

# GPU Computing with CUDA (and beyond)

## Part 1: a (gentle) introduction to CUDA

Johannes Langguth  
Simula Research Laboratory

# GPUs for Scientific Computing



What, you want me to use a toy for scientific computing ?

Galen Gisler, Geilo Winter School, 2008

# The Price of a Teraflop



1997 ASCI Red: US\$ 73M

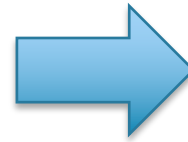


Simulates explosions

# The Price of a Teraflop



1997 ASCI Red: US\$ 73M



2019 GTX 1650: US\$ 149

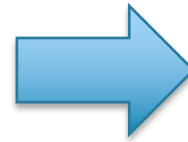


Simulates explosions

# The Price of a Teraflop



1997 ASCI Red: US\$ 73M



2019 GTX 1650: US\$ 149



Simulates explosions



Simulates explosions

# A short history of CUDA / GPGPU

## B.C. (before CUDA)

- 1999 NVIDIA launches the first GeForce gaming cards
- 2001 GeForce 3 introduces programable shaders
- 2006 All GeForce 8 GPUs are CUDA – compatible

## 2007 CUDA 1.0 launched

- 2007 First **Tesla** cards for scientific computing
- 2010 **Tianhe-1A** with **Fermi** GPU becomes fastest supercomputer
- 2012 **Kepler** introduces more cache, dynamic parallelism
- 2015 GPUs for deep learning become big business
- 2016 **Pascal** architecture more than 3x faster than **Kepler**
- 2018 **Summit** with **Volta** becomes the fastest supercomputer

# CUDA: Compute Unified Device Architecture

What is CUDA ?

- a) Parallel computing platform for NVIDIA GPUs
- b) Language extension for C, C++, and Fortran
- c) Software ecosystem
- d) All of the above

What does CUDA stand for ?

**CUDA** = Compute Unified Device Architecture  
(not commonly spelled out anymore)

# CUDA: Compute Unified Device Architecture

## Why bother with CUDA ?

- Highly mature software
- All NVIDIA GPUs support CUDA
- The majority of GPU applications is written in CUDA
- CUDA allows low-level performance programming with reasonable productivity
- **Cheap teraflops**



# CUDA: Programming Basics

GPU `__global__ void mykernel(void) {` ← Kernel  
`}`

---

CPU `int main(void) {`  
`mykernel<<<1,1>>>();` ← Kernel launch  
`return 0;`  
`}`

**CUDA functions are called kernels**

# CUDA: Programming Basics



Host



Device

```
int main(void) {  
    mykernel<<<1,1>>>();  
    return 0;  
}
```

Kernel launch



```
__global__ void mykernel(void) {  
}
```

Kernel

**GPU becomes active when called upon by the CPU**

# Compiling CUDA Programs

```
__global__ void mykernel(void) {  
}  
int main(void) {  
    mykernel<<<1,1>>>();  
    return 0;  
}
```

```
nvcc test.cu
```

**Cuda programs are compiled with nvcc**

# A more interesting CUDA Program

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

We need to allocate space for a, b, and c on the GPU

# Moving data between Device and Host

```
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);  
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);  
  
add<<<1,1>>>(d_a, d_b, d_c);  
  
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
```

Use `cudaMemcpy()` to move data

# Vector addition in CUDA

```
for(int i=0; i<n; i++)  
    c[i]=a[i]+b[i];
```

← for loop:  
CPU programming

```
add<<<n, 1>>>(d_a, d_b, d_c);
```

blocks                      threads per block

← kernel launch:  
CUDA programming

Kernel is launched on `blocks * threads` threads

# Parallel Vector addition in CUDA

```
for (int i=0; i<n; i++)  
    c[i]=a[i]+b[i];
```

C

```
add<<<n,1>>>(d_a, d_b, d_c);
```

CUDA

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

Use `blockIdx.x` to index variables in different blocks

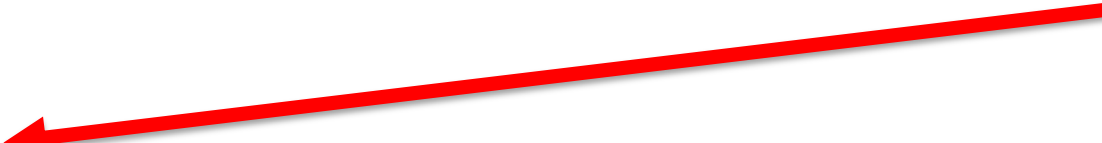
# Parallel Vector addition in CUDA

Blocks

```
add<<<n,1>>>(d_a, d_b, d_c);  
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

Threads

```
add<<<1,n>>>(d_a, d_b, d_c);  
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

