# Introduction to GPU Programming with CUDA

**Mark Gates**

Supercomputing ’19

Nov 17, 2019

Examples and slides available at: icl.utk.edu/~mgates3/gpu-tutorial/

# Outline

**GPU Architecture**

- Threads & thread blocks
- Streaming Multiprocessor (SM)
- Specs

**Ex 0: kernel**

- Kernel syntax
- Compiling

**Ex 1: matrix add**

- Basic Runtime API
- Thrust library
- Coalesced memory loads

**Ex 2: matrix norm**

- Shared memory
- Reductions

**Ex 3: matrix multiply**

- Reusing data in shared memory
- Kernel strategy

**Optimization**

**Ex 4: hybrid computing**

- Streams
- Profiling

**Runtime API**

- Devices, Streams, Events, Memory

# GPU Architecture

Threads and thread blocks

Streaming Multiprocessors (SMs)

Hiding memory latency

# Why GPUs?

**High peak performance (Gflop/s)**

**High speed memory (GDDR, HBM, MCDRAM)**

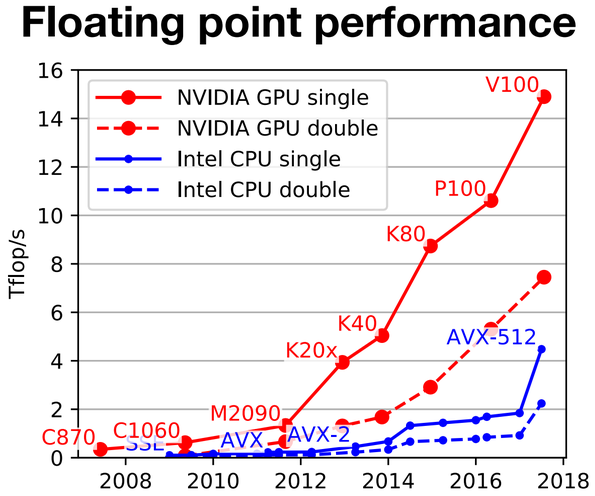
**High parallelism — 1000s of threads**

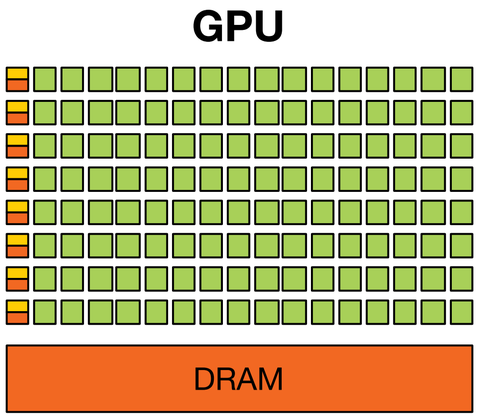
**Simpler, low-overhead execution model**

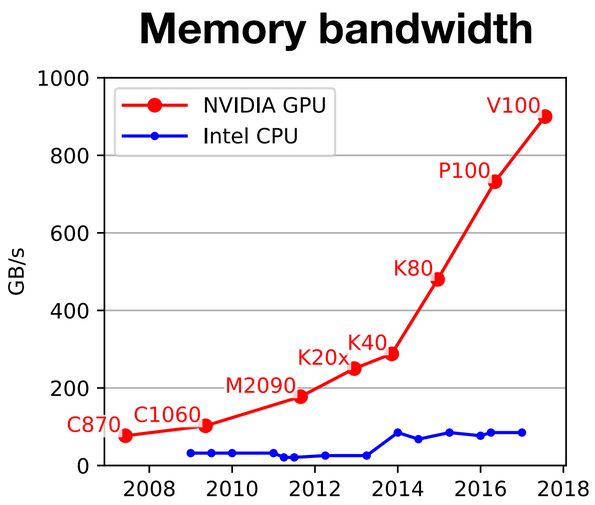
- No out-of-order execution
- No speculative execution
- Threads run in lock-step
- Blocks are independent

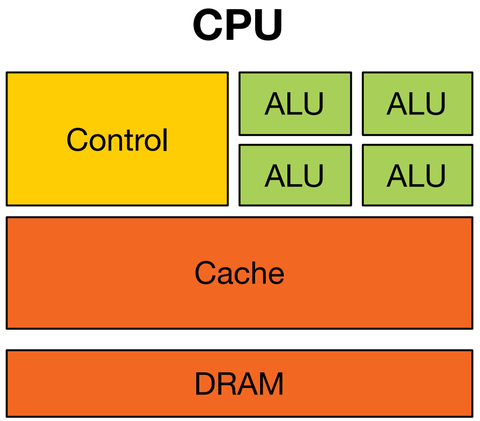
**Devotes more transistors to compute (green)**

**Energy efficient — high flop/s / watt**




Data from NVIDIA and Intel specs, diagrams from NVIDIA manuals

# CUDA: Threads

**Thread block (1D, 2D, or 3D)**

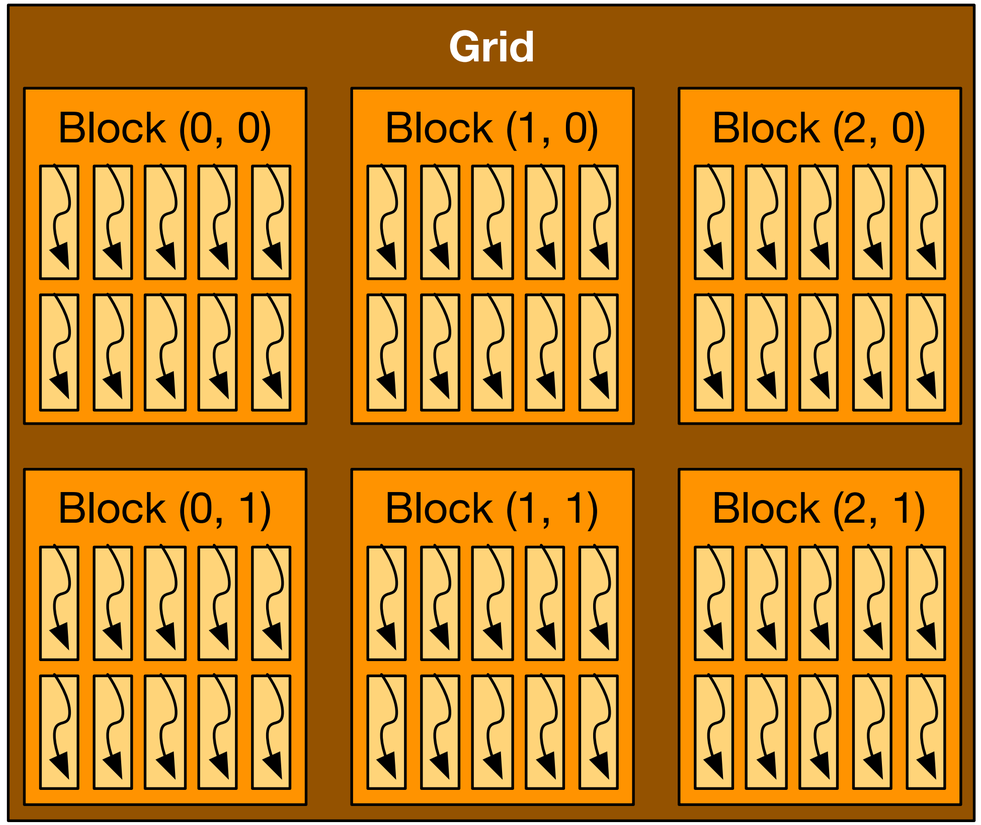
- Example: 2D grid of 5x2 threads
- **dim3 threads( 5, 2 );**
- Typically, # threads is multiple of 32 (warp size)
- Think of warp size roughly as **vector length**
- Max. 1024 threads

**Grid of thread blocks (1D, 2D, or 3D)**

- Example: 2D grid of 3x2 blocks
- **dim3 blocks( 3, 2 );**
- Max. $2^{31}-1$ x 65535 x 65535

**Threads within block can synchronize**

**Blocks cannot synchronize — executed in any order!**



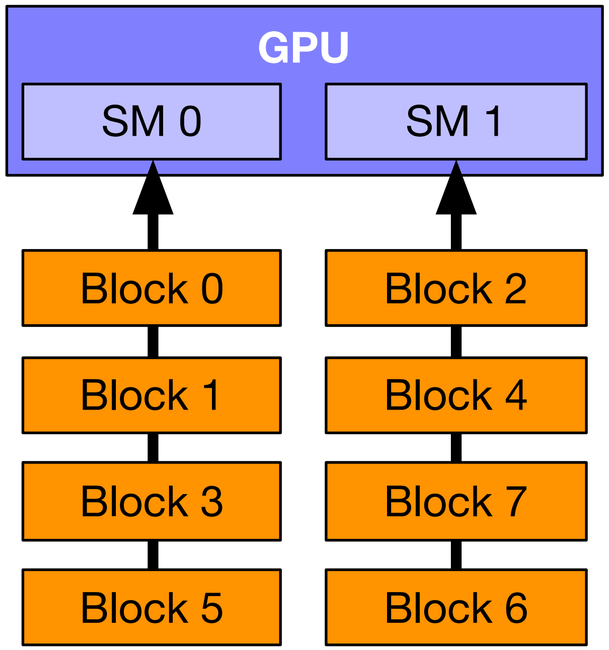

**Total: 6 blocks * 10 threads = 60 threads**

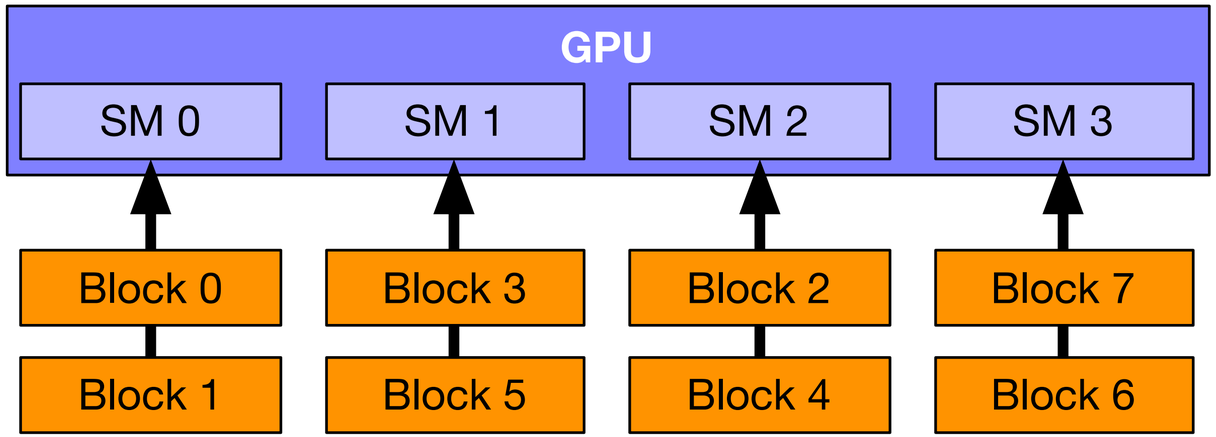
For limits see: CUDA C Programming Guide, Appendix H: Compute Capabilities

# CUDA: Streaming Multiprocessors (SMs)

**GPUs have several SM processors**

- Each SM has some number of CUDA cores (varies: 64–192)
- GTX 1060 has 10 SMs (consumer card)
- Volta V100 has 84 SMs (HPC card)



# Hiding memory latency

**Simplified execution model**

- Does not support out-of-order execution
- Does not support speculative execution, branch prediction, etc.

