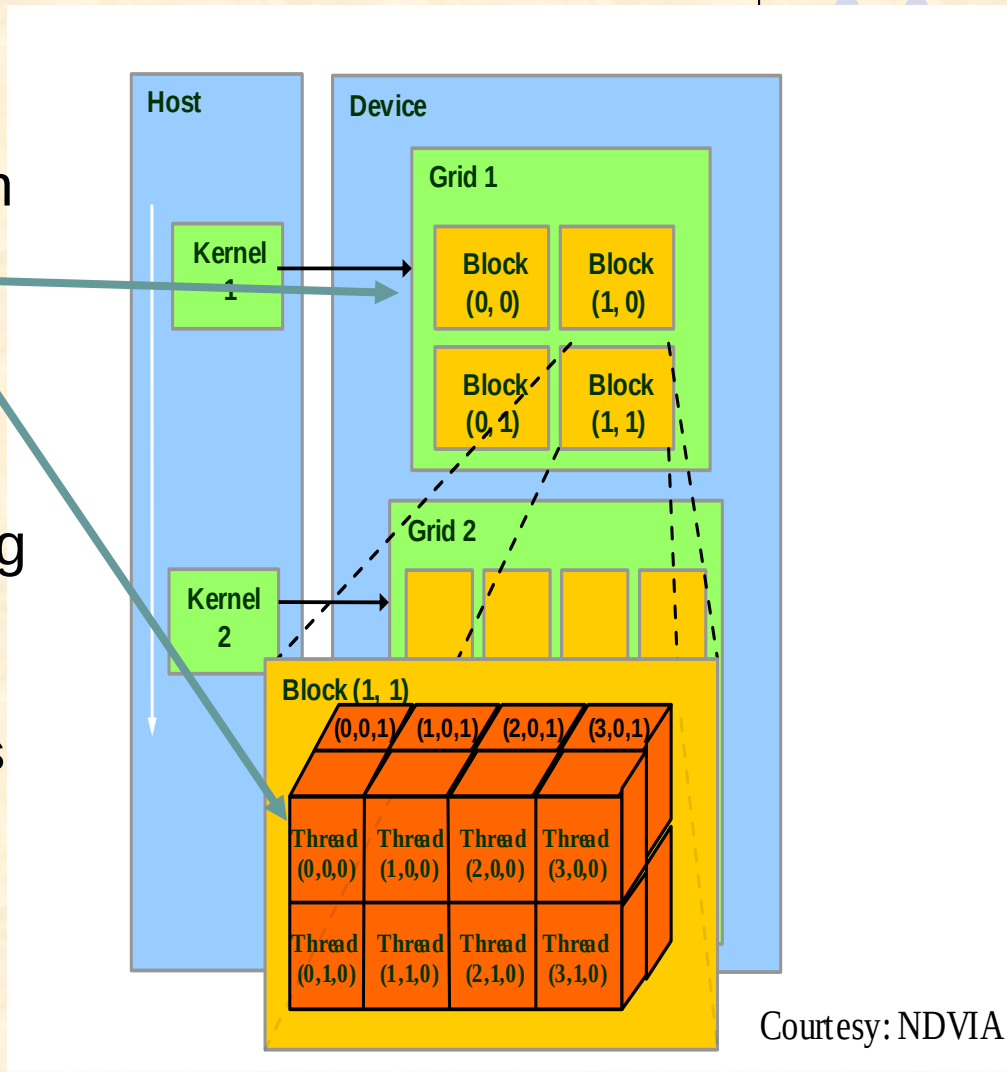
# SIMD + SPMD = STMD

- SIMD for threads of one warp
- SPMD for warps
- STMD



# Block IDs and Thread IDs

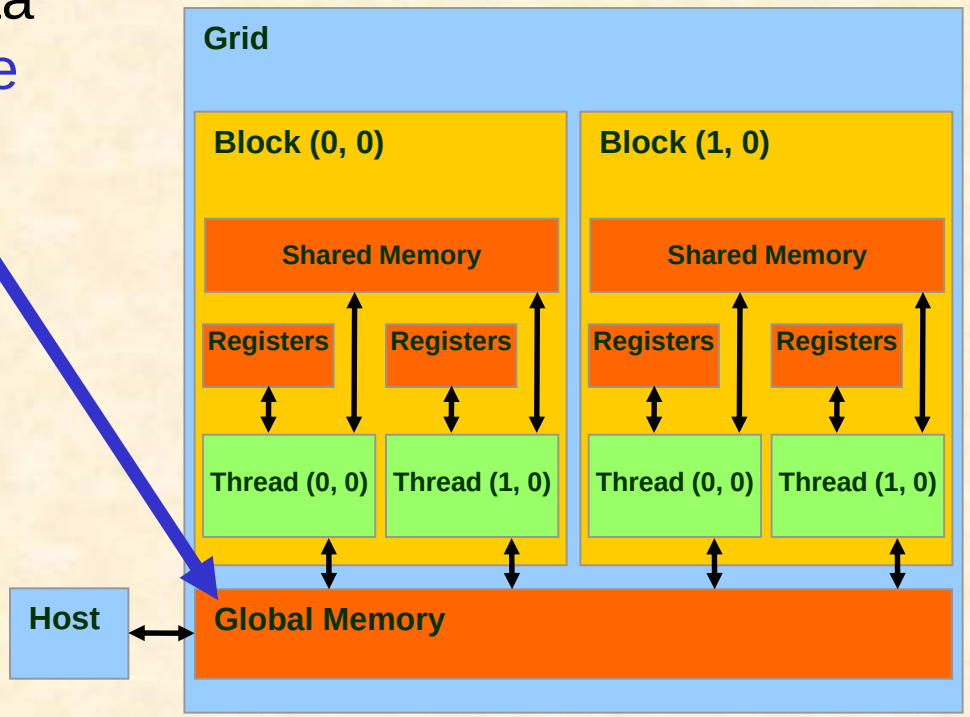
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