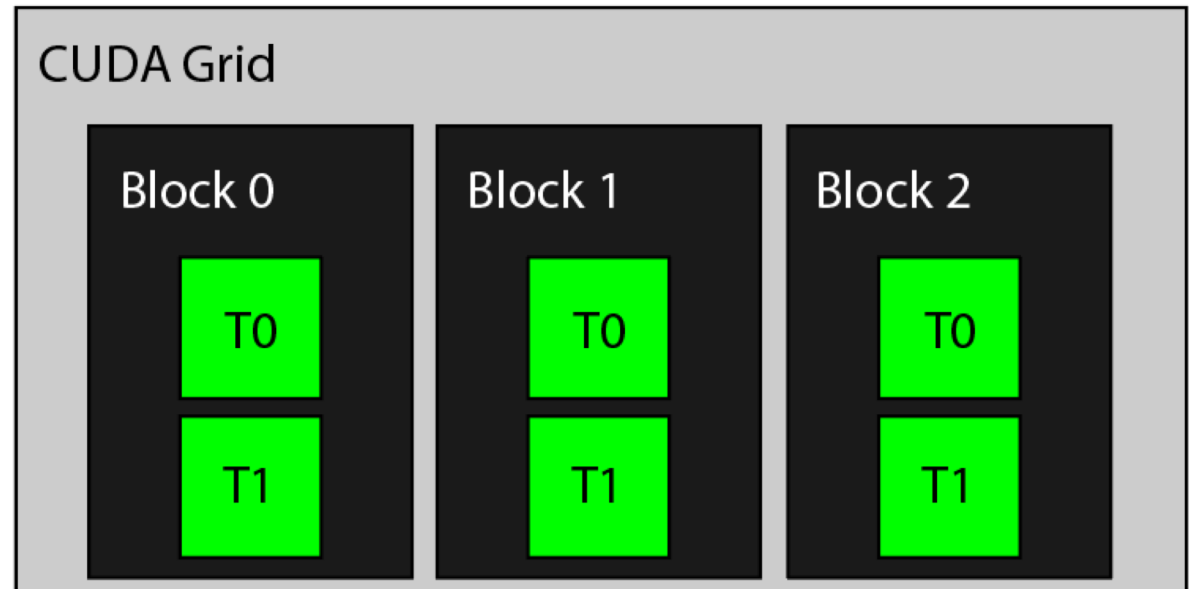
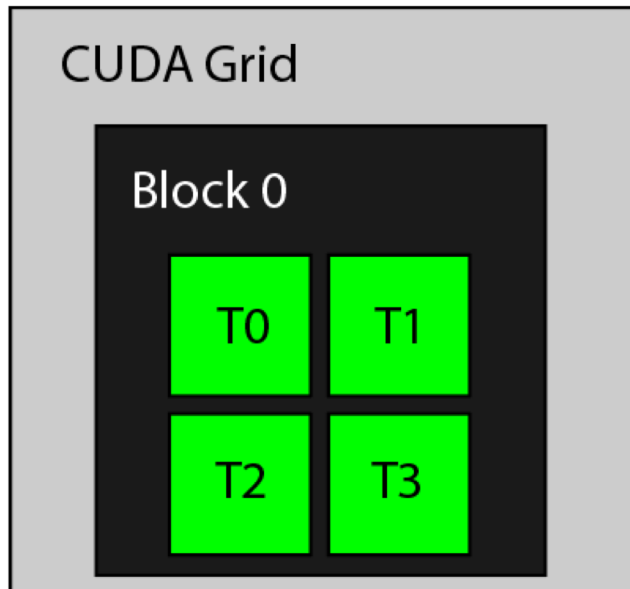


Use `threadIdx.x` to index variables in different threads within a block



# Threads vs Blocks

Why differentiate between threads and blocks ?



- Threads are located on a single multiprocessor
- Threads share fast memory
- Threads are executed together
- Threads per block are limited to 1024