**SM execution model**

- Schedules *warp* of 32 threads that run in lock-step (caveat: Volta)
- Warp loads 32 elements from memory
- While load is pending, context switches to another warp
- Implies we need many warps per SM to hide latency and achieve good performance

# Two version numbers

**CUDA architecture or compute capability is hardware version; code names**

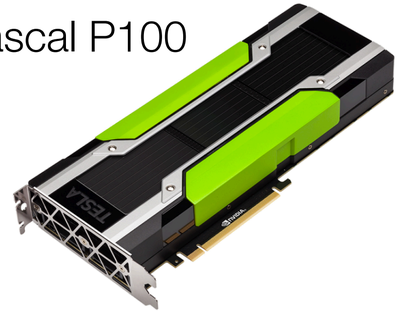
- CUDA arch 1 Tesla*
- CUDA arch 2 Fermi
- CUDA arch 3 Kepler
- CUDA arch 5 Maxwell
- CUDA arch 6 Pascal
- CUDA arch 7 Volta
- CUDA arch 7.5 Turing

*** Confusingly, there's also the Tesla line of HPC GPU cards, spanning all CUDA architectures.**

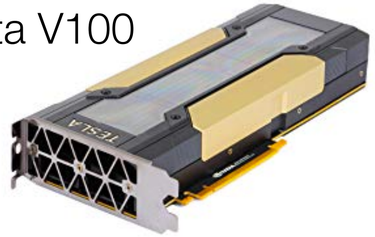
Fermi C870


Kepler K40


Pascal P100




Volta V100




**CUDA Toolkit version is the software version**

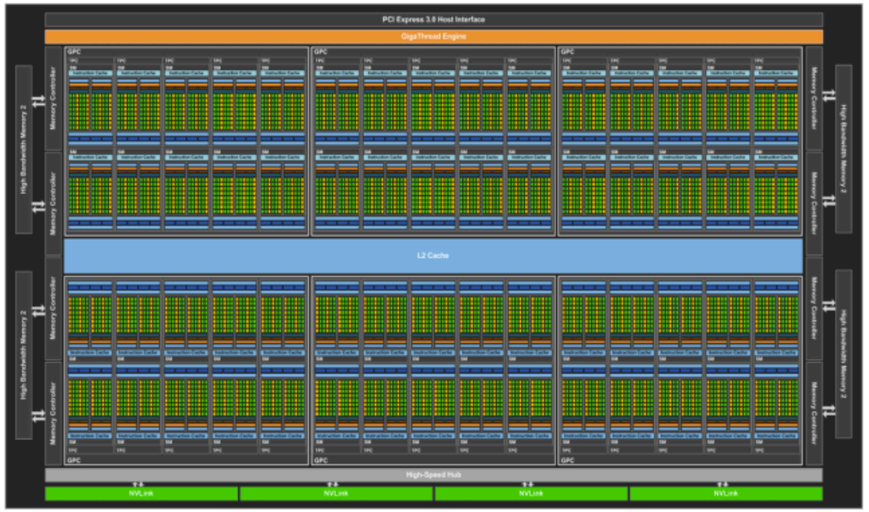
- CUDA 1.x — 2007, supports arch 1.0 – 1.1
- CUDA 2.x — 2008, supports arch 1.0 – 1.3
- CUDA 3.x — 2010, supports arch 1 – 2
- CUDA 4.x — 2011, supports arch 1 – 2
- CUDA 5.x — 2012, supports arch 1 – 2
- CUDA 6.x — 2014, supports arch 1 – 3
- CUDA 7.x — 2015, supports arch 2 – 5
- CUDA 8.x — 2016, supports arch 2 – 6
- CUDA 9.x — 2017, supports arch 3 – 7.2
- CUDA 10.x — 2018, supports arch 3 – 7.5

https://en.wikipedia.org/wiki/CUDA

https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

# Limits: Pascal

**P100 GPU (from NVIDIA whitepaper)**




**CUDA architecture 6.0 (Pascal)**

| | | |
|---|---|---|
| • CUDA cores / SM | 64 | |
| • Blocks / SM | 32 | |
| • Warps (32 threads) / SM | 64 | |
| • Threads / SM | 2048 | |
| • 32-bit Registers / SM | 64 Ki | (i.e., 256 KiB) |
| • 32-bit Registers / thread | 255 | |
| • Shared memory / SM | 64 KiB | |
| • Shared memory / block | 48 KiB | |

**Pascal P100 has 56 SMs (HPC card)**

- Up to 1,792 blocks and 114,688 threads resident, up to 3584 threads executing per cycle
- Kernel launch can (should) be much larger

CUDA C Programming Guide: Appendix H

Pascal Architecture Whitepaper

Pascal P100 Datasheet



Figures from NVIDIA whitepapers

# Limits: Volta

**V100 GPU (from NVIDIA whitepaper)**


## CUDA architecture 7.0 (Volta)

- CUDA cores / SM 64
- Blocks / SM 32
- Warps (32 threads) / SM 64
- Threads / SM 2048
- 32-bit Registers / SM 64 Ki (i.e., 256 KiB)
- 32-bit Registers / thread 255
- Shared memory / SM 96 KiB
- Shared memory / block 96 KiB (using > 48 KiB requires special configuration)

## Volta V100 has 84 SMs (HPC card)

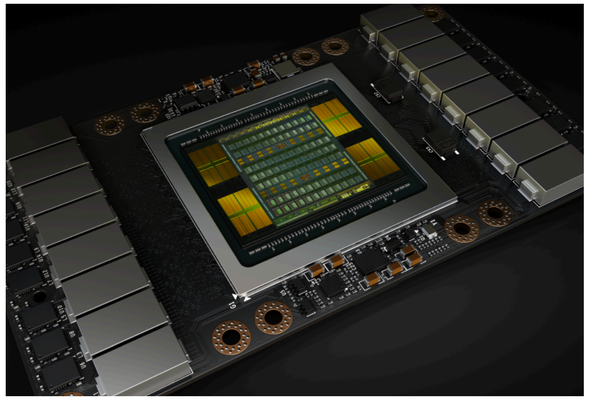
- Up to 2,688 blocks and 172,032 threads resident, up to 5376 threads executing per cycle
- Kernel launch can (should) be much larger

CUDA C Programming Guide: Appendix H

Volta Architecture Whitepaper

Volta V100 Datasheet

# NVIDIA Volta V100




Figures from NVIDIA whitepapers

# Example 0: kernel

Basic CUDA kernel syntax

Compiling CUDA kernel

# CUDA kernel syntax

**__global__ keyword marks GPU kernel**

- Always void
- Executed by each thread

**dim3 blocks, threads**

- 1D, 2D, or 3D grid

**<<< ... >>> triple chevron syntax**

- Launches kernel asynchronously; returns immediately
- Optional shmem: dynamic shared memory, typically 0
- Optional stream: recommended

**cudaGetLastError**

- Launch errors: invalid blocks, threads, etc.

```
// ex00-empty.cu
// compile: nvcc -o ex00-empty ex00-empty.cu

// GPU Kernel: executed by multiple threads on GPU.
__global__
void kernel( ... args ... )
{
    ... code ...
}

// CPU Driver: executed on CPU, launches kernel on GPU.
void driver( ... args ..., cudaStream_t stream )
{
    dim3 blocks( 10, 20, 30 );  // 6000 blocks of
    dim3 threads( 8, 16 );      // 128 threads each.
    kernel<<< blocks, threads, shmem, stream >>>(
        ... args ... );
    cudaError_t err = cudaGetLastError();
    throw_error( err );
}
```

# Compiling CUDA code

**Compile with NVIDIA compiler nvcc**

- `nvcc -O3 -c -o ex00-empty.o ex00-empty.cu` `# create .o object`
- `nvcc -O3 -o ex00-empty ex00-empty.cu` `# create executable`

**GPU kernel code compiled by nvcc itself**

- Supports limited subset of C++
- Doesn't support most of std standard library — use Thrust instead

**For CPU driver code, nvcc invokes host compiler (g++, clang++, ...)**

- Supports full C++
- Can pass flags to host compiler and linker
  - `nvcc -Xcompiler "-fPIC -Wall -Wno-unused-function" -o ex00-empty ex00-empty.cu`

CUDA Compiler Driver NVCC

# Example 1: matrix add

ceildiv for # blocks

Kernel blockIdx, blockDim, threadIdx

Disabling threads

Basic runtime API — malloc, memcpy, free

Thrust — device_vector, copy, raw pointers

Optimize for coalesced memory loads

# Matrix add example

**Add two matrices**

- C = A + B

**Thread blocks overlay matrices**

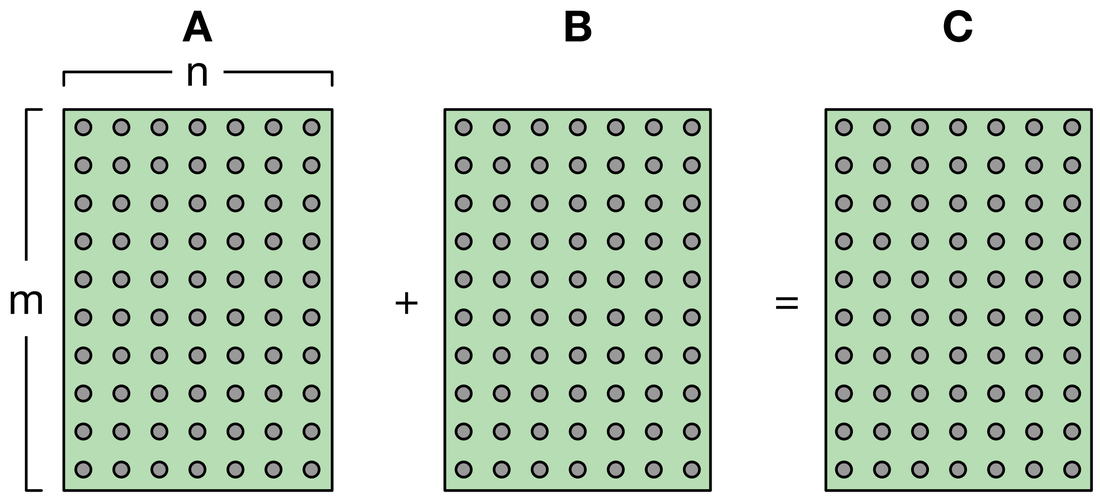
- `ceildiv( m, mb )` = 3 block rows
  `ceildiv( n, nb )` = 2 block cols

**Each thread computes one element of C:**

- Cij = Aij + Bij

**Threads outside matrix are disabled**

**Note 4x4 threads is sub-optimal;
want multiples of 32 threads**



# Matrix add example

**Add two matrices**

- C = A + B

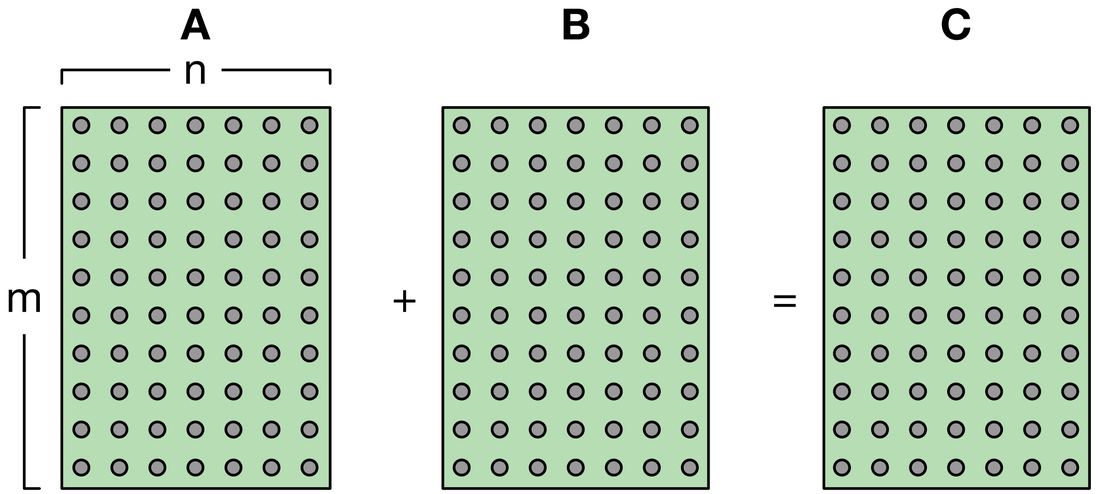
**Thread blocks overlay matrices**

- `ceildiv( m, mb )` = 3 block rows
  `ceildiv( n, nb )` = 2 block cols

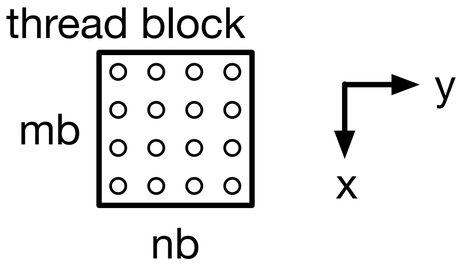
**Each thread computes one element of C:**