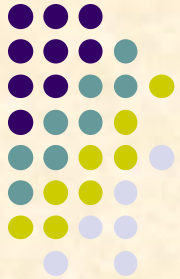

Courtesy: NDVIA

# CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory will come later



# CUDA API Highlights: Easy and Lightweight



- The API is an extension to the ANSI C programming language

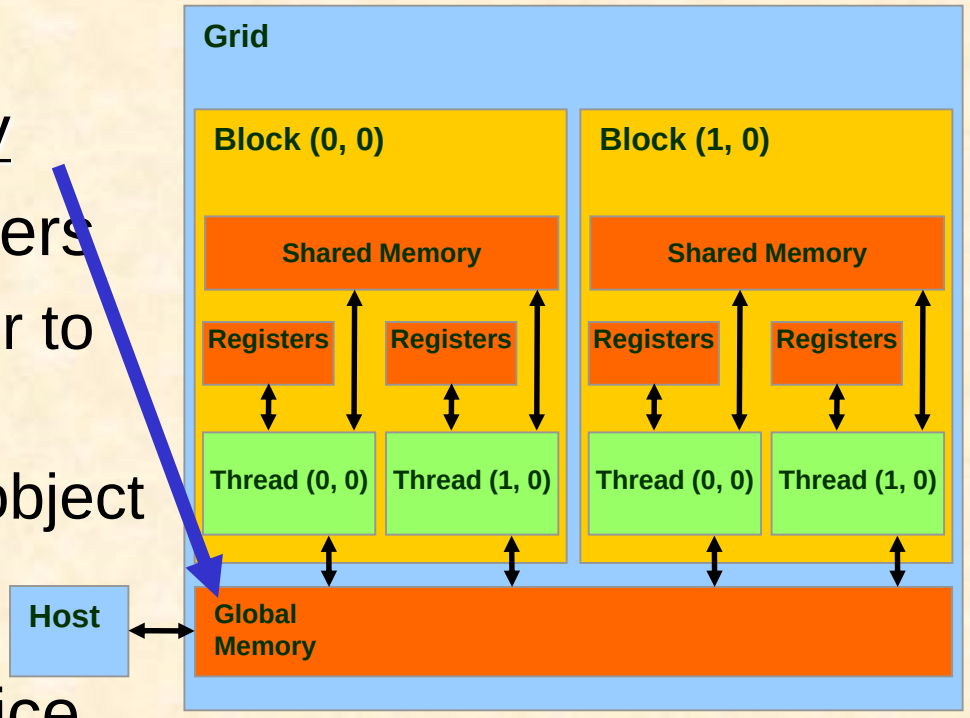
  → Low learning curve

- The hardware is designed to enable lightweight runtime and driver

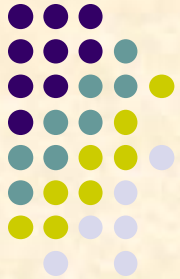
  → High performance

# CUDA Device Memory Allocation

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



# CUDA Device Memory Allocation (cont.)

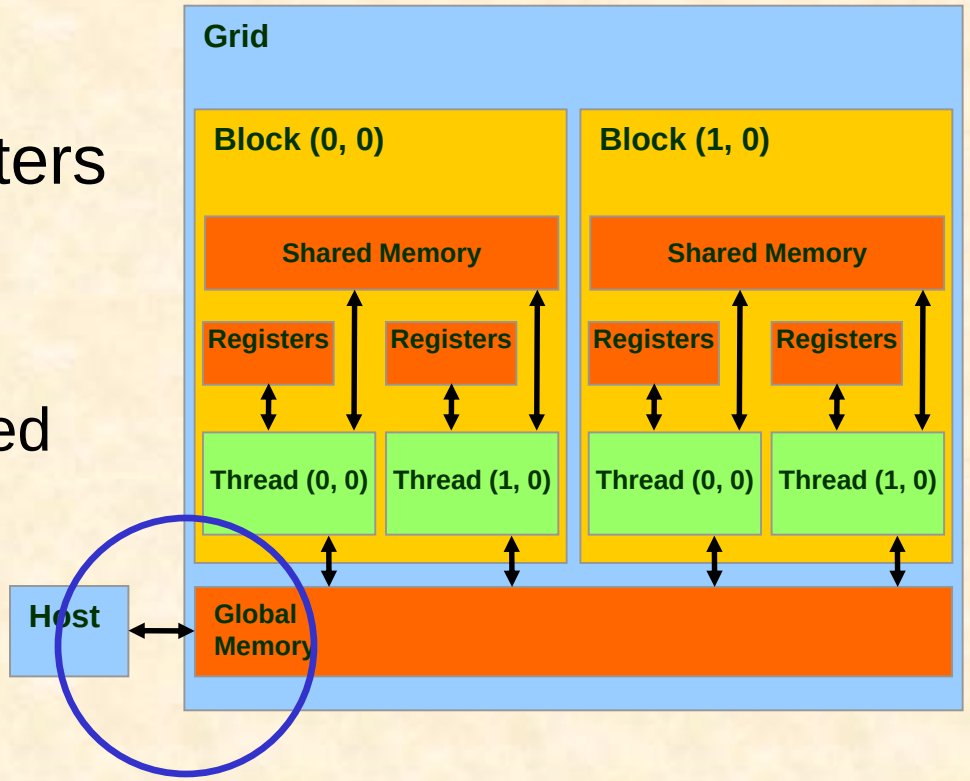


- Code example:
  - Allocate a  64 * 64 single precision float array
  - Attach the allocated storage to Md
  - “d” is often used to indicate a device data structure

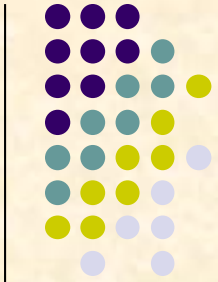
```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

# CUDA Host-Device Data Transfer

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



# CUDA Host-Device Data Transfer (cont.)

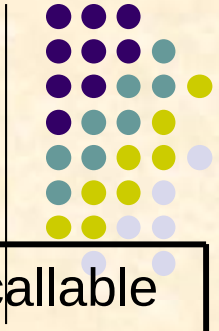


- Code example:
  - Transfer a  64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

**cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);**

**cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);**

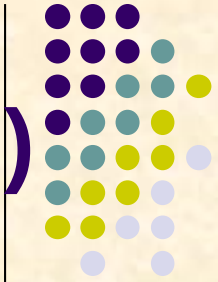
# CUDA Function Declarations



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

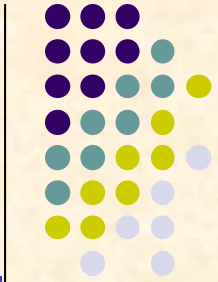
- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

# 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



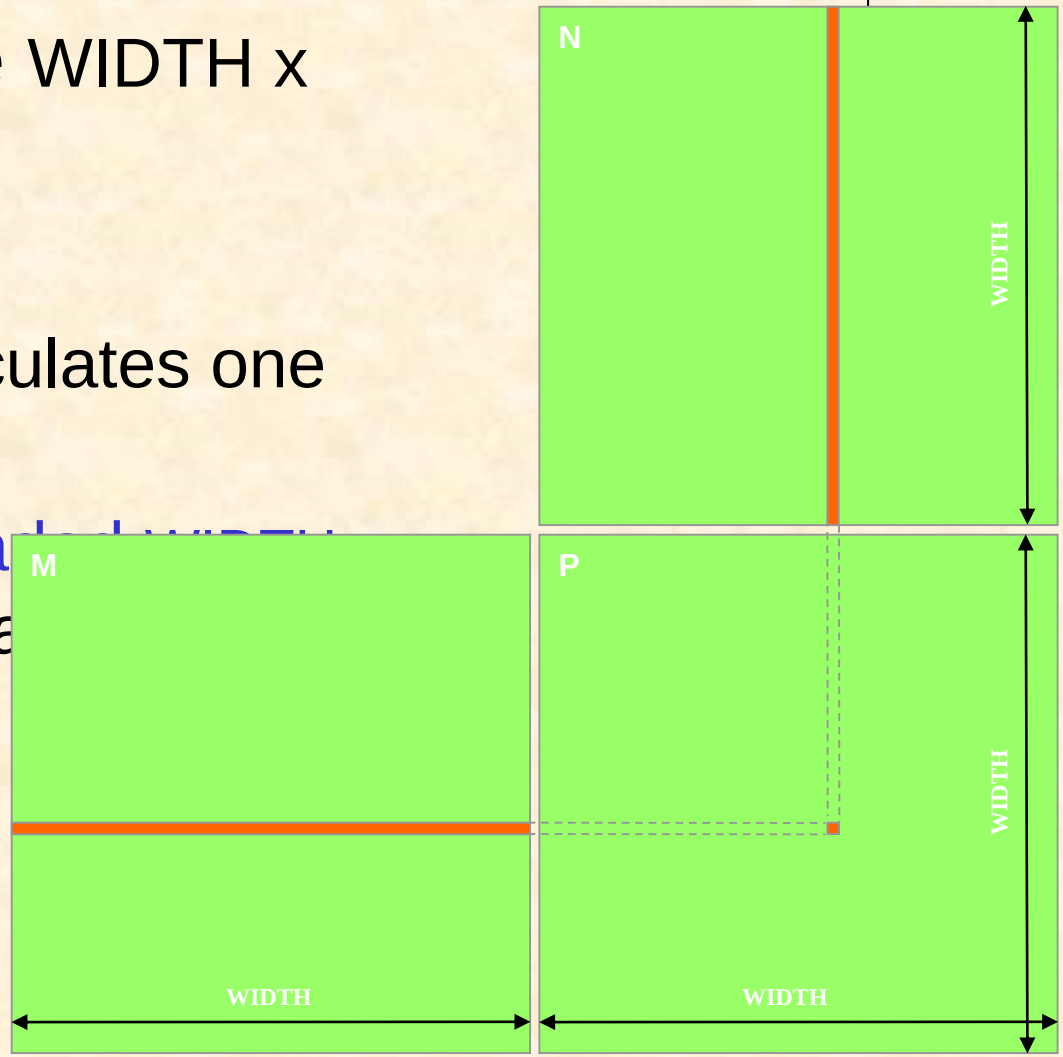
- A kernel function must be called with an execution configuration:

```
__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
   >>>(...);
```

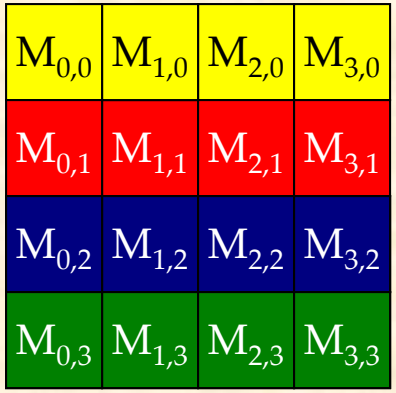
- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

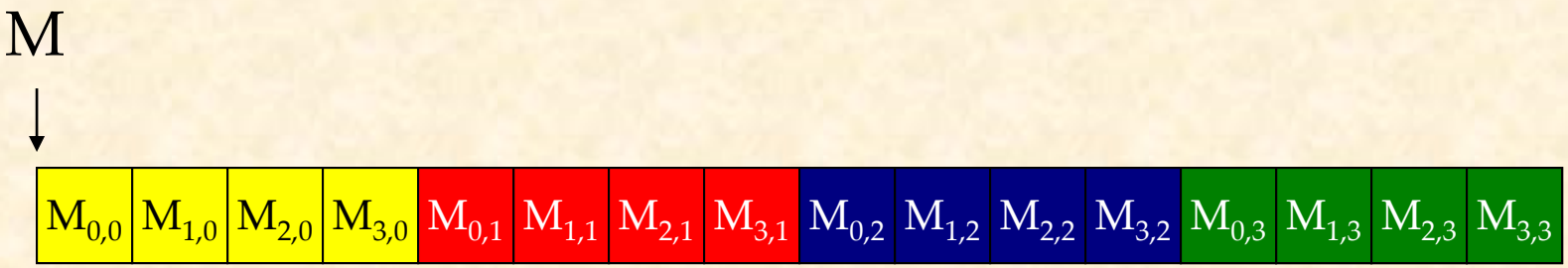
# Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loa times from globa



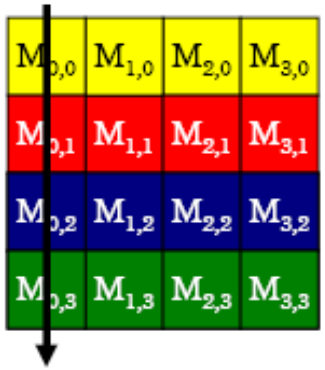
# Memory Layout of a Matrix in C

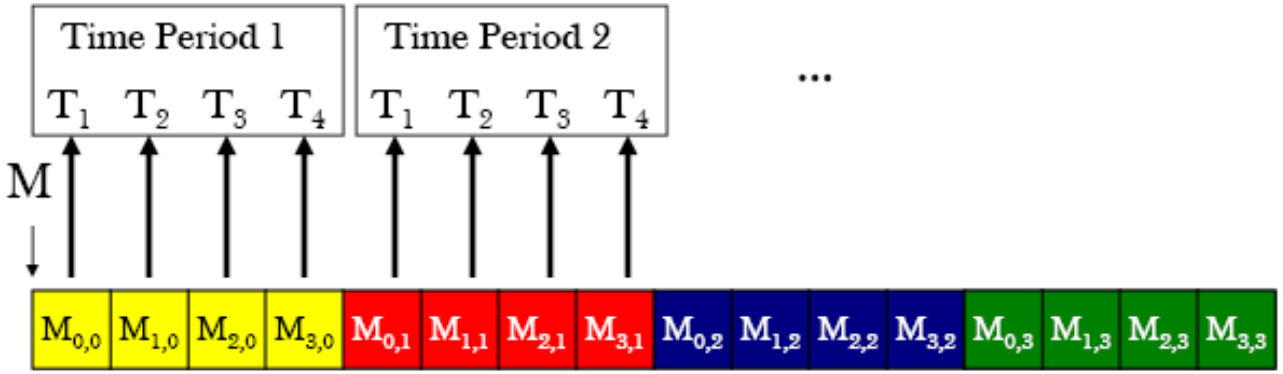


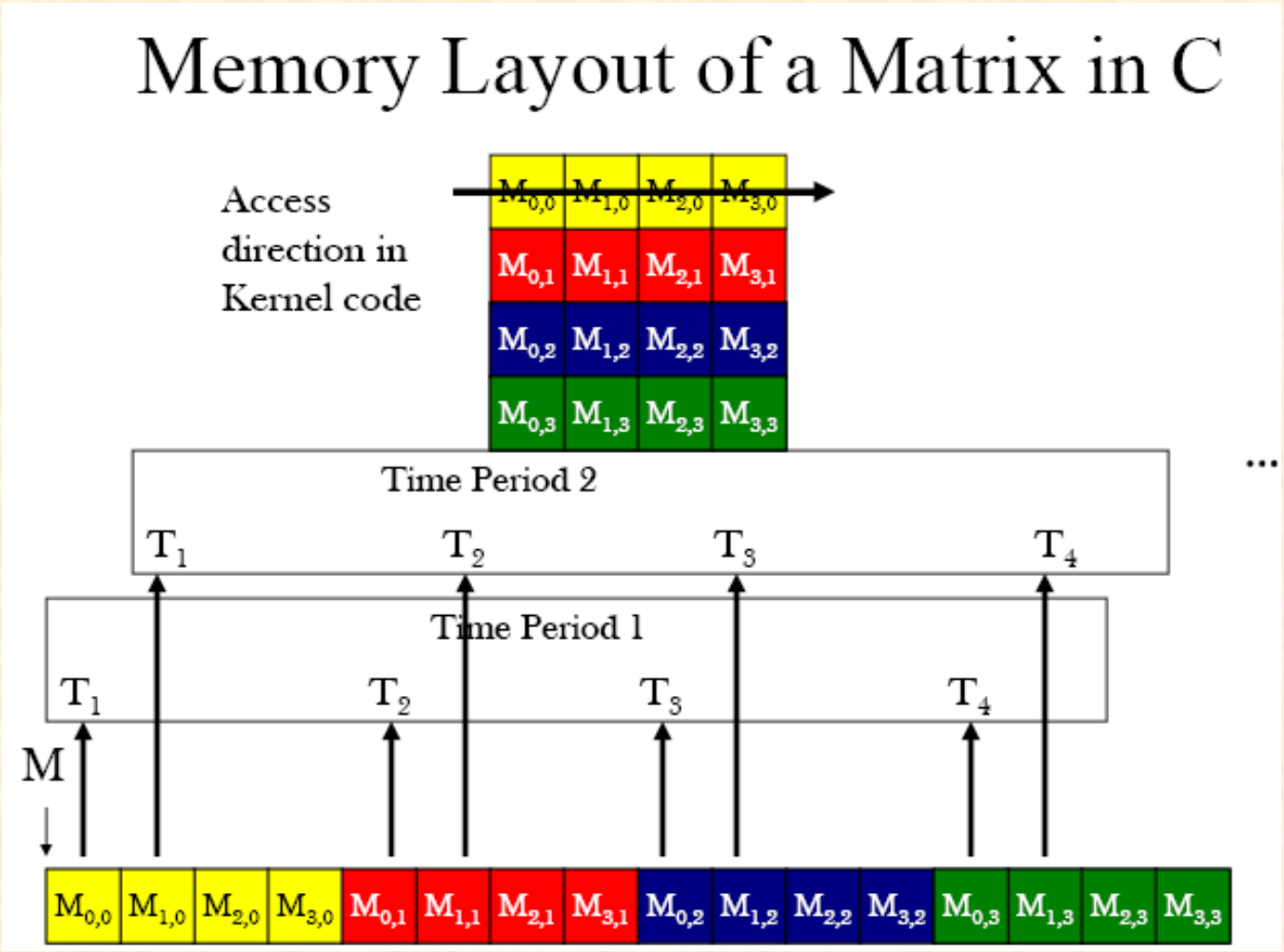
# Memory Layout of a Matrix in C

Access direction in Kernel code



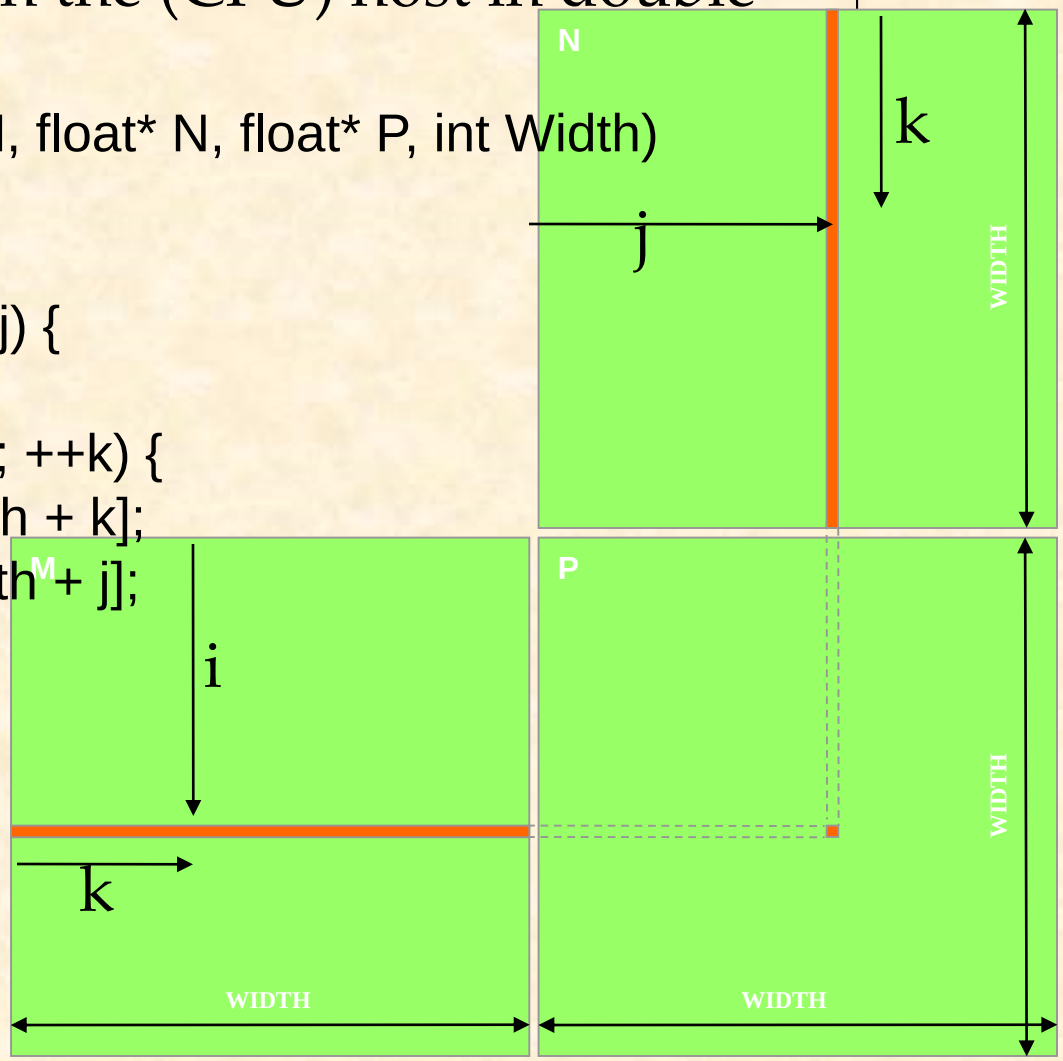



# Memory Layout of a Matrix in C



# Step 1: Matrix Multiplication
# A Simple Host Version in C

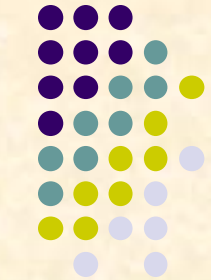
```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



# Step 2: Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;
   …
1. // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