# Parallel Vector addition with Threads and Blocks

```
add<<<ceil(n/128),128>>>(d_a, d_b, d_c);
```

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

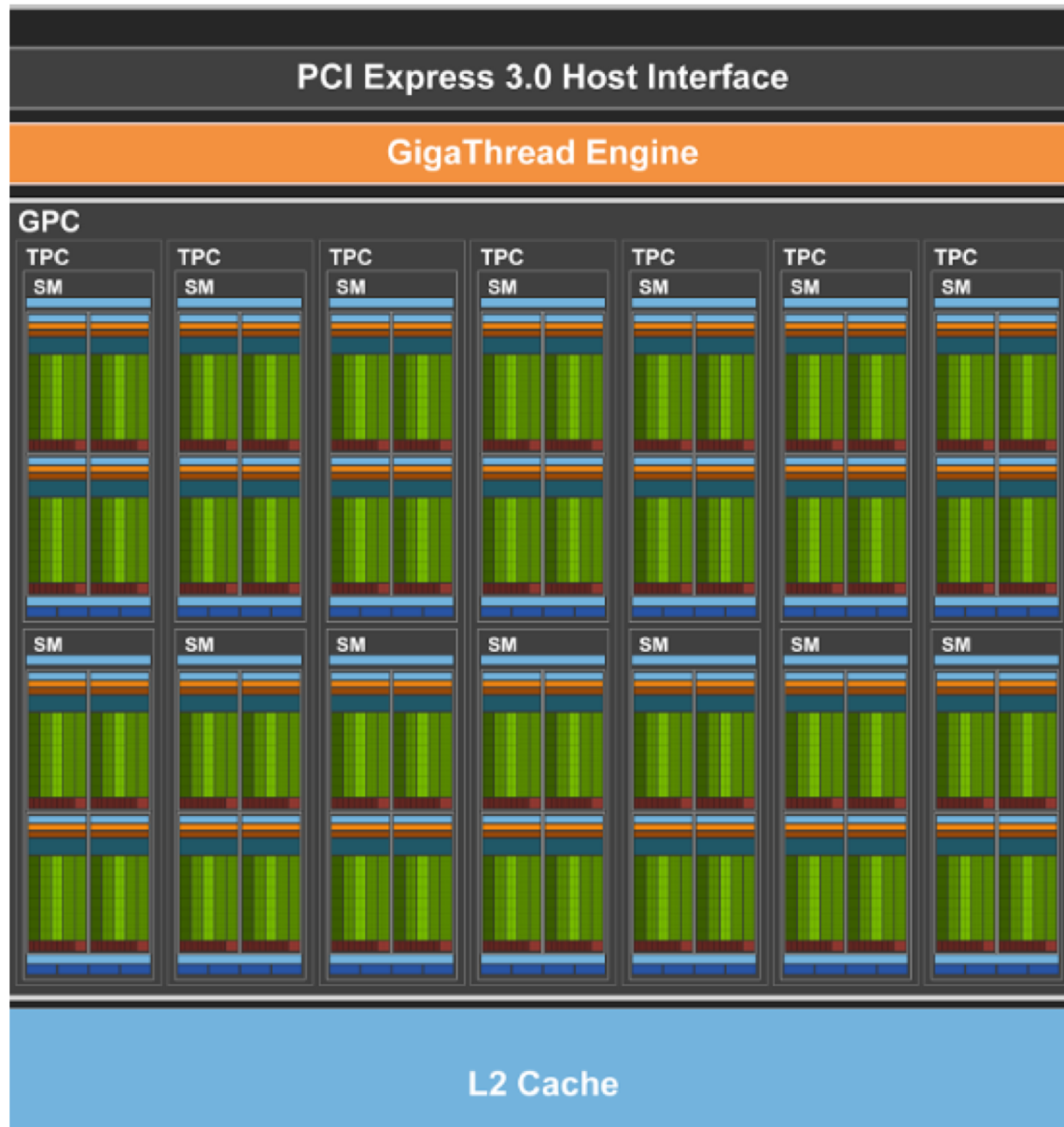


In a 1D grid `blockDim.x` is equal to threads per block

# V100 Volta Overview



# V100 Volta zoomed in



80 SMs per GPU

# Thread Execution Exampel in CUDA

```
add<<<10080,1024>>>(d_a, d_b, d_c);
```

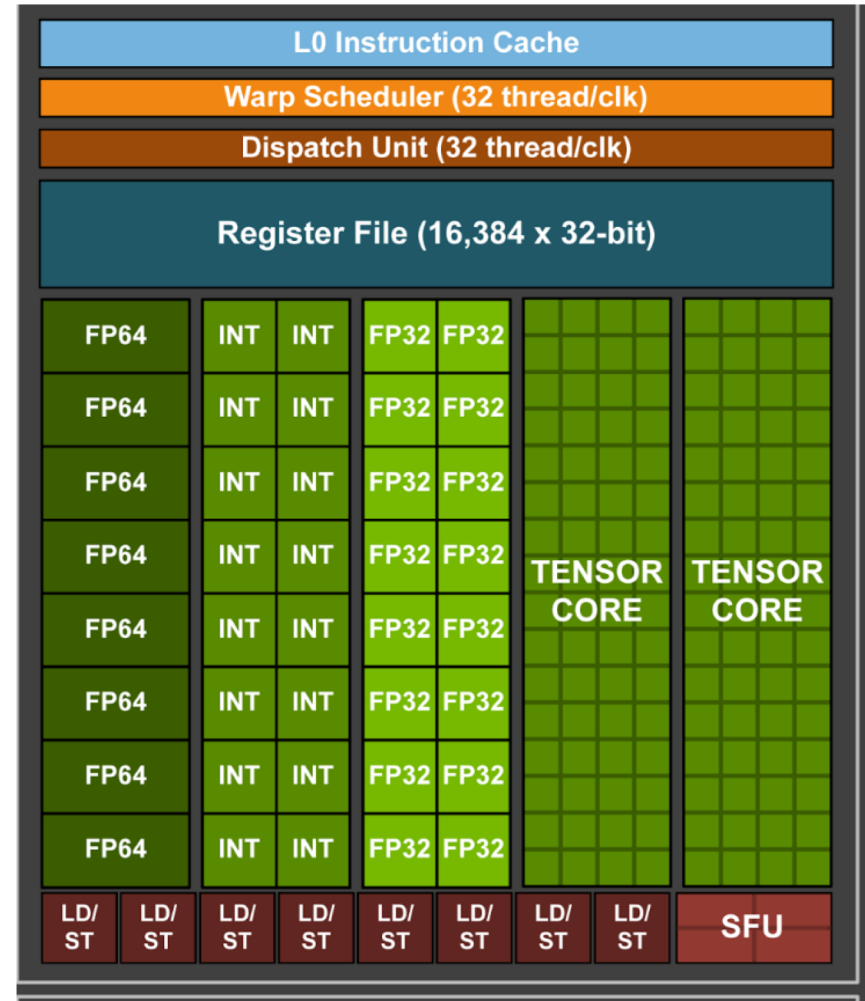
- $10000 * 1024 = 10,321,920$  threads in total
- 2 thread blocks per streaming multiprocessor (SM)
- 160 thread blocks / 81,920 threads run in parallel
- 63 consecutive thread block executions

# Thread Execution Example in CUDA

```
add<<<10080,1024>>>(d_a, d_b, d_c);
```

- $10000 * 1024 = 10,321,920$  threads in total
- 2 thread blocks per streaming multiprocessor (SM)
- 160 thread blocks / 163,840 threads run in parallel
- 63 consecutive thread block executions
- Advantage: Programmer does not have to worry about consecutive/concurrent computation
- Disadvantage: Threads in different blocks do not “see” each other