- Cij = Aij + Bij

**Threads outside matrix are disabled**




**Note 4x4 threads is sub-optimal;
want multiples of 32 threads**



# Matrix add example

**Add two matrices**

- C = A + B

**Thread blocks overlay matrices**

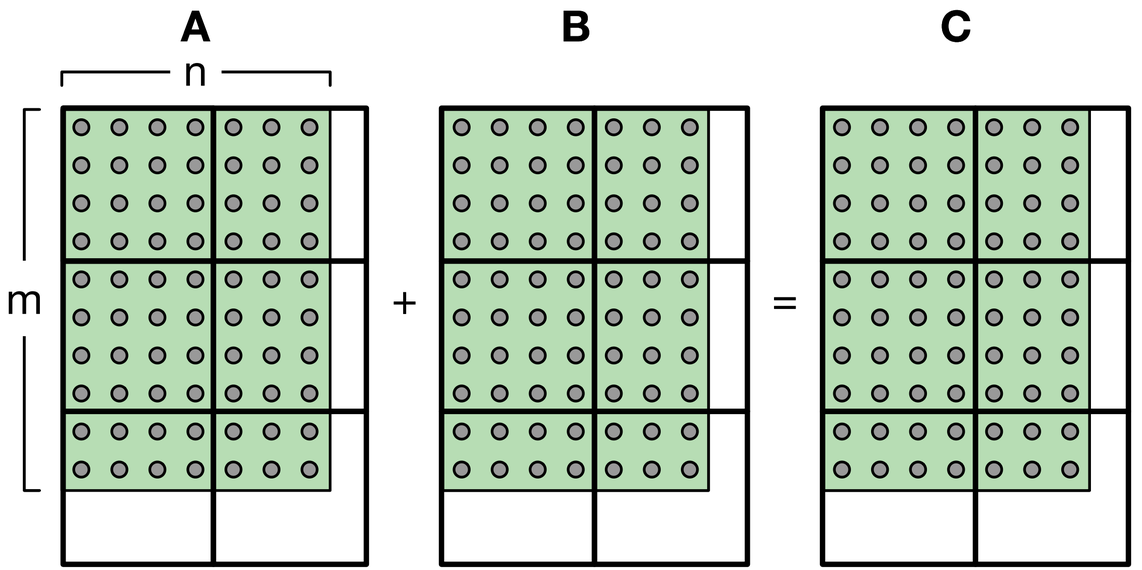
- `ceildiv( m, mb )` = 3 block rows
  `ceildiv( n, nb )` = 2 block cols

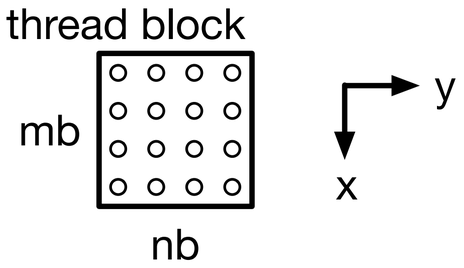
**Each thread computes one element of C:**

- Cij = Aij + Bij

**Threads outside matrix are disabled**

**Note 4x4 threads is sub-optimal;
want multiples of 32 threads**



# Matrix add example

**Add two matrices**

- C = A + B

**Thread blocks overlay matrices**

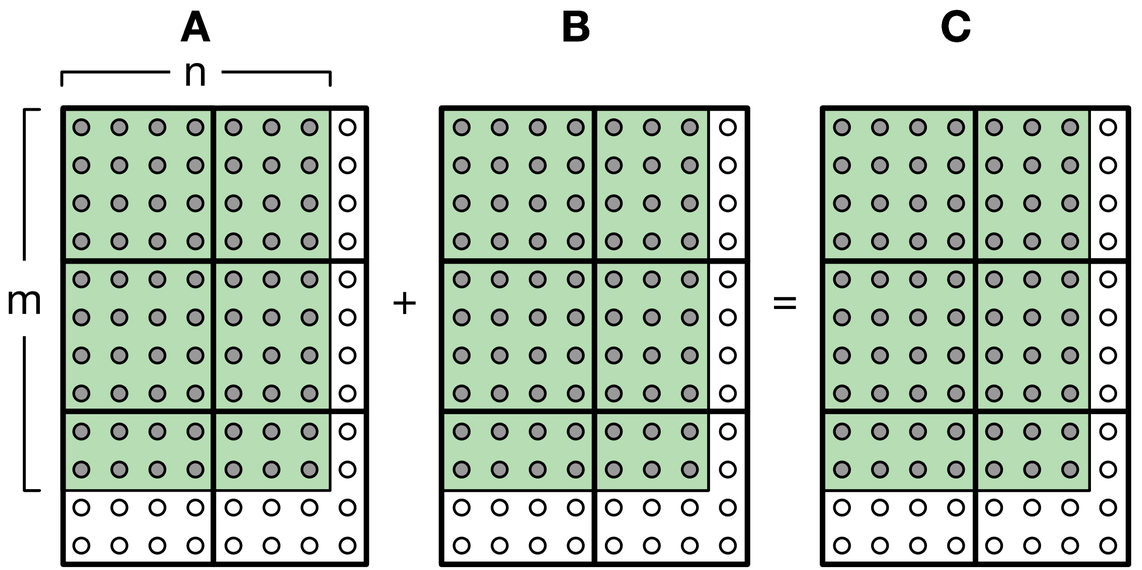
- `ceildiv( m, mb )` = 3 block rows
  `ceildiv( n, nb )` = 2 block cols

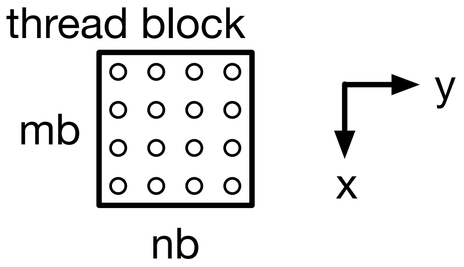
**Each thread computes one element of C:**

- Cij = Aij + Bij

**Threads outside matrix are disabled**

**Note 4x4 threads is sub-optimal;
want multiples of 32 threads**



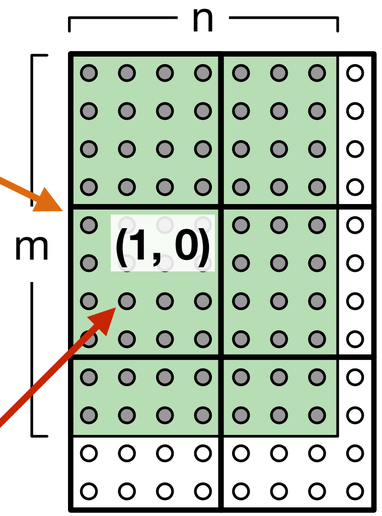
# CUDA: Kernel syntax

Add two matrices:

- C = A + B

Get thread ID, block ID, and block dimensions from predefined variables

- **blockIdx.x, y**
  - index of block, ex: (1, 0)
- **blockDim.x, y**
  - # threads, ex: (4, 4)
- **threadIdx.x, y**
  - index of thread within block, ex: (2, 1)




Threads outside matrix are disabled by if condition

Here, every thread is independent

```cpp
/// ex01-add-matrix.cu
/// GPU Kernel: adds two m-by-n matrices: C = A + B.
/// Each thread adds one element, Cij = Aij + Bij.
template <typename T>
__global__
void matrix_add_kernel(
    int m, int n,
    T const* A, T const* B, T* C, int ld )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < m && j < n) {
        C[ i + j*ld ] = A[ i + j*ld ] + B[ i + j*ld ];
    }
}

/// CPU Driver: adds two m-by-n matrices: C = A + B.
template <typename T>
void matrix_add(
    int m, int n, T const* A, T const* B, T* C, int ld,
    cudaStream_t stream )
{
    dim3 threads( 4, 4 );
    dim3 blocks( ceildiv( m, threads.x ),
                 ceildiv( n, threads.y ) );
    matrix_add_kernel<<< blocks, threads, 0, stream >>>
        ( m, n, A, B, C, ld );
    throw_error( cudaGetLastError() );
}
```

# CUDA: Runtime API

**#include <cuda_runtime.h>**

**cudaMalloc**

- Allocate GPU device memory
- Synchronizes GPU

**cudaMemcpy**

- cudaMemcpyDefault
  Direction determined from pointers
- cudaMemcpyHostToDevice,
  cudaMemcpyDeviceToHost,
  cudaMemcpyDeviceToDevice, etc.

**cudaFree**

- Free GPU memory

**C functions, callable from any .c file (don't need nvcc)**

```
/// ex01-add-matrix.cu, continued
template <typename T>
void test( int m, int n, ... )
{
    // Allocate and initialize matrices on CPU host.
    int ld = m;
    std::vector<T> hA( ld * n ), hB( ld * n ), hC( ld * n );
    rand_matrix( m, n, hA.data(), ld );
    rand_matrix( m, n, hB.data(), ld );

    // Allocate matrices on GPU device and copy from host to device.
    T *dA = nullptr, *dB = nullptr, *dC = nullptr;
    size_t size = ld * n * sizeof(T);
    throw_error( cudaMalloc( &dA, size ) );
    throw_error( cudaMalloc( &dB, size ) );
    throw_error( cudaMalloc( &dC, size ) );

    throw_error( cudaMemcpy( dA, hA.data(), size, cudaMemcpyDefault ));
    throw_error( cudaMemcpy( dB, hB.data(), size, cudaMemcpyDefault ));

    // Add matrices, for now using null stream.
    matrix_add( m, n, dA, dB, dC, ld, nullptr );

    // Copy C = dC (device to host).
    throw_error( cudaMemcpy( hC.data(), dC, size, cudaMemcpyDefault ));

    throw_error( cudaFree( dA ) );
    throw_error( cudaFree( dB ) );
    throw_error( cudaFree( dC ) );
}
```

# Thrust: STD for CUDA

**#include <thrust/device_vector.h>**

**device_vector<T>**

- Similar to std::vector<T>

**Copy from std::vector**

- CPU ⇒ GPU
- Copy to std::vector doesn't work

**Get raw pointer to pass to GPU kernel**

**thrust::copy**

- Similar to std::copy
- CPU ⇔ GPU

```
/// ex01-add-matrix-thrust.cu
template <typename T>
void test( int m, int n, ... )
{
    // Allocate and initialize matrices on CPU host.
    int ld = m;
    std::vector<T> A( ld * n ), B( ld * n ), C( ld * n );
    rand_matrix( m, n, A.data(), ld );
    rand_matrix( m, n, B.data(), ld );

    // Allocate matrices on GPU device and copy from host to device.
    thrust::device_vector<T> dA_vec, dB_vec, dC_vec;
    dA_vec = hA;
    dB_vec = hB;
    dC_vec = hC;

    // Get raw pointers to pass to CUDA kernel.
    T *dA, *dB, *dC;
    dA = thrust::raw_pointer_cast( dA_vec.data() );
    dB = thrust::raw_pointer_cast( dB_vec.data() );
    dC = thrust::raw_pointer_cast( dC_vec.data() );

    // Add matrices, for now using null stream.
    matrix_add( m, n, dA, dB, dC, ld, nullptr );

    // Copy C = dC (device to host).
    // hC = dC_vec;  // doesn't work
    thrust::copy( dC_vec.begin(), dC_vec.end(), hC.begin() );
}
```

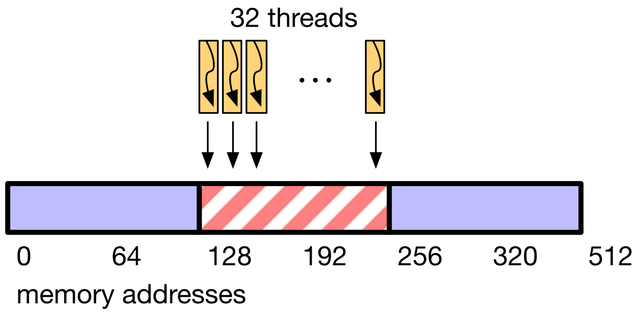
# First optimization: Coalesced memory access