     // Allocate P on the device
    cudaMalloc(&Pd, size);
```

# Step 3: Output Matrix Data Transfer (Host-side Code)

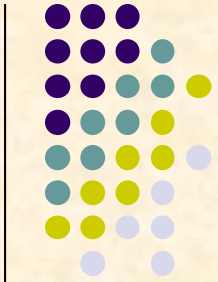


2. // Kernel invocation code – to be shown later
   …

3. // Read P from the device
   **cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);**

   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }

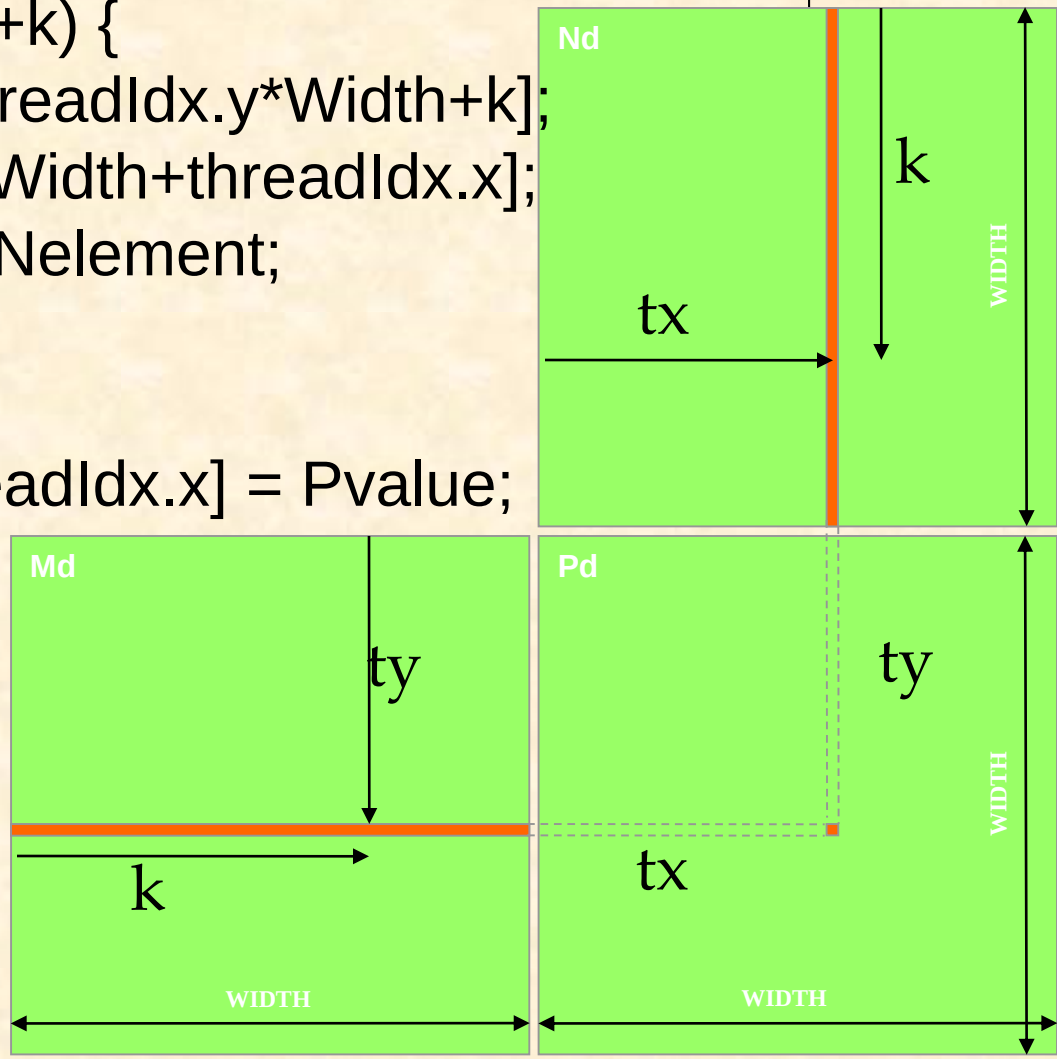
# Step 4: Kernel Function



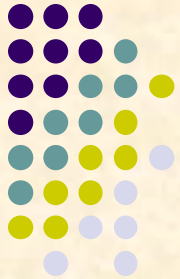
```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