# Thread execution on the SM



# SIMT – GPU version of SIMD

## Single Instruction Multiple Thread

- 32 threads execute the same instruction concurrently
- Best to use 64, 128, 256, 512, or 1024 threads per block
- 32 concurrently running threads are called a **Warp**
- Memory accesses are warp-sized
- **SIMT** is hidden from the programmer

 **this leads to two problems**



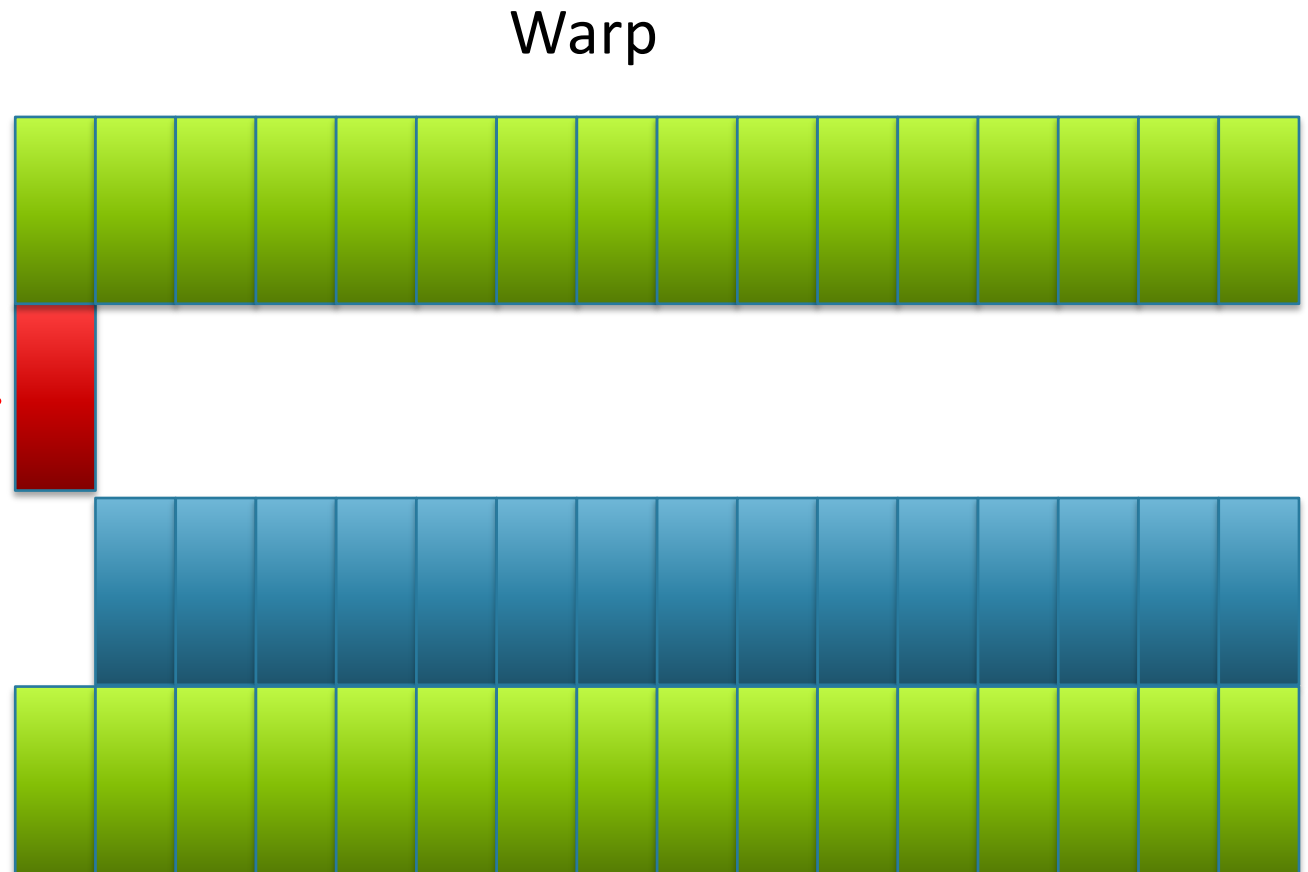
# SIMT Problems: Warp Divergence

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if(index == 0)  
        c[index] = a[index]*2;  
    else  
        c[index] = a[index] + b[index];  
}
```

**SIMT system cannot execute both paths at the same time.**

# SIMT Problems: Warp Divergence

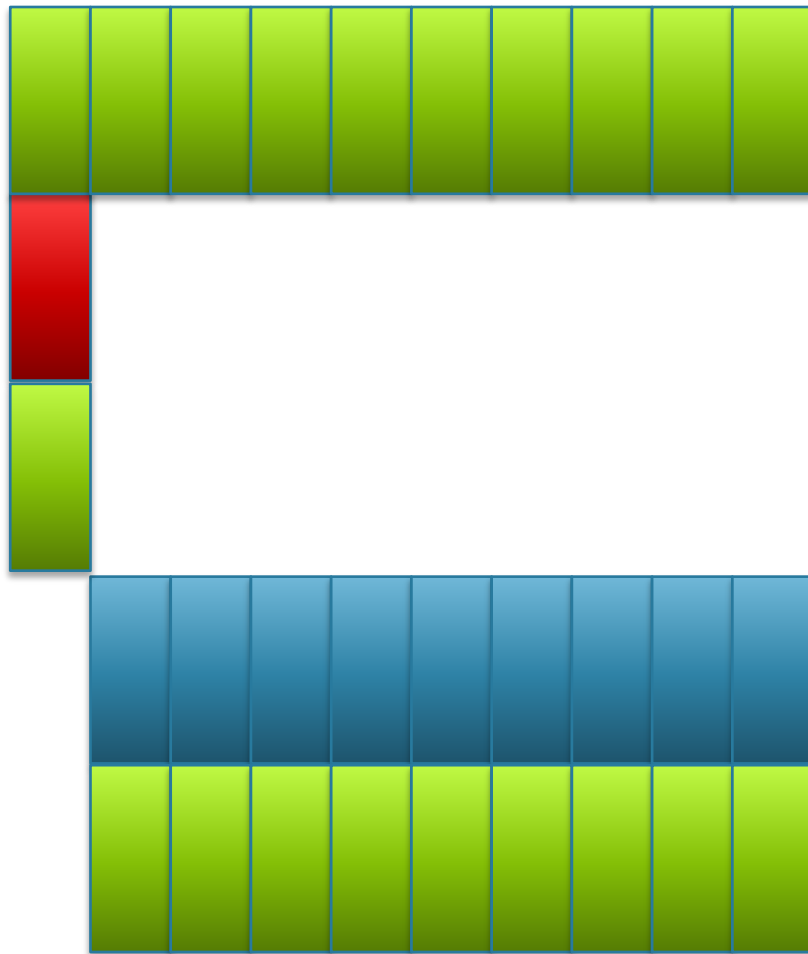
```
int index =  
threadIdx.x  
+ blockIdx.x  
* blockDim.x;  
if(index == 0)  
c[index]=a[index]*2;  
else  
c[index] = a[index]  
+ b[index];  
return 0;
```



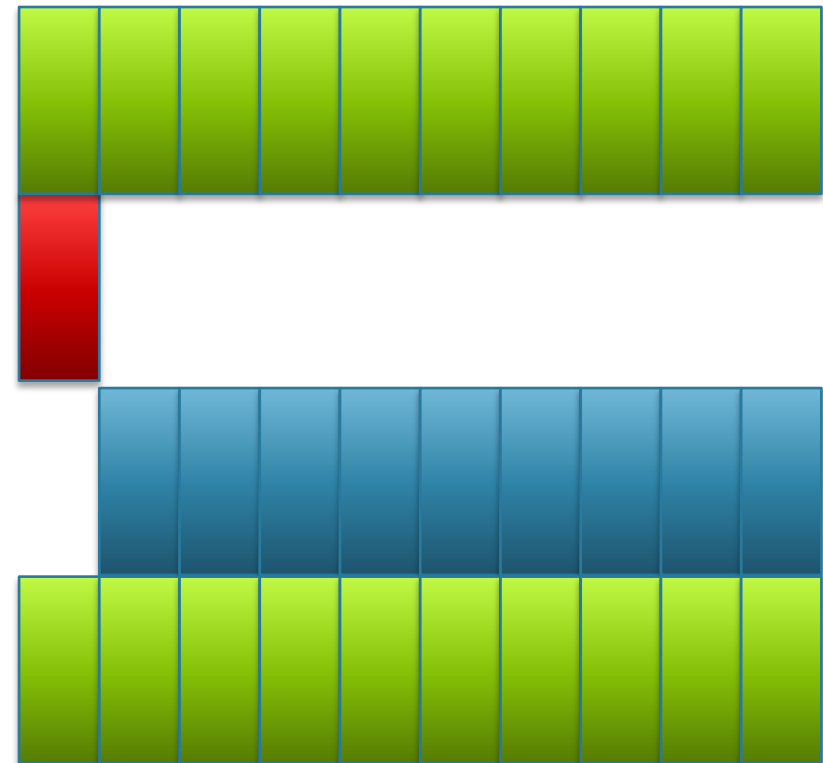
**Somewhat improved in Volta generation, but still SIMT (SIMD).**

# SIMT Problems: Warp Divergence

Pre-Volta