**If threads issue reads or writes within one cache line, generates one coalesced (vector) load**

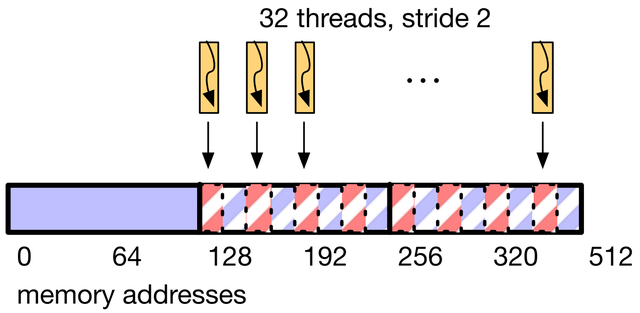
**Similar to optimizing for cache lines on CPUs**

- CUDA L1 cache line is 128 bytes (32 floats or 16 doubles)
- CUDA L2 cache line is 32 bytes (8 floats or 4 doubles)
- Ideally, loads are aligned to 128 byte boundaries

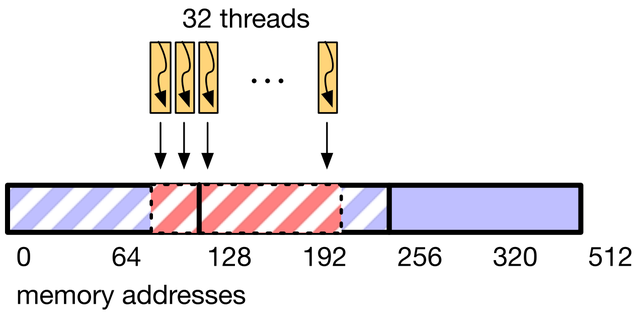
**Aligned 128 byte load reads one cache line**




**Load with stride 2 reads two cache lines**




**Misaligned load reads two cache lines**



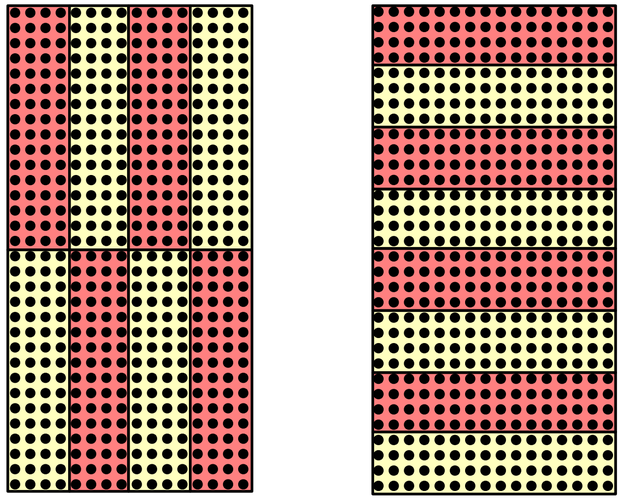

CUDA C Programmer Guide: H.3.2. Global Memory (3.x, Kepler)

CUDA C Best Practices: 9.2.1. Coalesced Access to Global Memory

# First optimization: Coalesced memory access

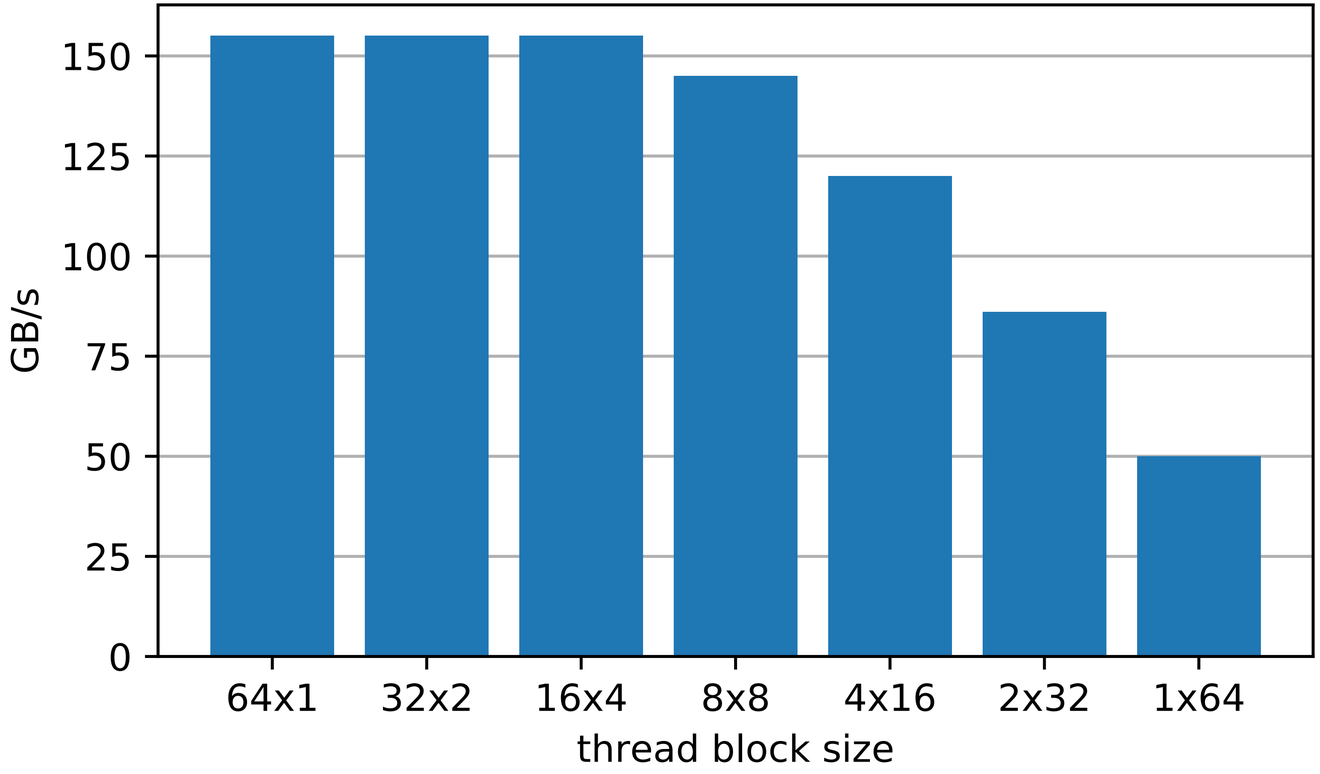
**Tune block dimensions of matrix-add kernel**

- All using 64 threads
- 8000 x 8000 matrix
- Stored column-major



16 x 4
thread block

4 x 16
thread block




single-precision results on GTX 1060
192 GB/s theoretical peak, achieve 80% of peak

# Example 2: norm

Memory hierarchy

Shared memory

__syncthreads

Parallel reduction

Column vs. row access

Thrust — reduction

# CUDA memory hierarchy

## Registers

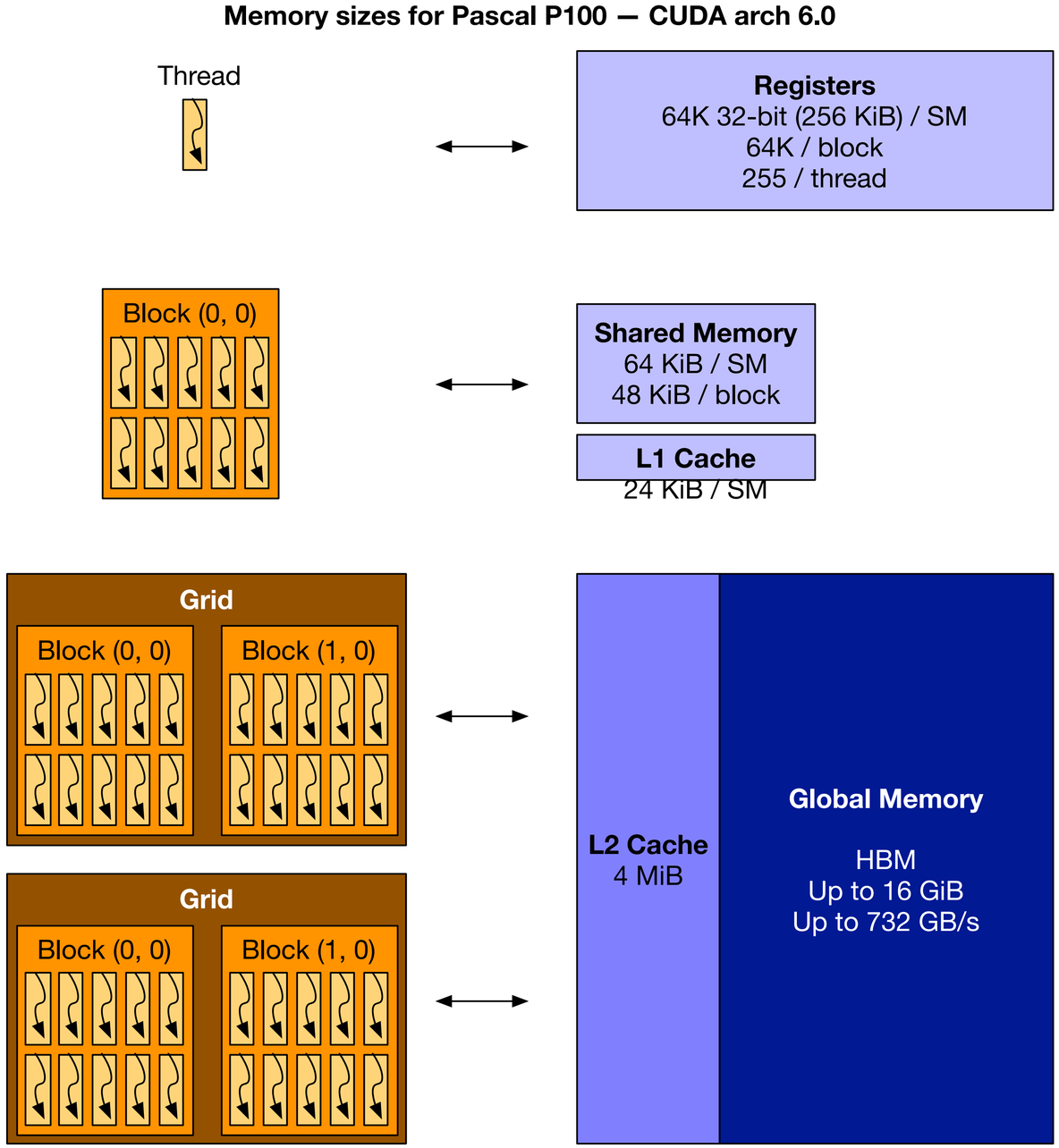
- Variables and arrays
- Local to each thread
  - Shuffle provides means to exchange

## Shared memory

- User-managed L1 cache
- **__shared__** keyword
- Local to each block
- **__syncthreads()** to ensure data consistency

## L1, L2 cache

## Global memory (GDDR, HBM)




CUDA C Programming Guide, Appendix H: Compute Capabilities

# Matrix norm

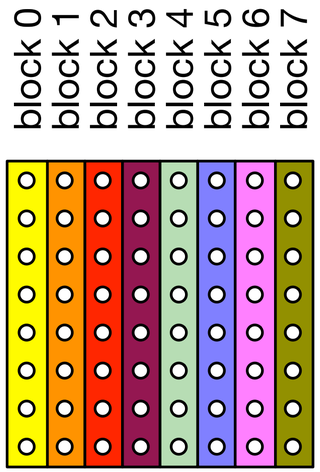
**One norm: maximum column sum**

- Version 1:
  - Assign one column per thread-block
  - Each thread does partial column sum
  - Parallel sum reduction to get total column sum
- Version 2:
  - Assign nb columns per thread-block
  - Each thread does one column