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

# Step 4: Kernel Function (cont.)

```
for (int k = 0; k < Width; ++k) {
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
}

Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



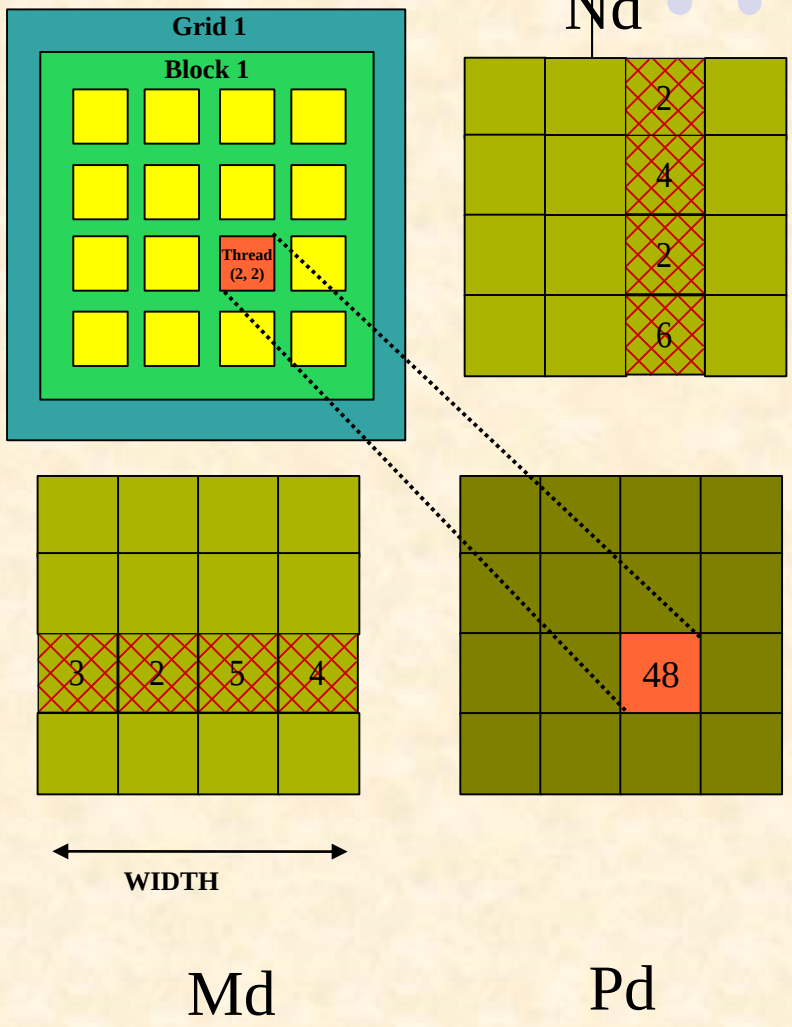
# Step 5: Kernel Invocation (Host-side Code)



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

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

# Only One Thread Block Used

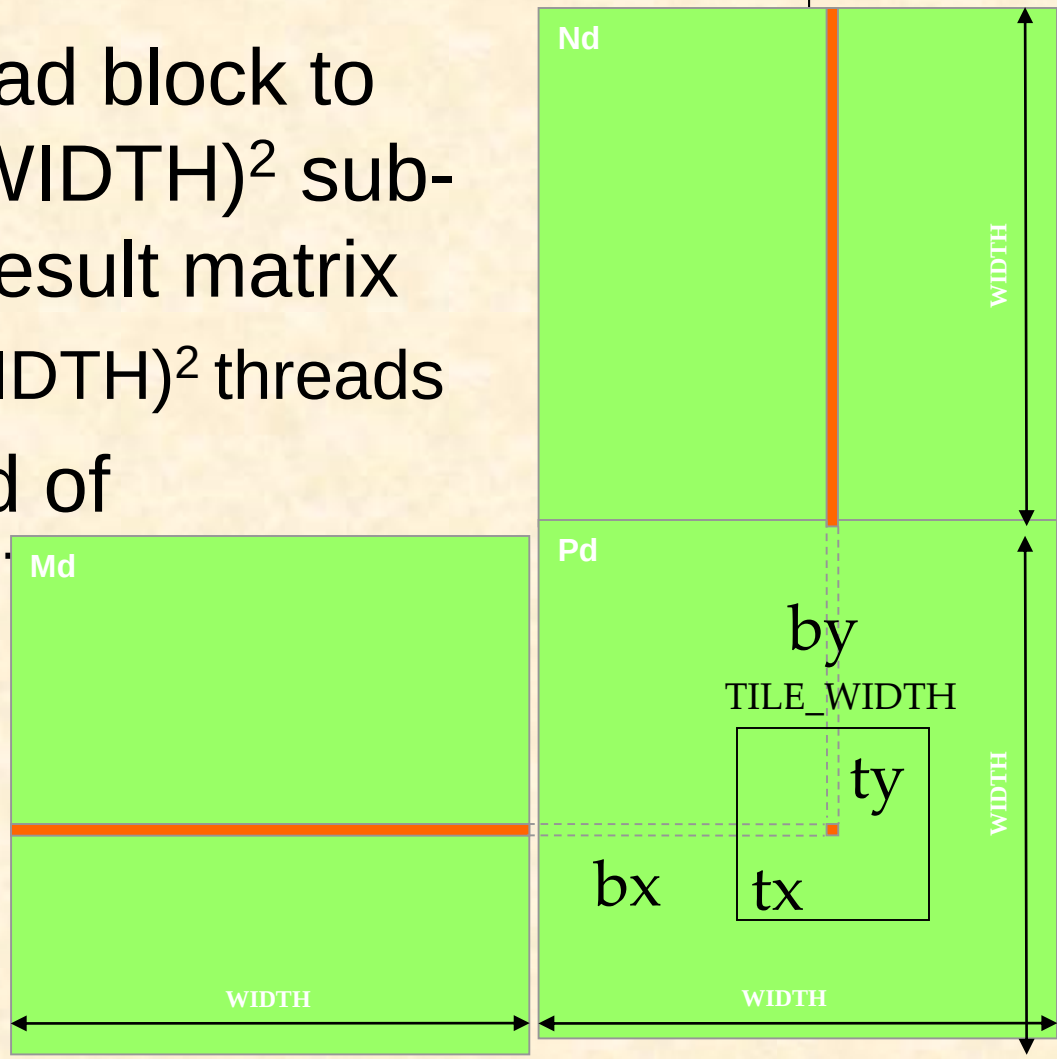
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)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



# Step 6: Handling Arbitrary Sized Square Matrices

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

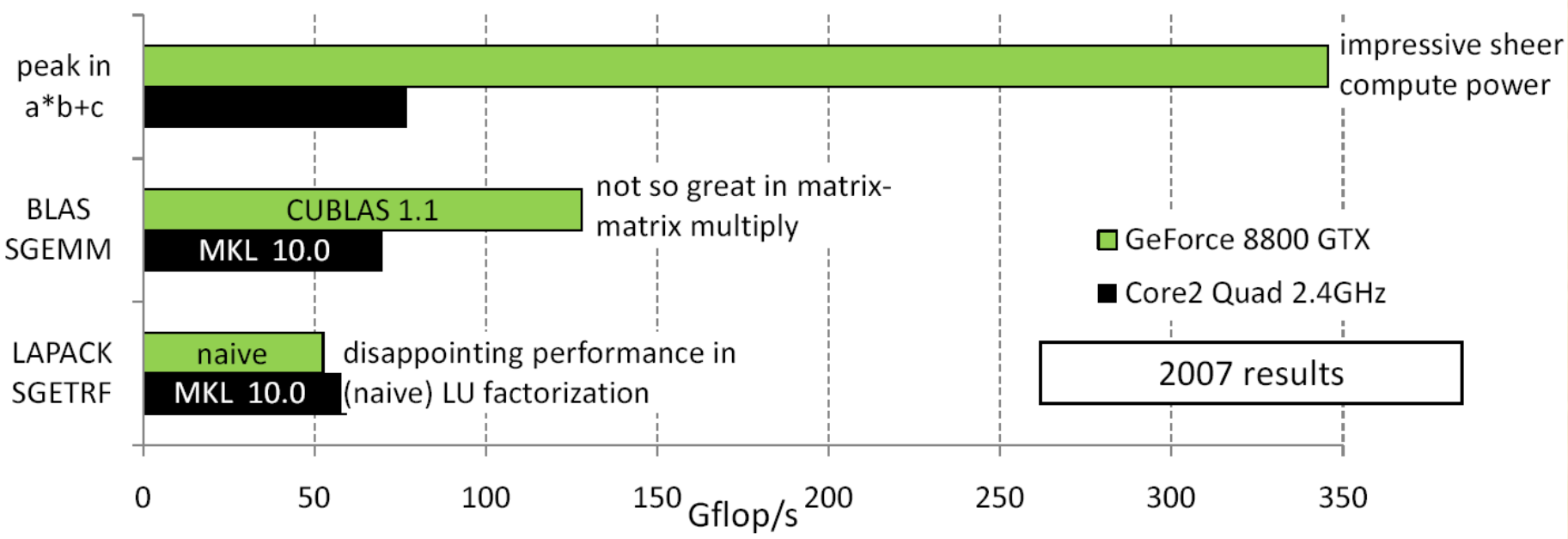
You still need to put a loop around the kernel call for cases where WIDTH/TILE_WIDTH is greater than max grid size (64K)!

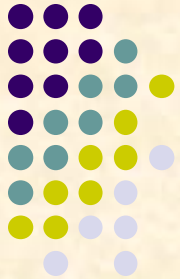


# SGEMM

- Vasily Volkov, James Demmel: *Benchmarking GPUs to tune dense linear algebra*. SC 2008

*Some following slides are from the presentation of the above paper at SC'08.*

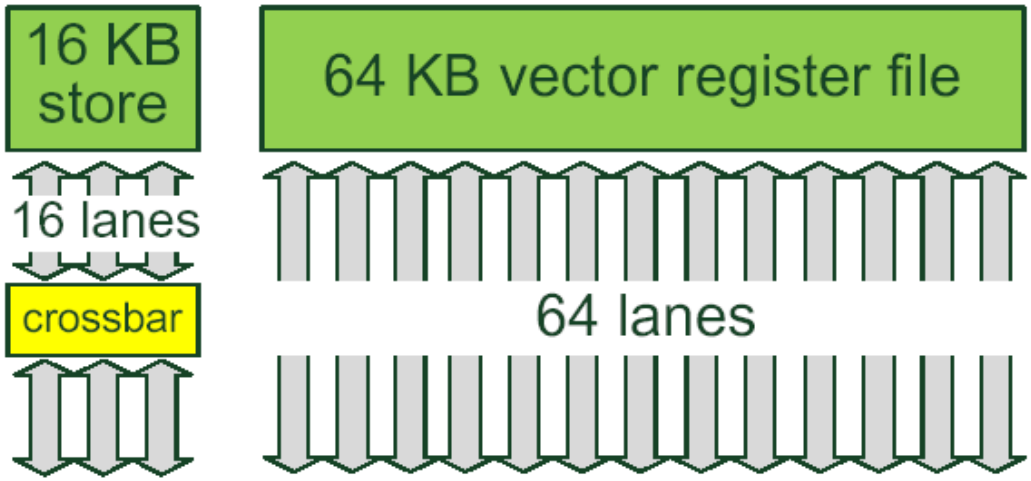

peak in a*b+c
BLAS SGEMM
LAPACK SGETRF
CUBLAS 1.1
MKL 10.0
naive
MKL 10.0
impressive sheer compute power
not so great in matrix-matrix multiply
disappointing performance in (naive) LU factorization
GeForce 8800 GTX
Core2 Quad 2.4GHz
2007 results
0
50
100
150
200
250
300
350
Gflop/s



- Memory bandwidth scales with number of memory controllers

| GPU (NVIDIA GeForce) | 8600 GTS | 9800 GTX | GTX 280 |
|---|---|---|---|
| Processor cores | 4 | 16 | 30 |
| Compute capability (a+b*c) | 93 Gflop/s | 429 Gflop/s | 624 Gflop/s |
| Memory controllers | 2 | 4 | 8 |
| Memory bandwidth | 32 GB/s | 70 GB/s | 141 GB/s |

# GPU Memory Hierarchy




- Register file is the fastest and the largest on-chip memory
  - **Keep as much data as possible in registers**
  - However, register file is constrained to vector operations
  - Can live with it — vectorized codes are common in HPC
- Shared memory permits indexed and shared access
  - However, it is 2–4x smaller and has 4x lower bandwidth than registers
    - Only 1 operand in shared memory is allowed versus 4 register operands
  - Moreover, some instructions run slower if using shared memory
  - **Use shared memory as a communication device only**
  - Avoid communication to improve performance

# Peak Throughput in Multiply-and-Add

- How much parallelism is *enough* to get the peak?

- Run **1 thread per processor core**
  - Purpose: smallest amount that can control all computing resources
- Assume **sufficient instruction-level parallelism** in the program
  - Purpose: hide pipeline latency
- Choose the shortest vector length that yields the peak
  - Purpose: satisfy inherent data-parallelism constraints
- Result: **98% of arithmetic peak at VL = 64**
  - Therefore, VL=64 is recommended for all compute-bound codes

- However, we never could surpass 66% of peak is using an operand in shared memory
  - We believe this is an inherent bottleneck in the architecture
  - We use this number in the throughput bounds below

# Matrix-Matrix Multiply: $C = C + A*B$

- GPU requires using block algorithms in matrix-matrix multiply:
  - Peak rates on one of the latest GPUs are 624 Gflop/s and 141 GB/s
  - This corresponds to 0.23 bytes per flop
  - But naïve matrix-matrix multiply requires 4 bytes per flop
  - Thus, it is bandwidth-bound unless data is reused 18 times
  - Using *M*x*N* blocks in *C* yields $2/(1/M+1/N)$ average reuse
- Use vector algorithms to efficiently use vector registers
  - Such as used on IBM 3090 Vector Facility and Cray X1:
  - Keep *A*'s and *C*'s blocks in registers
  - Keep *B*'s block in a shared storage
  - No other sharing is needed if *C*'s height = VL. We know VL=64 is best
- Choose large enough width of *C*'s block
  - 16 is enough as $2/(1/64+1/16) = 26$-way reuse
- Choose a convenient thickness for *A*'s and *B*'s blocks