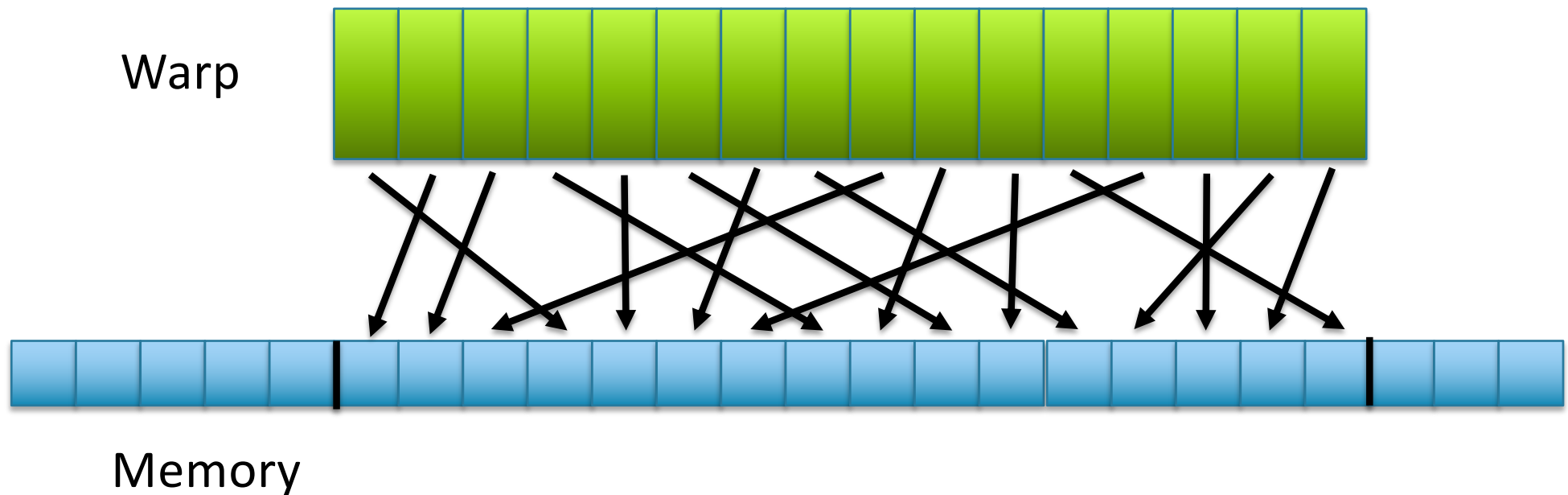


Volta, Turing, future



# SIMT Problems: Coalescing

A warp of 32 threads must read 32 contiguous elements from an array to get the maximum memory bandwidth, although elements can be swapped between the threads.



# Thread execution on the GPU

GPU code is most efficient when...

- Threads perform the same computation (no else statements)
- Threads read from consecutive memory locations
- Memory accesses are regular
- Enough threads to saturate device

# Enough threads to saturate device: Occupancy

80 SMs, 2048 threads each = 163,840 concurrent threads

## Do we need all of them ?

For FLOPS: perform up to  $2 * 32$  DP flops per cycle per SM

For memory: each SM needs to request about 6KB constantly



# Enough threads to saturate device: Occupancy

80 SMs, 2048 threads each = 163,840 concurrent threads

## Do we need all of them ?

For FLOPS: perform up to  $2 * 32$  DP flops per cycle per SM

For memory: each SM needs to request about 6KB constantly