**Inf norm: maximum row sum**

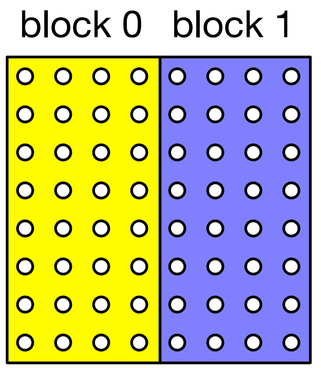
- Same implementations, using rows

**Version 1**

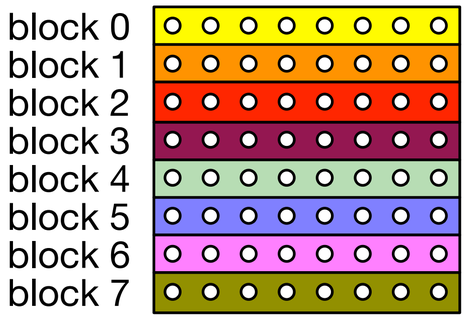



**Version 2**

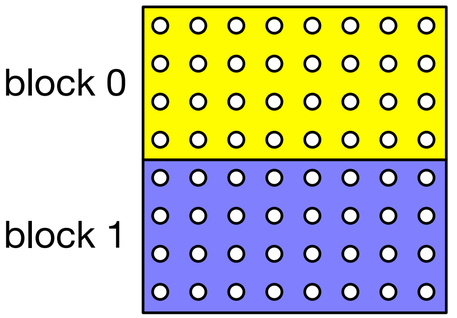
nb = 4 threads
for illustration




**Version 1**

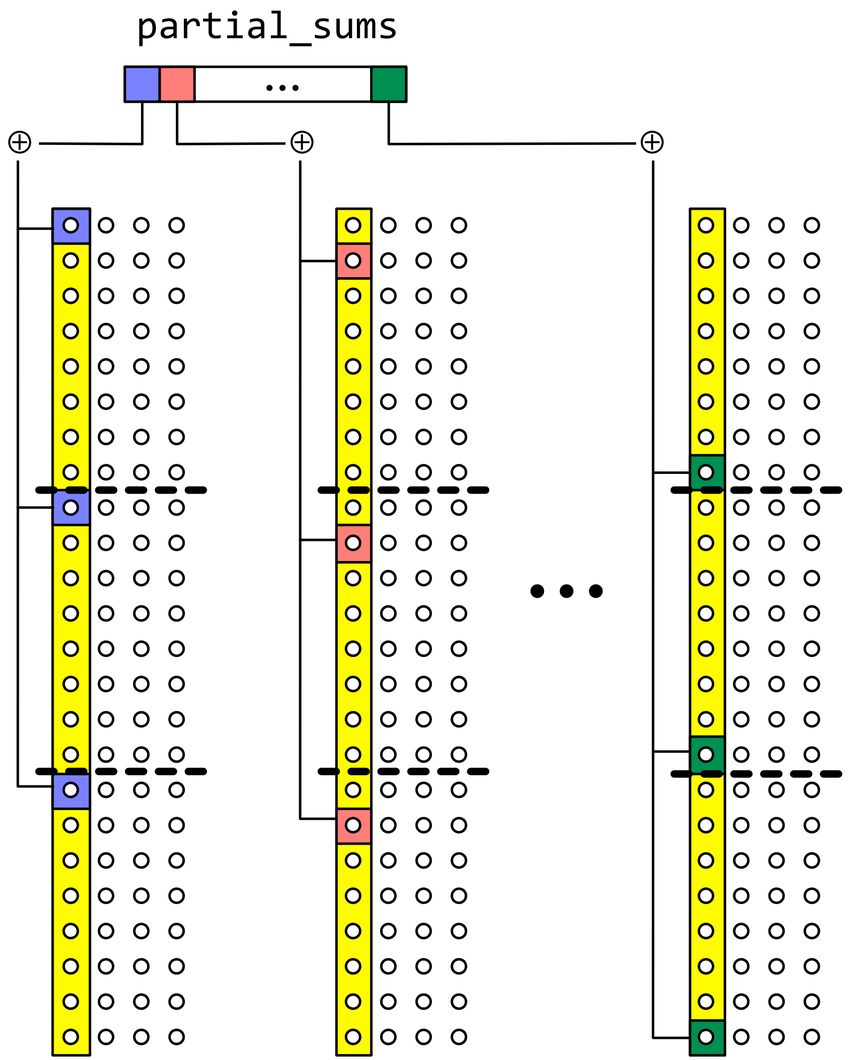



**Version 2**



# Matrix norm: Version 1, part A

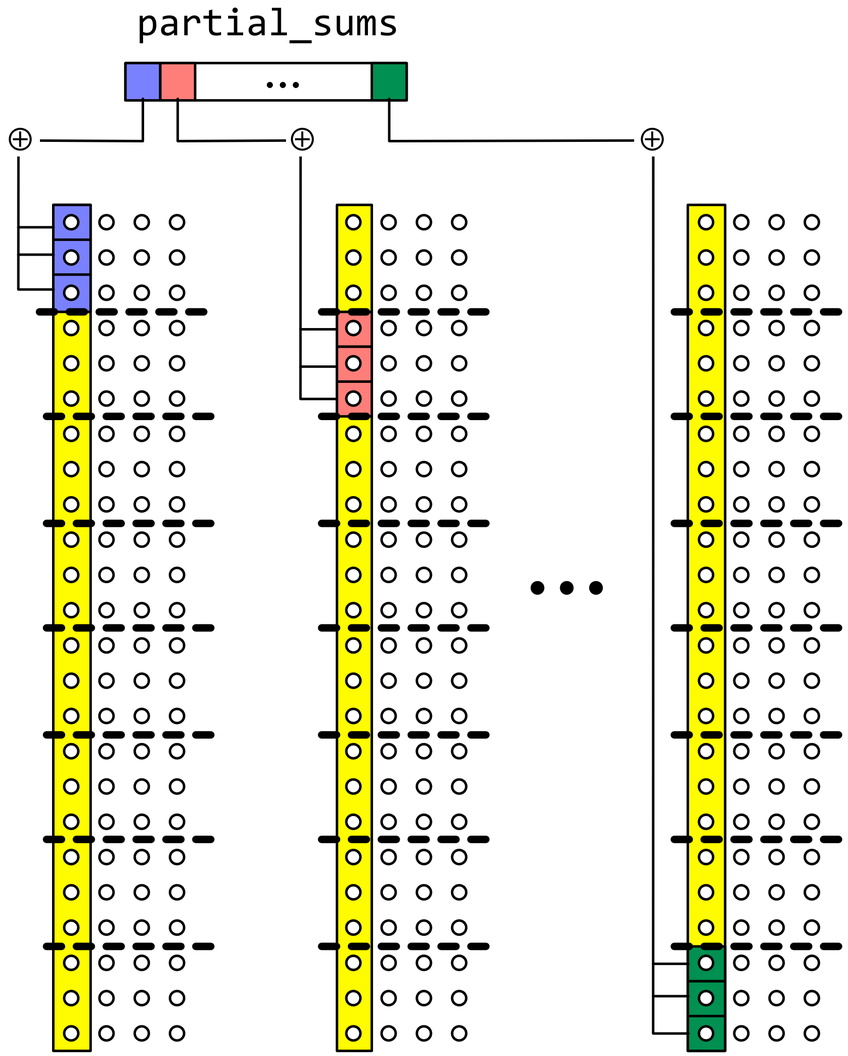
Thread block 0 sums column 0




```
/// GPU kernel: computes column sums:
///     sums[j] = sum_{i = 0...m} | Aij |.
///
/// Each thread-block computes one entry, sums[j].
/// Threads compute partial sums,
/// then sum-reduce to get sums[j].
///
template <typename T>
__global__
void norm_one_kernel_v1(
    int m, int n, T const* A, int ld, T* sums )
{
    // Partial column sums, one per thread in thread block.
    const int nb = norm_one_nb;
    __shared__ T s_partial_sums[ nb ];

    // Shift to j-th column.
    int j = blockIdx.x;
    T const* Aj = &A[ j*ld ];

    // Thread tid sums:
    // s_partial_sums[ tid ] = sum_i A(tid + i*nb, j).
    int tid = threadIdx.x;
    s_partial_sums[ tid ] = 0;
    for (int i = tid; i < m; i += nb) {
        s_partial_sums[ tid ] += abs( Aj[ i ] );
    }

    // continued ...
```

# Matrix norm: Version 1b, part A

## Thread block 0 sums column 0 — alternate ordering



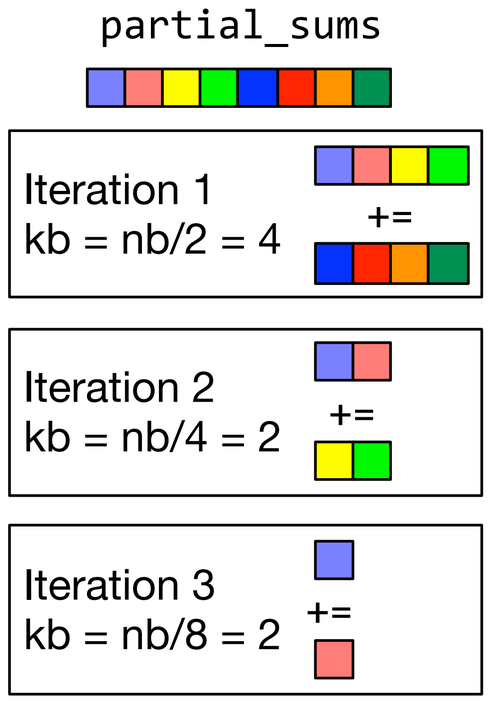

```
/// GPU kernel: computes column sums:
///     sums[j] = sum_{i = 0...m} | Aij |.
///
/// Each thread-block computes one entry, sums[j].
/// Threads compute partial sums,
/// then sum-reduce to get sums[j].
///
template <typename T>
__global__
void norm_one_kernel_v1(
    int m, int n, T const* A, int ld, T* sums )
{
    // Partial column sums, one per thread in thread block.
    const int nb = norm_one_nb;
    __shared__ T s_partial_sums[ nb ];

    // Shift to j-th column.
    int j = blockIdx.x;
    T const* Aj = &A[ j*ld ];

    // Thread tid sums:
    // s_partial_sums[ tid ] = sum_i A(tid + i*nb, j).
    int tid = threadIdx.x;
    s_partial_sums[ tid ] = 0;
    // original:
    // for (int i = tid; i < m; i += nb) {
    int chunk = ceildiv( m, nb );
    int begin = tid*chunk;
    int end = min( (tid+1)*chunk, m );
    for (int i = begin; i < end; ++i) {
        s_partial_sums[ tid ] += abs( Aj[ i ] );
    }

    // continued ...
```

# Matrix norm: Version 1, part B

Binary tree based sum reduction of s_partial_sums

**All threads** call **__syncthreads()** between updates to shared memory s_partial_sums

- Even threads that are not computing must sync, otherwise can deadlock




```
    // ... continued

    // Parallel binary tree sum reduction;
    // result in s_partial_sums[ 0 ].
    int kb = nb / 2;
    while (kb > 0) {
        __syncthreads();
        if (tid < kb)
            s_partial_sums[ tid ] += s_partial_sums[ tid + kb ];
        kb /= 2;
    }

    // Save thread block's result.
    if (tid == 0) {
        sums[ j ] = s_partial_sums[ 0 ];
    }
}
```

# Matrix norm: Version 2

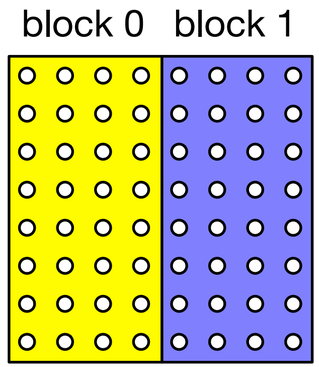
Assign nb columns (rows) per thread-block

Each thread does one column (row)

No shared memory, no synchronization

**Version 2**

nb = 4 threads
for illustration




```
/// GPU kernel: computes col sums:
///     sums[j] = sum_{j = 0...n} | Aij |.
///
/// Each thread-block computes nb column sums,
///     sums[ k, ..., k+nb ], k = blockIdx*nb.
/// Each thread sums one col.
///
template <typename T>
__global__
void norm_one_kernel_v2(
    int m, int n, T const* A, int ld, T* sums )
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j < n) {
        // Shift to j-th col.
        T const* Aj = &A[ j*ld ];

        // Thread sums down col i.
        T sum = 0;
        for (int i = 0; i < m; ++i) {
            sum += abs( Aj[ i ] );
        }

        // Save thread's result.
        sums[ j ] = sum;
    }
}
```