```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;
    __shared__ float bs[16][17];
    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    const float *Blast = B + k;
    do
    {
#pragma unroll
        for( int i = 0; i < 16; i += 4 )
            bs[threadIdx.x][threadIdx.y+i]  = B[i*ldb];
        B += 16;
        __syncthreads();
#pragma unroll
        for( int i = 0; i < 16; i++, A += lda )
        {
            c[0] += A[0]*bs[i][0];    c[1] += A[0]*bs[i][1];    c[2] += A[0]*bs[i][2];    c[3] += A[0]*bs[i][3];
            c[4] += A[0]*bs[i][4];    c[5] += A[0]*bs[i][5];    c[6] += A[0]*bs[i][6];    c[7] += A[0]*bs[i][7];
            c[8] += A[0]*bs[i][8];    c[9] += A[0]*bs[i][9];    c[10] += A[0]*bs[i][10];c[11] += A[0]*bs[i][11];
            c[12] += A[0]*bs[i][12];c[13] += A[0]*bs[i][13];c[14] += A[0]*bs[i][14];c[15] += A[0]*bs[i][15];
        }
        __syncthreads();
    } while( B < Blast );
    for( int i = 0; i < 16; i++, C += ldc )
        C[0] = alpha*c[i] + beta*C[0];
}
```

Compute pointers to the data

Declare the on-chip storage

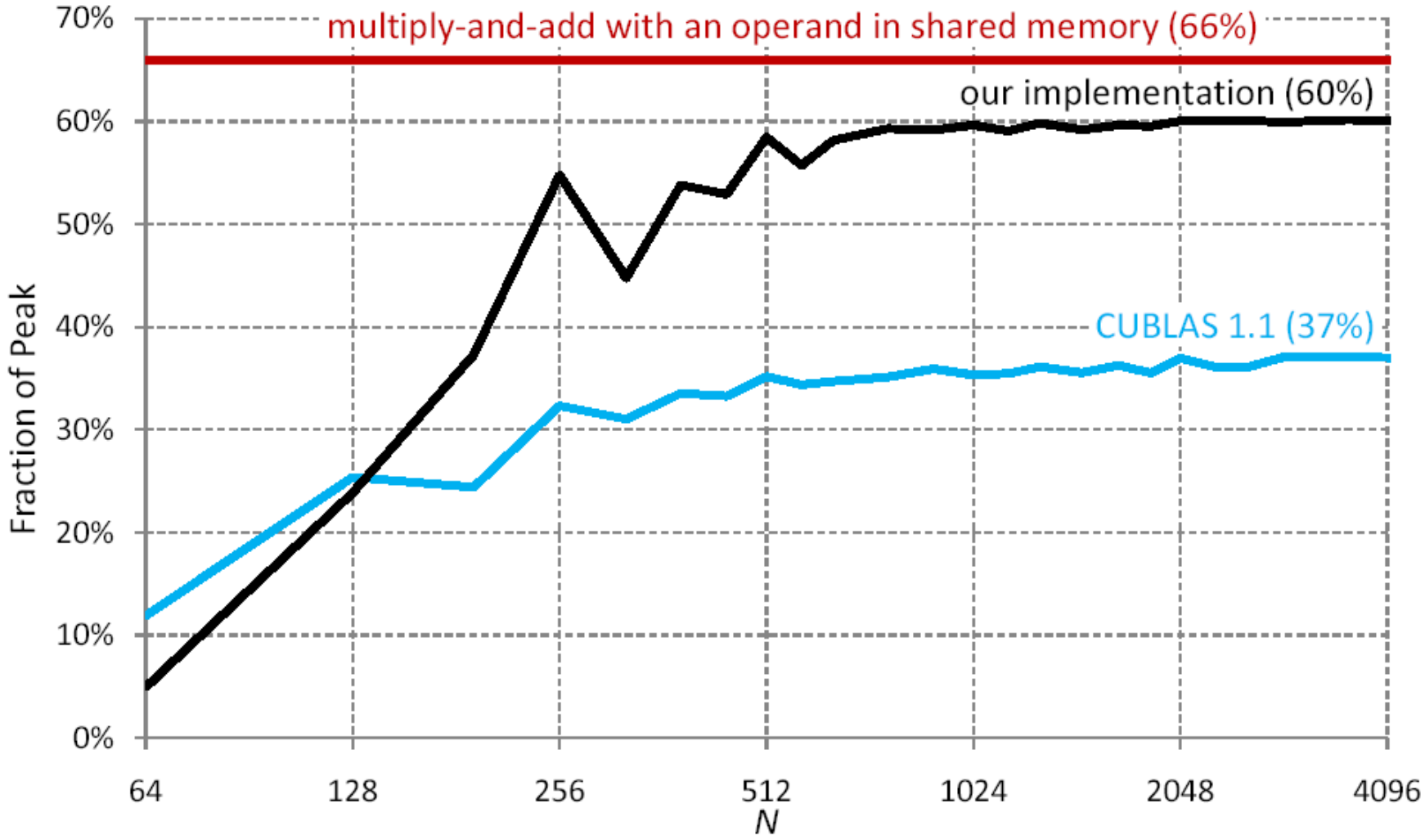
Read next B's block

The bottleneck:
Read A's columns
Do Rank-1 updates

Store C's block to memory

# Our code vs. CUBLAS 1.1

Performance in multiplying two *N*x*N* matrices on GeForce 8800 GTX:




What causes CUBLAS 1.1 to run slower than our code?

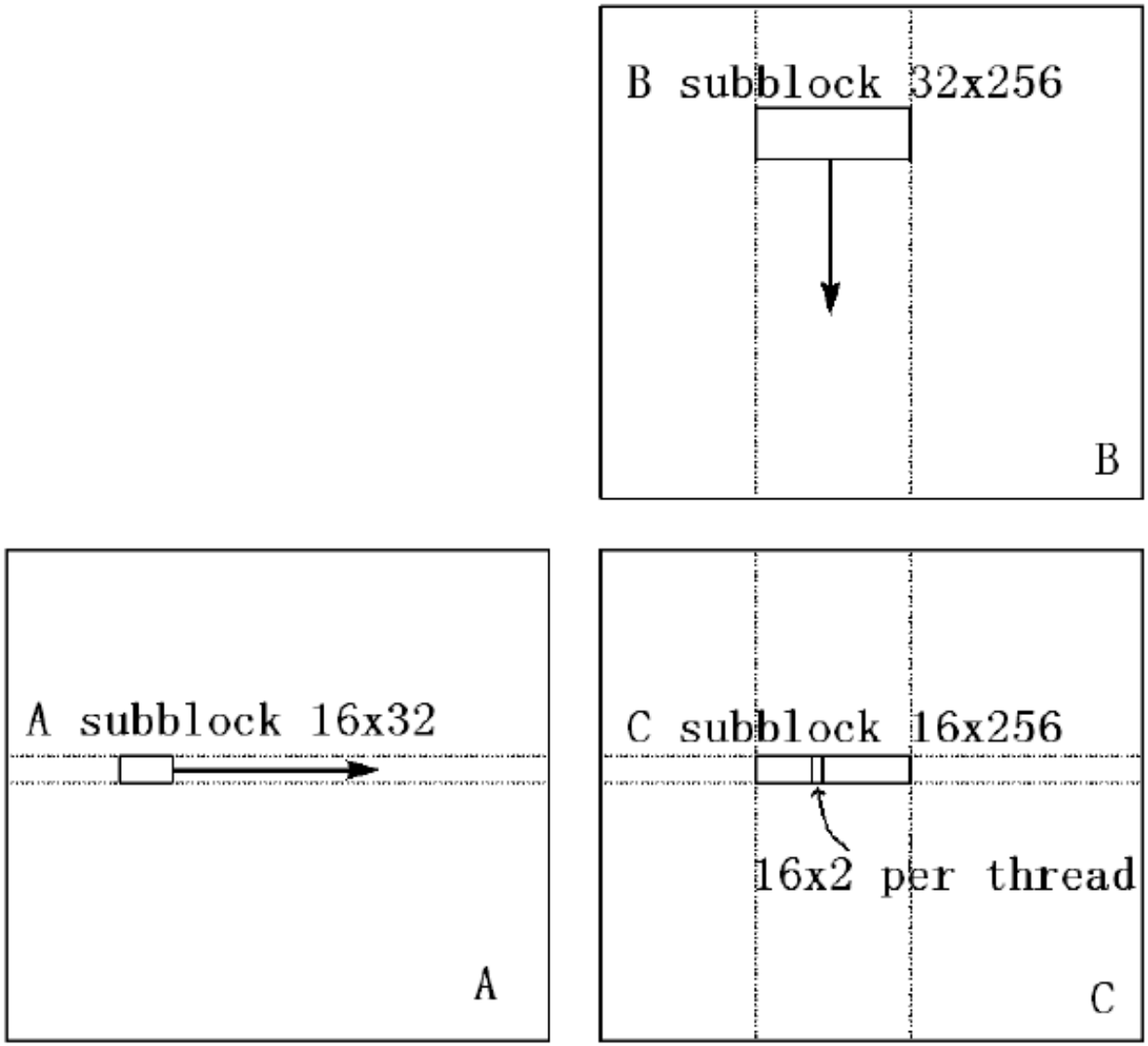
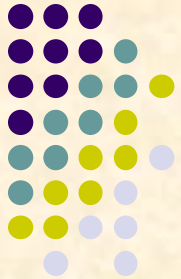



**Figure 3: Matrix multiplication: A×B=C with all arrays in row major**

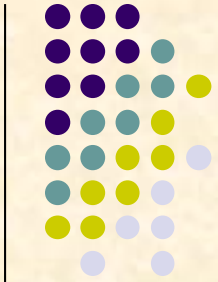


**Hand-Tuned SGEMM on GT200 GPU**

Lung-Sheng Chien

Department of Mathematics, Tsing Hua university, R.O.C. (Taiwan)

| | GTX295[1] | GTX285 | TeslaC1060 |
|---|---|---|---|
| **# of Streaming Processor** | 240 | 240 | 240 |
| **Core Frequency** | 1242MHz | 1476 MHz | 1.3 GHz |
| **Memory Speed** | 999MHz | 1242 MHz | 800 MHz |
| **Memory Interface** | 448-bit (7 channel) | 512-bit (8 channel) | 512-bit (8 channel) |
| **Memory Bandwidth (GB/s)** | 112 | 159 | 102 |
| **SP, peak (Gflop/s)** | 894 | 1063 | 933 |
| **SP without dual issue** | 596.2 | 708.5 | 624 |
| **DP, peak (Gflop/s)** | 74.5 | 88.6 | 78 |
| **DRAM (MByte)** | 896 | 1024 | 4096 |

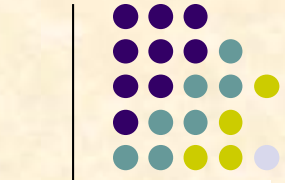


There are two kinds of MAD when operands are "float",

(1) "MAD dest, src1, src2, src3" corresponds to $dest = src1 \times src2 + src3$ where dest, src1, src2 and src3 are all registers.

(2) "MAD dest, [smem], src2, src3" corresponds to $dest = [smem] \times src2 + src3$ where [smem] denotes shared memory.

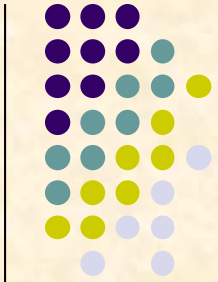
These two MAD operations have different pipeline latency and throughput. In our method, we change the later in Volkov's code to the former to improve performance.