## What does **constantly** mean ?

Memory latency:  $\sim 1000$  cycles @ 1.6 GHz = 625ns

800 GB/s = 10 GB/s per SM = 6250 KB

Full occupancy = 2048 threads

$$\underline{6250/2048 \sim 3 \text{ Byte/thread}}$$

# Enough threads to saturate device: Occupancy

High occupancy helps in hiding latency

Low occupancy leads to stalls when threads wait for new data

## Reasons for low occupancy:

- Block size smaller than 64 (maximum of 32 blocks per SM)
- Insufficient shared memory
- Insufficient registers



# How many Registers can a Kernel use ?

High occupancy helps in hiding latency

Low occupancy leads to stalls when threads wait for new data

**Volta V100** (same for most GPUs)

- Maximum of 255 registers per thread
- 64k registers of 32 bit (64 bit values take 2 registers.)
- $64k/2048 = 32$

# How many Registers can a Kernel use ?

High occupancy helps in hiding latency

Low occupancy leads to stalls when threads wait for new data

**Volta V100** (same for most GPUs)

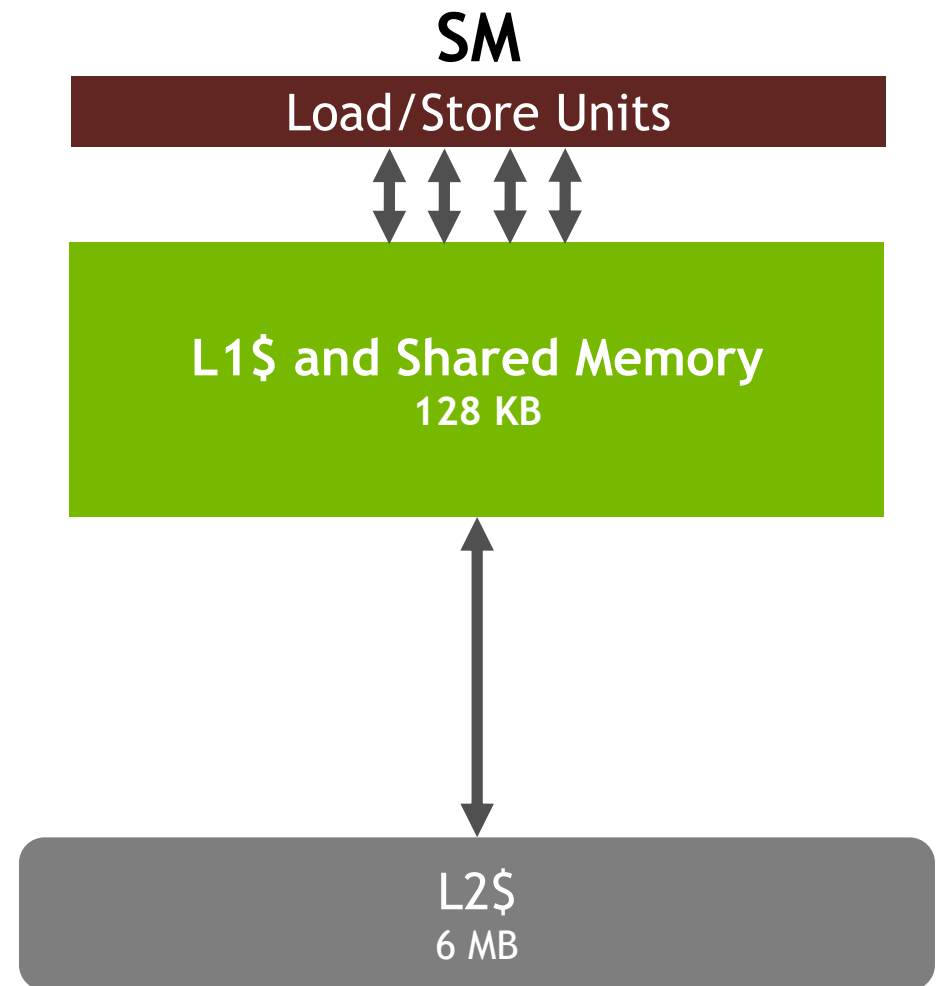