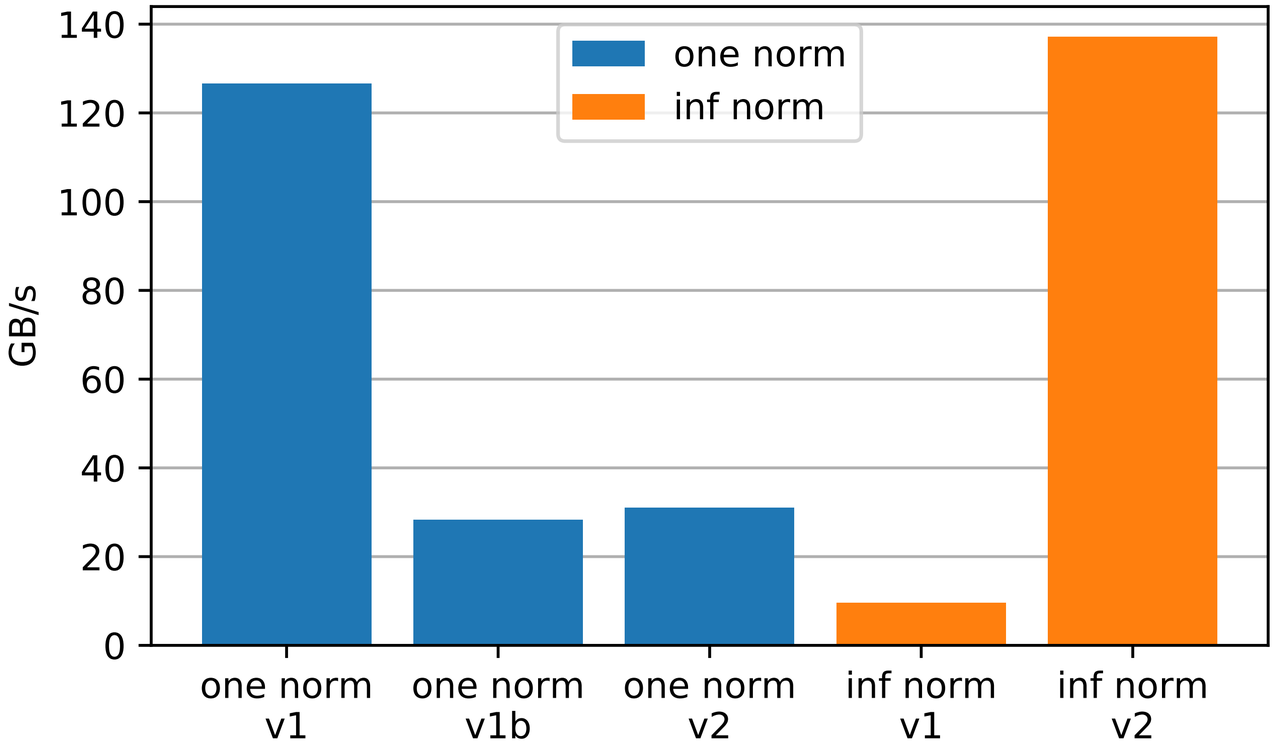
# Matrix norm

**One norm: column sums**

- Version 1
  - Coalesced loads
  - Alternate ordering (1b) suffers from non-coalesced (strided) loads
- Version 2
  - Suffers from non-coalesced loads

**Inf norm: row sums**

- Version 1
  - Suffers from non-coalesced loads
- Version 2
  - Coalesced loads



# Matrix norm: final step

**Find max of column (row) sums**

- Max reduction of n elements

**Could implement similar to column-sum kernel**

**Thrust provides reductions**

- thrust::plus
- thrust::multiplies
- thrust::logical_and
- thrust::maximum
- thrust::minimum

```
/// CPU driver computes one norm of matrix, maximum column sum:
///     max_{j = 0...n} sum_{i = 0...m} | Aij |.
///
/// CPU driver launches kernels on GPU.
///
template <typename T>
T matrix_norm_one(
    int m, int n, T const* A, int ld, cudaStream_t stream,
    int version, int verbose )
{
    // Workspace for n column sums.
    thrust::device_vector<T> sums_vec( n );
    T* sums = thrust::raw_pointer_cast( sums_vec.data() );

    // Compute column sums.
    // m blocks, nb threads each.
    int blocks = m;
    norm_one_kernel_v1<<< blocks, norm_one_nb, 0, stream >>>
        ( m, n, A, ld, sums );
    throw_error( cudaGetLastError() );

    // Get max column sum.
    T result = thrust::reduce(
        sums_vec.begin(), sums_vec.end(), 0,
        thrust::maximum<T>() );
    return result;
}
```

http://thrust.github.io/doc/group__reductions.html

http://thrust.github.io/doc/group__predefined__function__objects.html

# Example 3: matrix multiply

Device and host functions

Loading to shared memory

Kernel strategy

# Matrix Multiply: C = A*B

## Adapted from CUDA C Programmer Guide

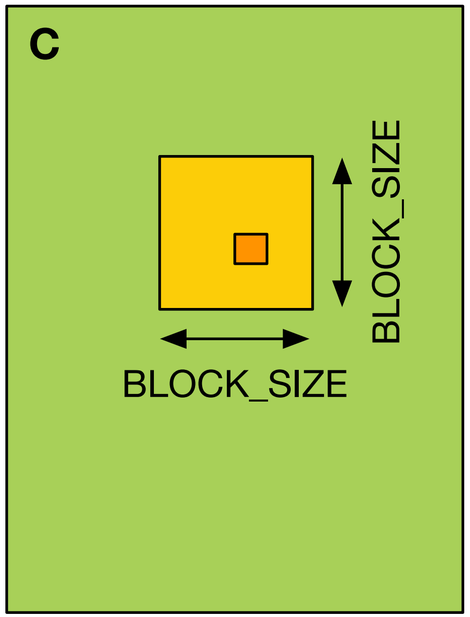
- Modified: column-major, renamed variables
- Assumes dimensions divisible by BLOCK_SIZE

## Device functions

- **__device__** marks GPU device function, callable from GPU kernel
- **__host__** marks CPU function, callable from CPU code
- **__device__ __host__** marks function callable from both CPU code and GPU kernel

```
// ex03-matrix-multiply.cu
// Thread block size (16 x 16)
#define BLOCK_SIZE 16

//---------------------------------------------------------
// Matrices are stored in column-major order:
// M(row, col) = *(M.elements + row + col*M.stride)
//
typedef struct {
    int cols;
    int rows;
    int stride;
    float* elements;
} Matrix;

//---------------------------------------------------------
// Get a matrix element, A( row, col ).
//
__device__
float GetElement( const Matrix A, int row, int col )
{
    return A.elements[row + col*A.stride];
}

//---------------------------------------------------------
// Set a matrix element, A( row, col ) = value.
//
__device__ __host__
void SetElement( Matrix A, int row, int col, float value )
{
    A.elements[row + col*A.stride] = value;
}
```

# Matrix Multiply: C = A*B




Each thread-block computes one BLOCK_SIZE x BLOCK_SIZE block of C

Each thread computes one element of C

```
// Get block sub-matrix A( blockRow, blockCol ).
__device__
Matrix GetSubMatrix( Matrix A, int blockRow, int blockCol )
{
    Matrix Asub;
    Asub.cols     = BLOCK_SIZE;
    Asub.rows     = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[BLOCK_SIZE * blockRow
                    + A.stride * BLOCK_SIZE * blockCol];
    return Asub;
}
```

```
//-----------------------------------------------------------
// Matrix multiplication kernel called by MatMul()
//
__global__
void MatMulKernel( Matrix A, Matrix B, Matrix C )
{
    // Block row and column
    int blockRow = blockIdx.x;
    int blockCol = blockIdx.y;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix( C, blockRow, blockCol );

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.x;
    int col = threadIdx.y;

    // continued ...
```

# Matrix Multiply: C = A*B

**Outer *k* loop over block row of A / block col B**

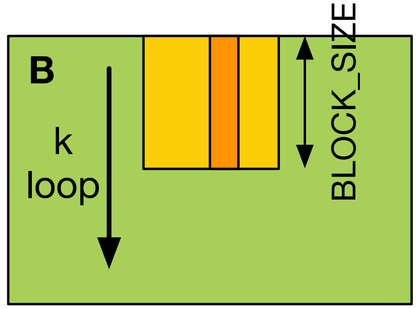
- Load block of A into sA
- Load block of B into sB

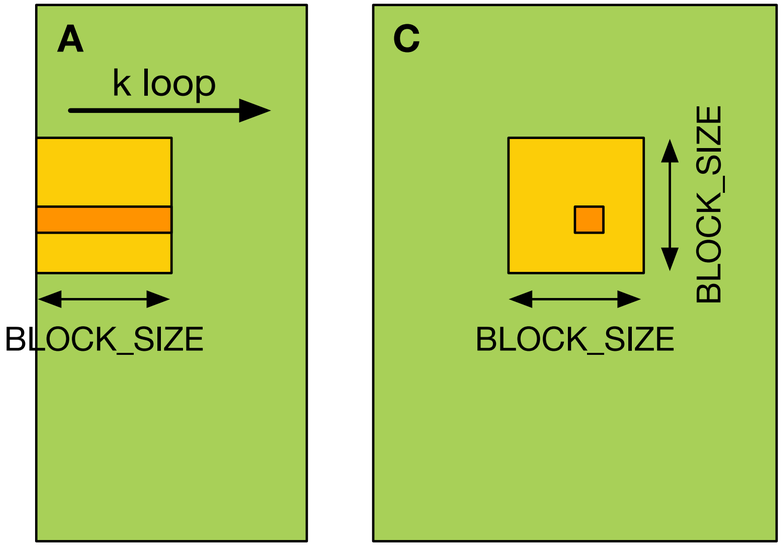
**Inner *e* loop**

- Each thread multiplies row of sA and col of sB

**Reuse**

- Elements in sA and sB reused BLOCK_SIZE times




```
    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int k = 0; k < (A.cols / BLOCK_SIZE); ++k) {
        // Get sub-matrix Asub of A and Bsub of B.
        Matrix Asub = GetSubMatrix( A, blockRow, k );
        Matrix Bsub = GetSubMatrix( B, k, blockCol );

        // Shared memory used to store Asub and Bsub
        __shared__ float sA[ BLOCK_SIZE ][ BLOCK_SIZE ];
        __shared__ float sB[ BLOCK_SIZE ][ BLOCK_SIZE ];

        // Load Asub and Bsub from memory to shared memory
        // Each thread loads one element of each sub-matrix
        sA[col][row] = GetElement( Asub, row, col );
        sB[col][row] = GetElement( Bsub, row, col );

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

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += sA[e][row] * sB[col][e];

        // Synchronize to ensure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
```

# Matrix Multiply: C = A*B

**Outer *k* loop over block row of A / block col B**

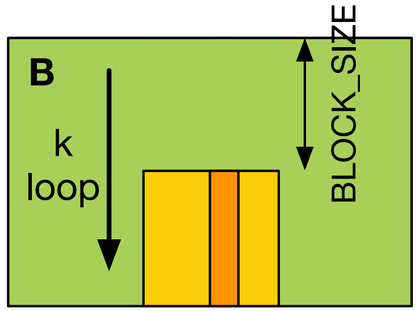
- Load block of A into sA
- Load block of B into sB