```
__shared__ float b[16];
int threadNum = threadIdx.x;
unsigned int start_time = 1664;
unsigned int end_time = 1664;
for( int j = threadNum ; j < 16 ; j+=NUM_THREADS){
  b[j] = data[j] ;
}
float A_reg = data[0] ;
float c_reg = data[2] ;
__syncthreads();

start_time = clock();
#pragma unroll
for( int j = 0 ; j < 16 ; j++){
#pragma unroll
  for( int i = 0 ; i < 16 ; i++){
    A_reg = A_reg * b[i] + c_reg ;
  }
}
end_time = clock();
__syncthreads();

timings[threadNum].start_time = start_time;
timings[threadNum].end_time = end_time;
```

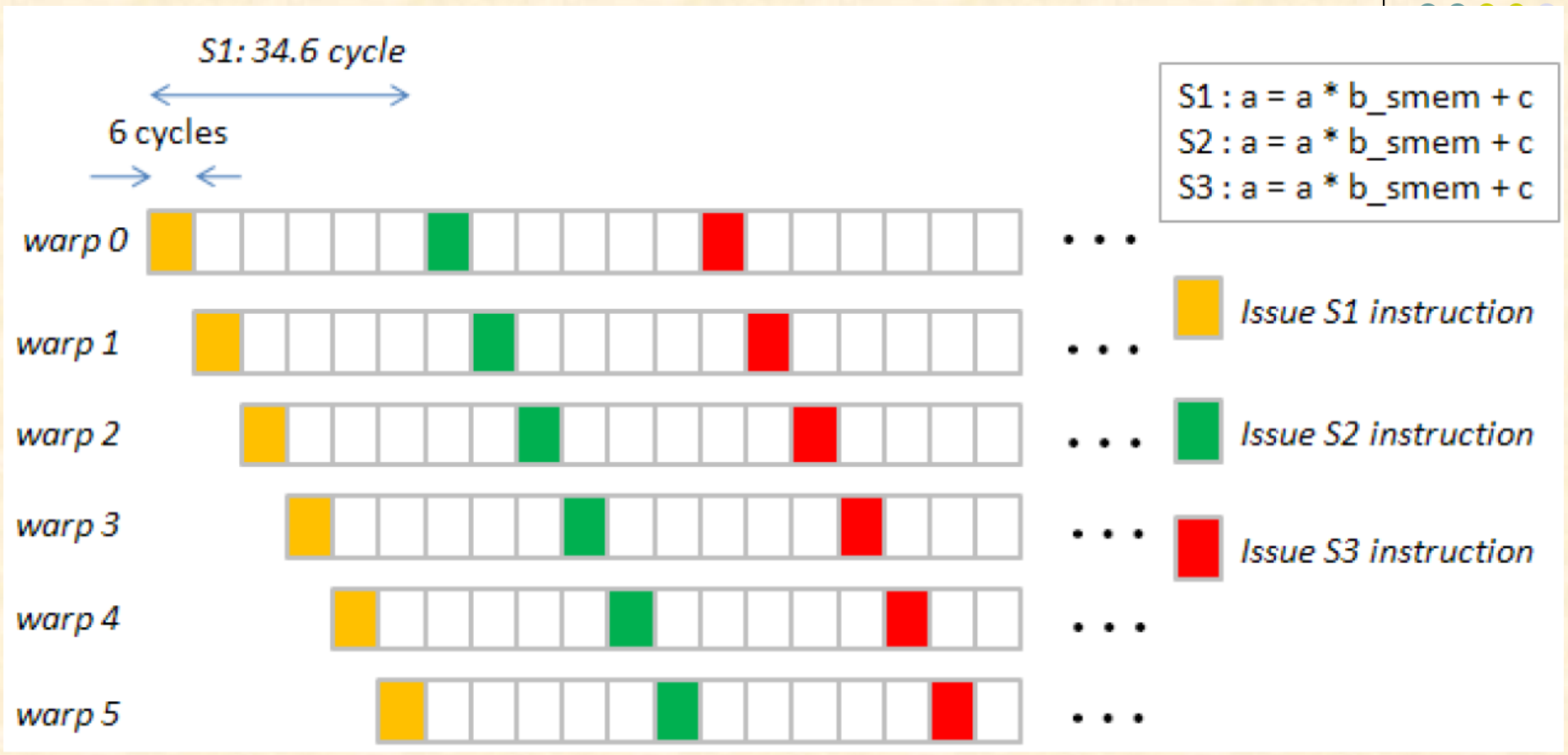
result of decuda

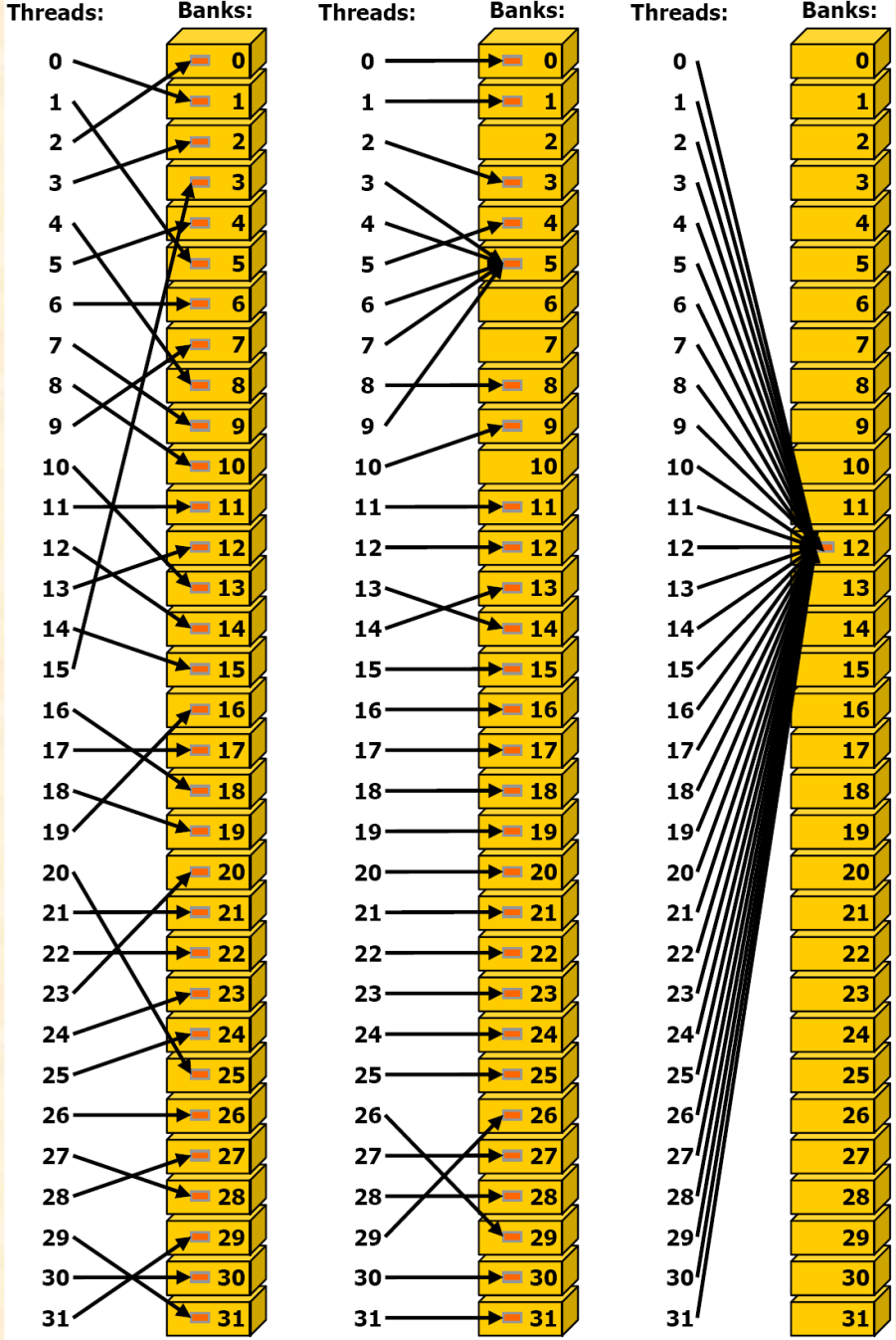
```
25 mov.b32 $r1, s[0x0010]
26 mov.u32 $r3, g[$r1]
27 add.b32 $r1, s[0x0010], 0x00000008
28 mov.u32 $r2, g[$r1]
29 bar.sync.u32 0x00000000
30 mov.b32 $r1, %clock
31 shl.u32 $r1, $r1, 0x00000001
32 mad.rn.f32 $r3, s[0x0020], $r3, $r2
33 mad.rn.f32 $r3, s[0x0024], $r3, $r2
   . . .
286 mad.rn.f32 $r3, s[0x0058], $r3, $r2
287 mad.rn.f32 $r2, s[0x005c], $r3, $r2
288 mov.b32 $r3, %clock
289 shl.u32 $r3, $r3, 0x00000001
290 bar.sync.u32 0x00000000
```



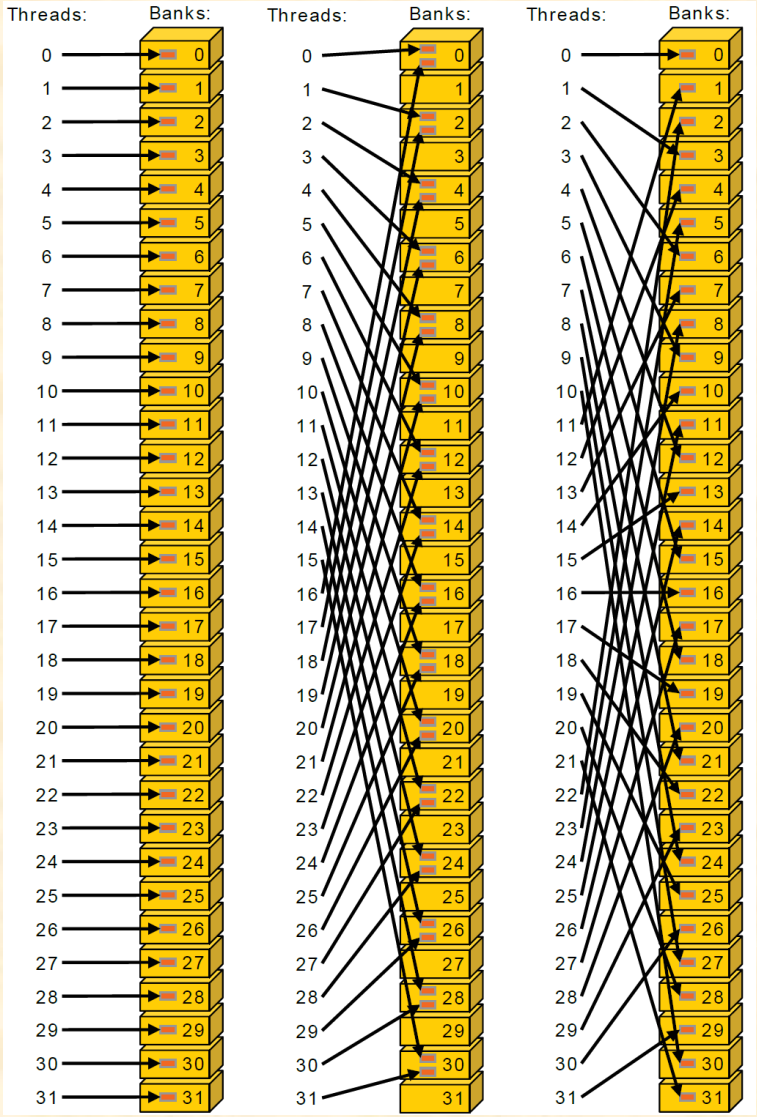
| NUM_THREADS | 1 | 64 | 128 | 192 | 224 | 256 | 288 | 320 | 384 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|
| Minimum time | 34.6 | 34.6 | 34.7 | 38.6 | 42.4 | 46.6 | 54.0 | 60.2 | 72.1 | 96.1 |
| Maximum time | 34.6 | 34.6 | 34.8 | 39.3 | 43.2 | 49.5 | 54.7 | 60.6 | 72.7 | 96.9 |
| Total time for one a = a*b_smem +c | 34.6 | 34.6 | 34.6 | 36 | 42 | 48 | 54 | 60 | 72 | 96 |

Table 2: average number of cycles per "MAD dest, [smem], src2, src3" on TeslaC1060. Pipeline latency is 34.6 cycle and throughput is 6 cycle /warp.


S1: 34.6 cycle
6 cycles
S1 : a = a * b_smem + c
S2 : a = a * b_smem + c
S3 : a = a * b_smem + c
warp 0
warp 1
warp 2
warp 3
warp 4
warp 5
Issue S1 instruction
Issue S2 instruction
Issue S3 instruction




Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right: Conflict-free broadcast access (all threads access the same word).




**Left**

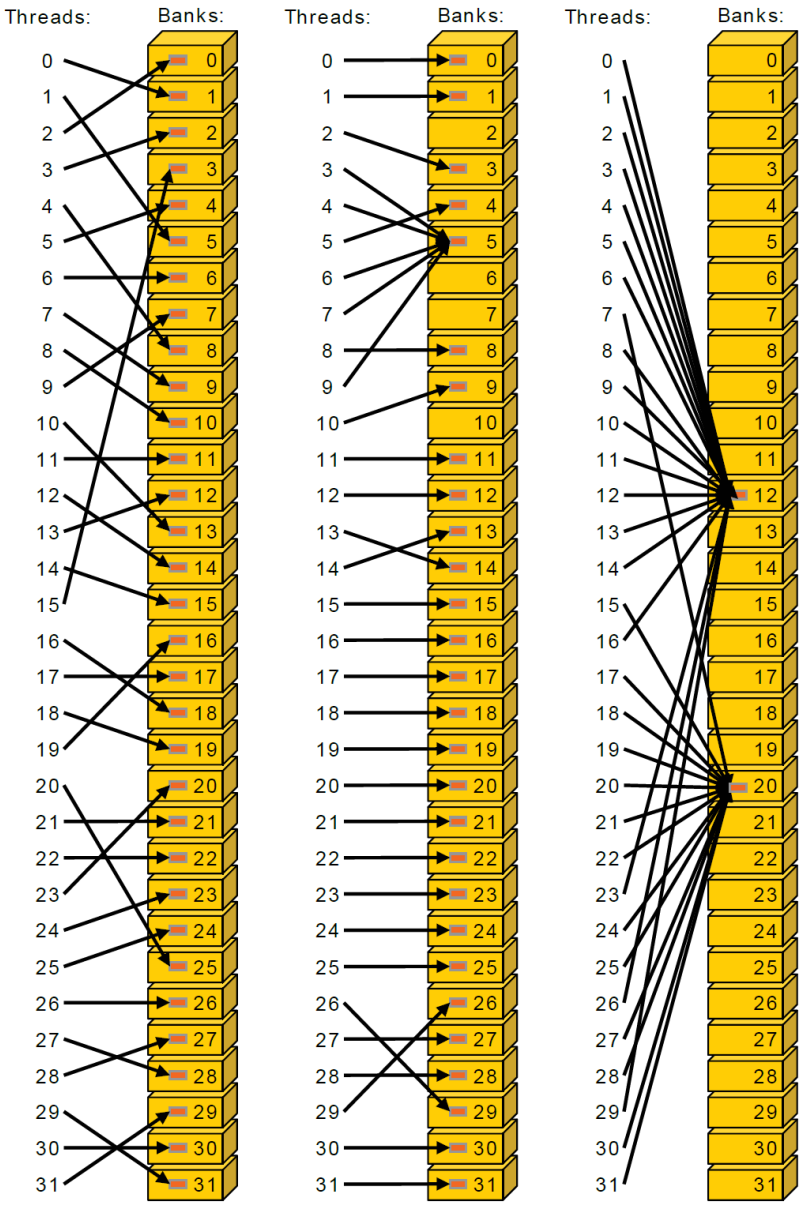
Linear addressing with a stride of one 32-bit word (no bank conflict).

**Middle**

Linear addressing with a stride of two 32-bit words (two-way bank conflict).

**Right**

Linear addressing with a stride of three 32-bit words (no bank conflict).




**Left**

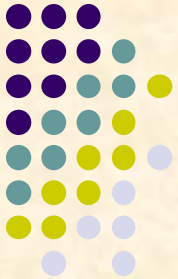
Conflict-free access via random permutation.

**Middle**

Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

**Right**

Conflict-free broadcast access (threads access the same word within a bank).



| | Compute Capability | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Technical Specifications** | **3.0** | **3.2** | **3.5** | **3.7** | **5.0** | **5.2** | **5.3** | **6.0** | **6.1** | **6.2** | **7.0** | **7.5** |