- Maximum of 255 registers per thread
- 64k registers of 32 bit (64 bit values take 2 registers.)
- $64k/2048 = 32$

**At 2048 threads, each thread can use 32 registers**

- Thread blocks must fit entirely in registers.
- 33 register kernel: 1 block of 1024 vs 31 blocks of 64 threads

# L1 Cache and Shared Memory

- Each SM can use up to 96 KB L1 as shared memory
- Shared memory is user managed
- 20-40x lower latency than DRAM
- 15x higher bandwidth than DRAM
- No coalescing necessary

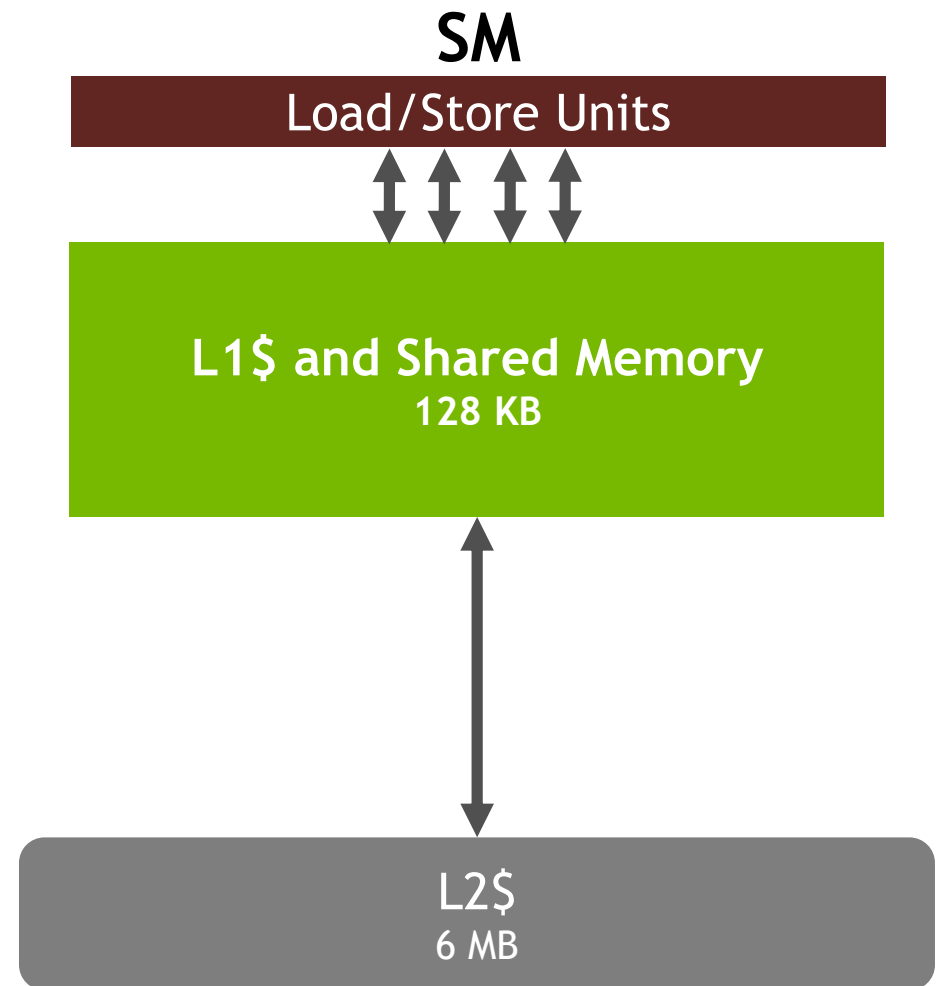


# L1 Cache and Shared Memory

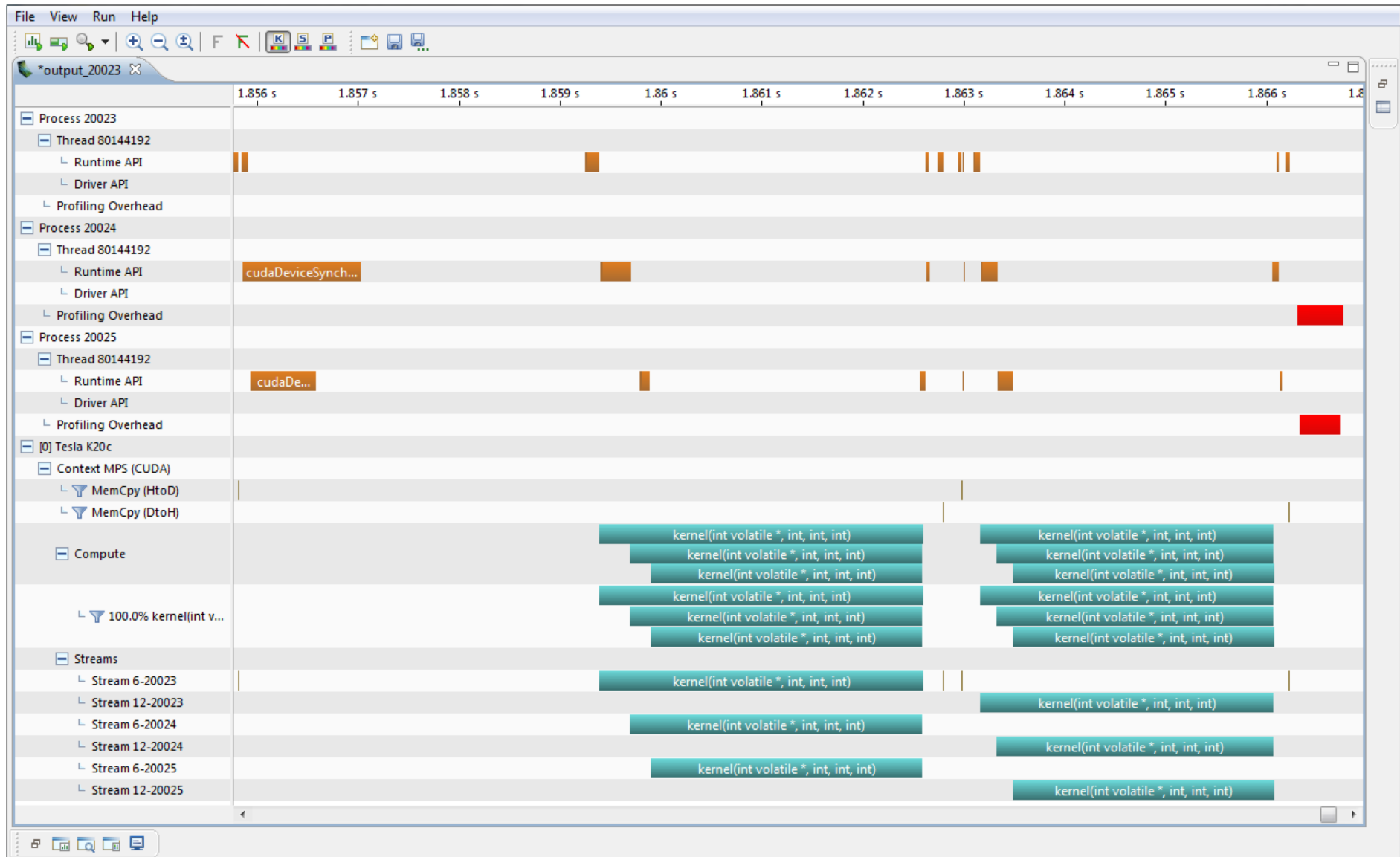
- Each SM can use up to 96 KB L1 as shared memory
- Shared memory is user managed
- 20-40x lower latency than DRAM
- 15x higher bandwidth than DRAM
- No coalescing necessary

2048 threads / 96KB shared memory

~ 21 Byte per thread at  
max. occupancy



# How to check Occupancy: Nvprof profiler



# More on CUDA: nvcc compiler

Part of the CUDA toolkit

Free to download, works with all NVIDIA GPUs

Usage like normal C compiler