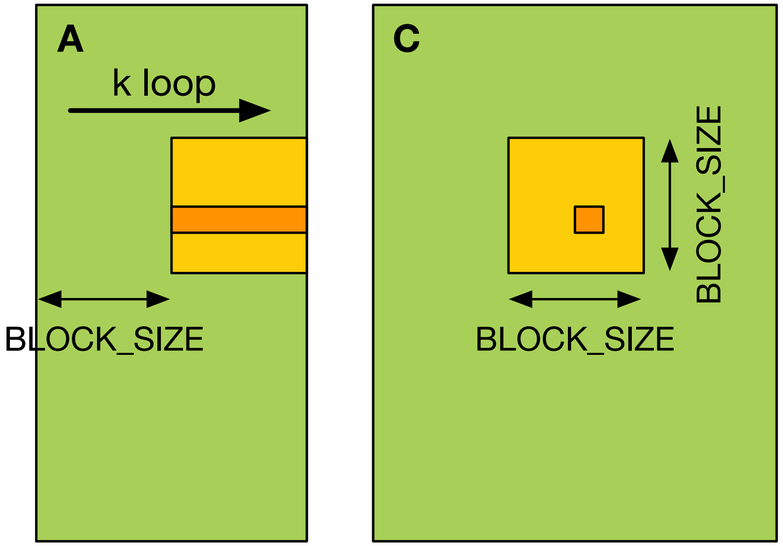
**Inner *e* loop**

- Each thread multiplies row of sA and col of sB

**Reuse**

- Elements in sA and sB reused BLOCK_SIZE times




```
    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int k = 0; k < (A.cols / BLOCK_SIZE); ++k) {
        // Get sub-matrix Asub of A and Bsub of B.
        Matrix Asub = GetSubMatrix( A, blockRow, k );
        Matrix Bsub = GetSubMatrix( B, k, blockCol );

        // Shared memory used to store Asub and Bsub
        __shared__ float sA[ BLOCK_SIZE ][ BLOCK_SIZE ];
        __shared__ float sB[ BLOCK_SIZE ][ BLOCK_SIZE ];

        // Load Asub and Bsub from memory to shared memory
        // Each thread loads one element of each sub-matrix
        sA[col][row] = GetElement( Asub, row, col );
        sB[col][row] = GetElement( Bsub, row, col );

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

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += sA[e][row] * sB[col][e];

        // Synchronize to ensure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
```

# Matrix Multiply: C = A*B

Save results to global memory

```
    // ... continued

    // Write Csub to device memory
    // Each thread writes one element
    SetElement( Csub, row, col, Cvalue );
}
// end MatMulKernel
```

Kernel assumes dimensions are divisible by BLOCK_SIZE

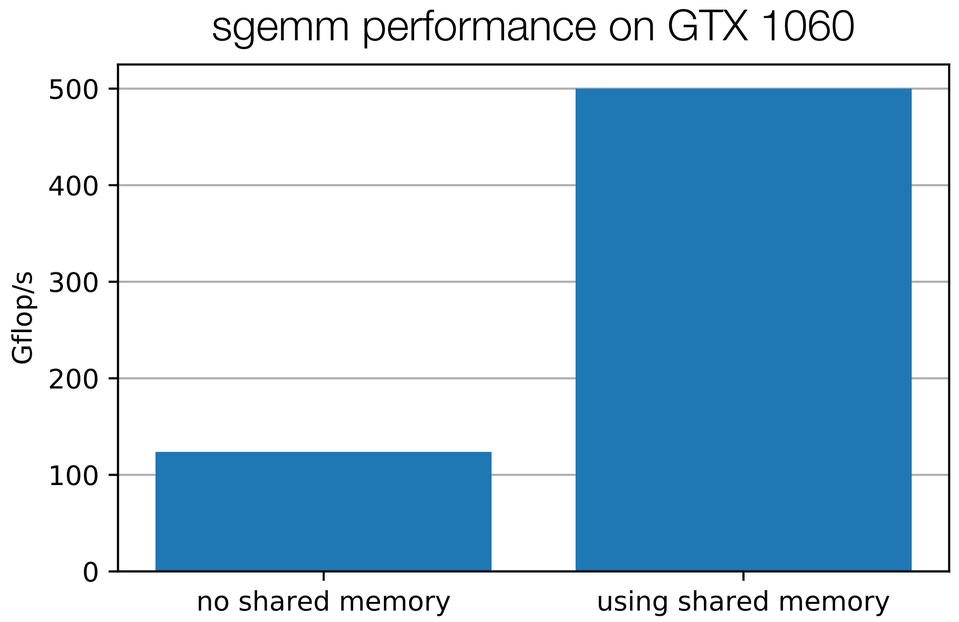
```
//--------------------------------------------------------------
// Matrix multiplication - Host code
// Matrix dimensions assumed to be multiples of BLOCK_SIZE!
//
void MatMul(
    const Matrix d_A, const Matrix d_B, Matrix d_C )
{
    // Check matrix dimensions
    assert( d_A.rows == d_C.rows );
    assert( d_A.cols == d_B.rows );
    assert( d_B.cols == d_C.cols );

    // Invoke kernel
    dim3 threads( BLOCK_SIZE, BLOCK_SIZE );
    dim3 blocks( d_C.cols / dimBlock.x,
                 d_C.rows / dimBlock.y );
    MatMulKernel<<< blocks, threads >>>( d_A, d_B, d_C );
    throw_error( cudaGetLastError() );
}
```

# Kernel strategy

**while (not done)**

- Load data from global device memory into shared memory and registers
- Synchronize threads
- Compute on data in shared memory and registers
- Synchronize threads

**save results**



# Optimization

Beware of thread divergence

Avoid shared memory bank conflicts

Compiling for specific architectures

Profiling and tracing code

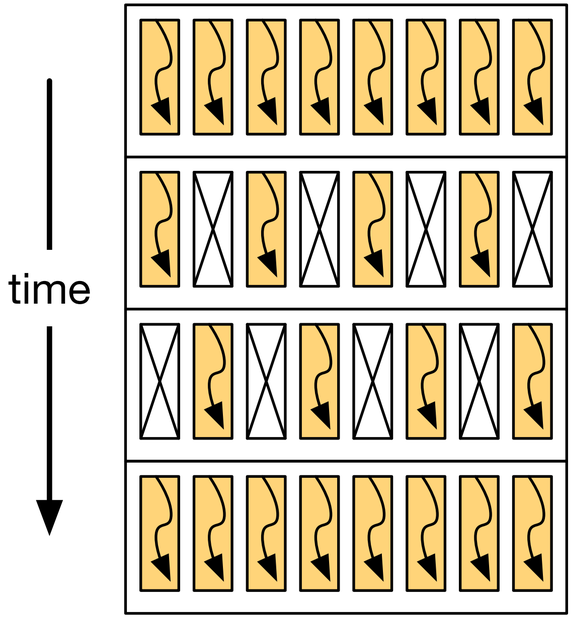
# Thread divergence

CUDA C Best Practices,12.1
Branching and Divergence

**Thread divergence happens when not all threads take the same branch**

- Executes both sides of branch, disabling threads that took other branch
- If all threads in warp take same branch, there's no divergence

**⇒ align divergence with warps**




```
A;

if (tid % 2 == 0) {
    B;
}
else {
    C;
}

D;
```

# Thread divergence

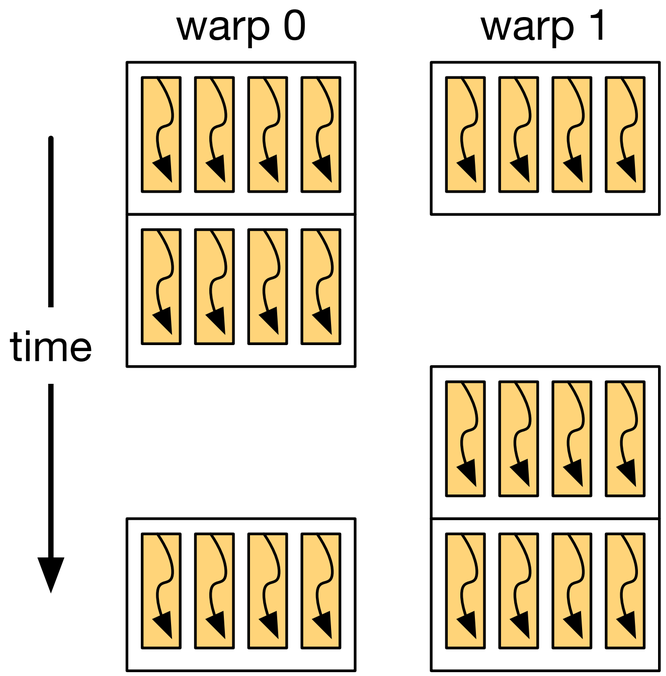
CUDA C Best Practices,12.1
Branching and Divergence

**Thread divergence happens when not all threads take the same branch**

- Executes both sides of branch, disabling threads that took other branch
- If all threads in warp take same branch, there's no divergence

**⇒ align divergence with warps**

For illustration, let warpsize = 4




```
A;

if ((tid/warpsize) % 2 == 0) {
    B;
}
else {
    C;
}

D;
```

# Shared memory bank conflicts

CUDA C Best Practices, 9.2.2.3
Shared Memory in Matrix Multiplication (C=AAT)

**Shared memory spread over 32 banks**

- If two threads access different data in the same bank, causes conflict

**Strided access to shared memory can cause bank conflicts**

- For illustration, assume 4 banks and 4 threads in warp
- Accessing row, all threads hit different banks (sA is row-major) — no conflicts!
- Accessing column, all threads hit same bank — **conflict!**
- Padding row by one eliminates conflicts

```
// row-major
__shared__ float sA[ 4 ][ 4 ];
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |

```
// row +1 padding
__shared__ float sA[ 4 ][ 4+1 ];
```

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 0 |
| 1 | 2 | 3 | 0 | 1 |
| 2 | 3 | 0 | 1 | 2 |
| 3 | 0 | 1 | 2 | 3 |

# Compiling for specific GPU architectures

CUDA Compiler Driver NVCC, 4.2.7
Options for Steering GPU Code Generation

**nvcc can generate:**

- Binary code for specific architecture (**code=sm_XY**)
- PTX assembly code that is forward compatible (**code=compute_XY**)

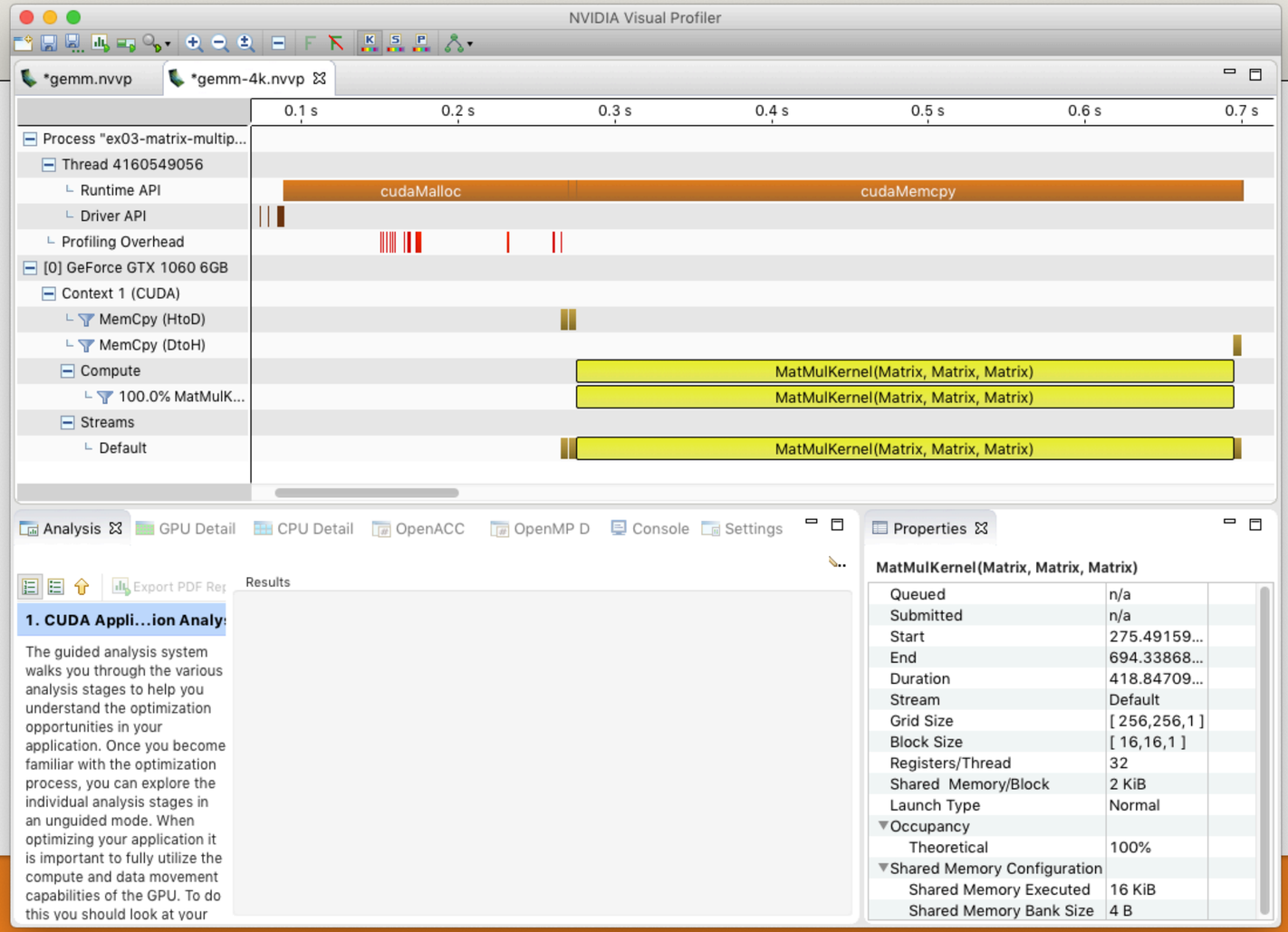
**Compiling for multiple architectures noticeably increases compile times and binary sizes.**

**Strategy is to compile binary for all desired architectures, plus PTX for only highest architecture**