| Maximum number of resident grids per device (Concurrent Kernel Execution) | 16 | 4 | 32 | | | | 16 | 128 | 32 | 16 | 128 | |
| Maximum dimensionality of grid of thread blocks | 3 | | | | | | | | | | | |
| Maximum x-dimension of a grid of thread blocks | $2^{31}-1$ | | | | | | | | | | | |
| Maximum y- or z-dimension of a grid of thread blocks | 65535 | | | | | | | | | | | |

| | Compute Capability | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Technical Specifications** | **3.0** | **3.2** | **3.5** | **3.7** | **5.0** | **5.2** | **5.3** | **6.0** | **6.1** | **6.2** | **7.0** | **7.5** |
| Maximum dimensionality of thread block | 3 | | | | | | | | | | | |
| Maximum x- or y-dimension of a block | 1024 | | | | | | | | | | | |
| Maximum z-dimension of a block | 64 | | | | | | | | | | | |
| Maximum number of threads per block | 1024 | | | | | | | | | | | |
| Warp size | 32 | | | | | | | | | | | |
| Maximum number of resident blocks per multiprocessor | 16 | | | | 32 | | | | | | | 16 |
| Maximum number of resident warps per multiprocessor | 64 | | | | | | | | | | | 32 |
| Maximum number of resident threads per multiprocessor | 2048 | | | | | | | | | | | 1024 |
| Number of 32-bit registers per multiprocessor | 64 K | | | 128 K | 64 K | | | | | | | |
| Maximum number of 32-bit registers per thread block | 64 K | 32 K | 64 K | | | | 32 K | 64 K | | 32 K | 64 K | |
| Maximum number of 32-bit registers per thread | 63 | 255 | | | | | | | | | | |
| Maximum amount of shared memory per multiprocessor | 48 KB | | | 112 KB | 64 KB | 96 KB | 64 KB | | 96 KB | 64 KB | 96 KB | 64 KB |
| Maximum amount of shared memory per thread block [25] | 48 KB | | | | | | | | | | 96 KB | 64 KB |
| Number of shared memory banks | 32 | | | | | | | | | | | |
| Amount of local memory per thread | 512 KB | | | | | | | | | | | |
| Constant memory size | 64 KB | | | | | | | | | | | |
| Cache working set per multiprocessor for constant memory | 8 KB | | | | | | | 4 KB | 8 KB | | | |
| Cache working set per multiprocessor for texture memory | Between 12 KB and 48 KB | | | | | | | Between 24 KB and 48 KB | | | 32 ~ 128 KB | 32 or 64 KB |
| Maximum width for a 1D texture reference bound to a CUDA array | 65536 | | | | | | | | | | | |

# H.6. Compute Capability 7.x

## H.6.1. Architecture

A multiprocessor consists of:

- 64 FP32 cores for single-precision arithmetic operations,
- 32 FP64 cores for double-precision arithmetic operations, [26]
- 64 INT32 cores for integer math,
- 8 mixed-precision Tensor Cores for deep learning matrix arithmetic
- 16 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

A multiprocessor statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

A multiprocessor has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 128 KB (*Volta*) or 96 KB (*Turing*).