```
nvcc -gencode arch=compute_30,code=sm_30          # binary for arch 3.x only
     -gencode arch=compute_50,code=sm_50          # binary for arch 5.x only
     -gencode arch=compute_50,code=compute_50     # PTX for arch ≥ 5.0
     -c -o foo.o foo.cu
```

# Profiling & tracing

```
>> nvprof -o gemm-4k.nvvp ./ex03-matrix-multiply -m 4096 -n 4096 -k 2048
m 4096, n 4096, k 2048
==43166== NVPROF is profiling process 43166, command: ./ex03-matrix-multiply -m 4096 -n 4096 -k 2048
==43166== Generated result file: /home/mgates/sc19/gemm-4k.nvvp

>> nvvp &
```



# Example 4: hybrid computing

Streams

nvToolsExt tracing

# Streams

**Streams allow for concurrent execution**

- Kernels & communication on one stream execute serially
- Kernels & communication on different streams may execute in parallel
- Async communication requires **pinned (page-locked) host memory**

**nvToolsExt tracing**

- Allows tracing CPU activity with nvprof

```
// ex04-streams.cu
#include <nvToolsExt.h>

//------------------------------------------
template <typename T>
void test( int m, int n )
{
    size_t size = m * n * sizeof(T);
    T *hA, *hB, *dA, *dB, *dworkA, *dworkB;
    cudaMallocHost( &hA, size );  // pinned
    cudaMallocHost( &hB, size );  // pinned
    ...

    cudaStream_t comm_stream;
    cudaStream_t compute_stream;
    cudaStreamCreate( &comm_stream );
    cudaStreamCreate( &compute_stream );

    // Fill in A.
    nvtxRangePush( "init_matrix( A )" );
    init_matrix( m, n, hA, m );
    nvtxRangePop();
```

# Streams

**While computing with dA on GPU**

- Copy hB ⇒ dB
- Compute with hA on CPU

```
// Copy hA => dA to device.
cudaMemcpyAsync( dA, hA, size, cudaMemcpyDefault,
                 comm_stream );

// Meanwhile, fill in B.
nvtxRangePush( "init_matrix( B )" );
init_matrix( m, n, hB, m );
nvtxRangePop();

// Wait for dA copy, then compute on it.
cudaStreamSynchronize( comm_stream );
gpu_norm_one( m, n, dA, m, dworkA, compute_stream );

// Meanwhile, copy B => dB to device.
cudaMemcpyAsync( dB, hB, size, cudaMemcpyDefault,
                 comm_stream );

// Meanwhile, do some CPU computation.
nvtxRangePush( "cpu_norm_one( A )" );
T normA = cpu_norm_one( m, n, hA, m );
nvtxRangePop();
```

# Streams

**While computing with dB on GPU**

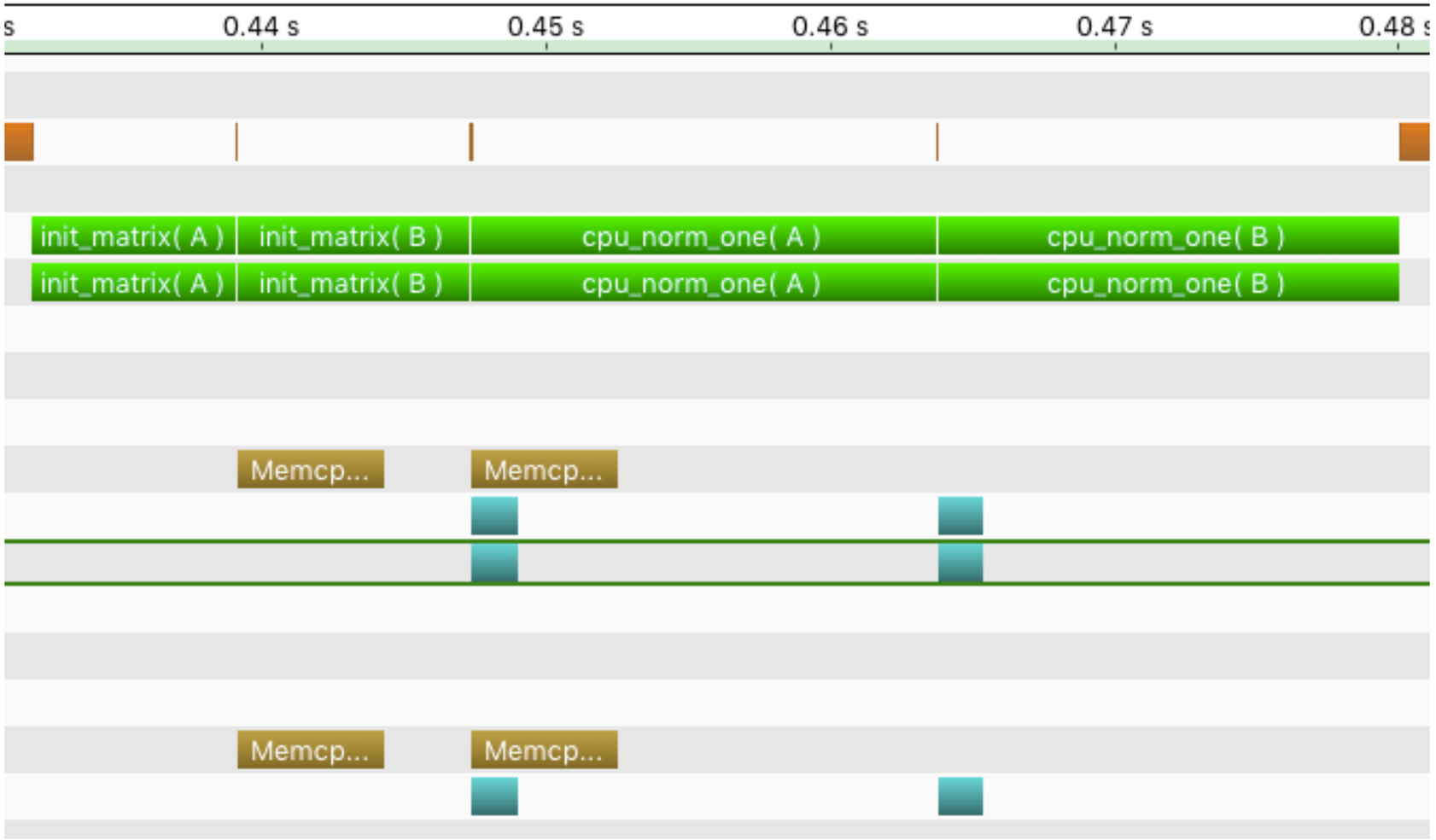
- Compute with hA on CPU

**Wait for GPU computation to finish**

```
    // Wait for dB copy, then compute on it (after A).
    cudaStreamSynchronize( comm_stream );
    gpu_norm_one( m, n, dB, m, dworkB, compute_stream );

    // Meanwhile, do some CPU computation.
    nvtxRangePush( "cpu_norm_one( B )" );
    T normB = cpu_norm_one( m, n, hB, m );
    nvtxRangePop();

    // Wait for computation on dB.
    cudaStreamSynchronize( compute_stream );
    cudaStreamDestroy( comm_stream );
    cudaStreamDestroy( compute_stream );
    cudaFreeHost( hA );
    cudaFreeHost( hB );
    ...
}
```

# Trace

Shows concurrent CPU computing, GPU computing, CPU ⇒ GPU communication



# Runtime API

Devices

Streams

Events

Memory allocation

Memory management

# Device management

**cudaGetDeviceCount( int* count )**

- Count of available GPUs

**cudaSetDevice( int dev )**

**cudaGetDevice( int* dev )**

- Thread-local GPU device ID

**cudaGetDeviceProperties**

**cudaDeviceGetAttr**

- Get all or one device properties
- ex04-properties.cu
  - CUDA arch is **major.minor**

CUDA Runtime API

```
>> ./ex05-properties
device count 1
-------------------------------------------------------------
device                                  0
name                                    Tesla V100-PCIE-16GB
totalGlobalMem                          15.75 GiB (16914055168)
sharedMemPerBlock                       48 KiB (49152)
regsPerBlock                            65536
warpSize                                32
memPitch                                2147483647
maxThreadsPerBlock                      1024
maxThreadsDim                           1024, 1024, 64
maxGridSize                             2147483647, 65535, 65535
clockRate                               1380 MHz (1380000 KHz)
totalConstMem                           64 KiB (65536)
major                                   7
minor                                   0
textureAlignment                        512
texturePitchAlignment                   32
deviceOverlap                           1
multiProcessorCount                     80
kernelExecTimeoutEnabled                0
integrated                              0
canMapHostMemory                        1
...
```

# Stream management

**cudaStreamCreate**

**cudaStreamDestroy**

- Creates and destroys stream

**cudaStreamSynchronize**

- Blocks CPU until all kernels in stream finish

**cudaStreamWaitEvent**

- Blocks stream (but not CPU) until event executes
- Synchronization between two different streams

**Default “null” stream has extra implicit synchronization**

- Recommend always using a non-null stream

CUDA Runtime API

# Event management

**cudaEventCreate**

**cudaEventDestroy**

- Creates and destroys event

**cudaEventRecord**

- Insert event into stream

**cudaEventSynchronize**

- Blocks CPU until event executes

**cudaStreamWaitEvent**

- Blocks stream (but not CPU) until event executes

CUDA Runtime API

# Memory allocation

**cudaMalloc** **cudaMalloc2D** **cudaMalloc3D** **cudaFree**

- Allocate and frees 1D, 2D, or 3D region
- 2D and 3D regions have pitches (aka strides, leading dimensions) with padding for better access

**cudaMallocHost** **cudaFreeHost**

- Allocates and frees pinned, page-locked CPU memory
- Pinned memory is accessible by device (DMA engine), so transfers are much faster, and can be async

**cudaHostRegister** **cudaHostUnregister**

- Pins and unpins existing CPU memory to be page-locked

**GPU memory allocation is synchronous — it blocks GPUs**

CUDA Runtime API

# Memory management

| | | |
|---|---|---|
| **cudaMemcpy** | **cudaMemcpy2D** | **cudaMemcpy3D** |
| **cudaMemcpyAsync** | **cudaMemcpy2DAsync** | **cudaMemcpy3DAsync** |
| **cublasGetMatrix** | **cublasSetMatrix** | |
| **cublasGetMatrixAsync** | **cublasSetMatrixAsync** | |

- Copy 1D, 2D, or 3D region
- Async versions run on stream; requires pinned CPU memory to be asynchronous

| | | |
|---|---|---|
| **cudaMemset** | **cudaMemset2D** | **cudaMemset3D** |
| **cudaMemsetAsync** | **cudaMemset2DAsync** | **cudaMemset3DAsync** |

- Set 1D, 2D, or 3D region to constant byte value, e.g., 0
- Async versions run on stream

CUDA Runtime API


THE UNIVERSITY OF
TENNESSEE
KNOXVILLE