```
nvcc test.cu
```

Select target GPU generation with:

```
-gencode arch=compute_xy,code=sm_xy
```

Switch underlying compiler

```
-ccbin
```



# More on CUDA: Samples

```
langguth@lizhi:~$ ls /usr/local/cuda-10.1/samples
0_Simple  1_Uutilities  2_Graphics  3_Imaging  4_Finance
5_Simulations  6_Advanced  7_CUDA Libraries
```

```
langguth@lizhi:~$ ls /usr/local/cuda
10.1/samples/0_Simple/
```

```
asyncAPI          fp16ScalarProduct  simpleAssert_nvrtc  simpleMPI          simpleSurfaceWrite  template
cdpSimplePrint    immaTensorCoreGemm simpleAtomicIntrinsics  simpleMultiCopy    simpleTemplates     UnifiedMemoryStreams
cdpSimpleQuicksort inlinePTX           simpleAtomicIntrinsics_nvrtc  simpleMultiGPU     simpleTemplates_nvrtc  vectorAdd
clock             inlinePTX_nvrtc    simpleCallback      simpleOccupancy    simpleTexture       vectorAddDrv
clock_nvrtc       matrixMul          simpleCooperativeGroups  simpleP2P          simpleTextureDrv    vectorAdd_nvrtc
cppIntegration    matrixMulCUBLAS    simpleCubemapTexture  simplePitchLinearTexture  simpleVoteIntrinsics
cppOverload      matrixMulDrv       simpleCudaGraphs     simplePrintf       simpleVoteIntrinsics_nvrtc
cudaOpenMP       matrixMul_nvrtc    simpleIPC            simpleSeparateCompilation  simpleZeroCopy
cudaTensorCoreGemm simpleAssert        simpleLayeredTexture  simpleStreams      systemWideAtomics
```

## vectorAdd

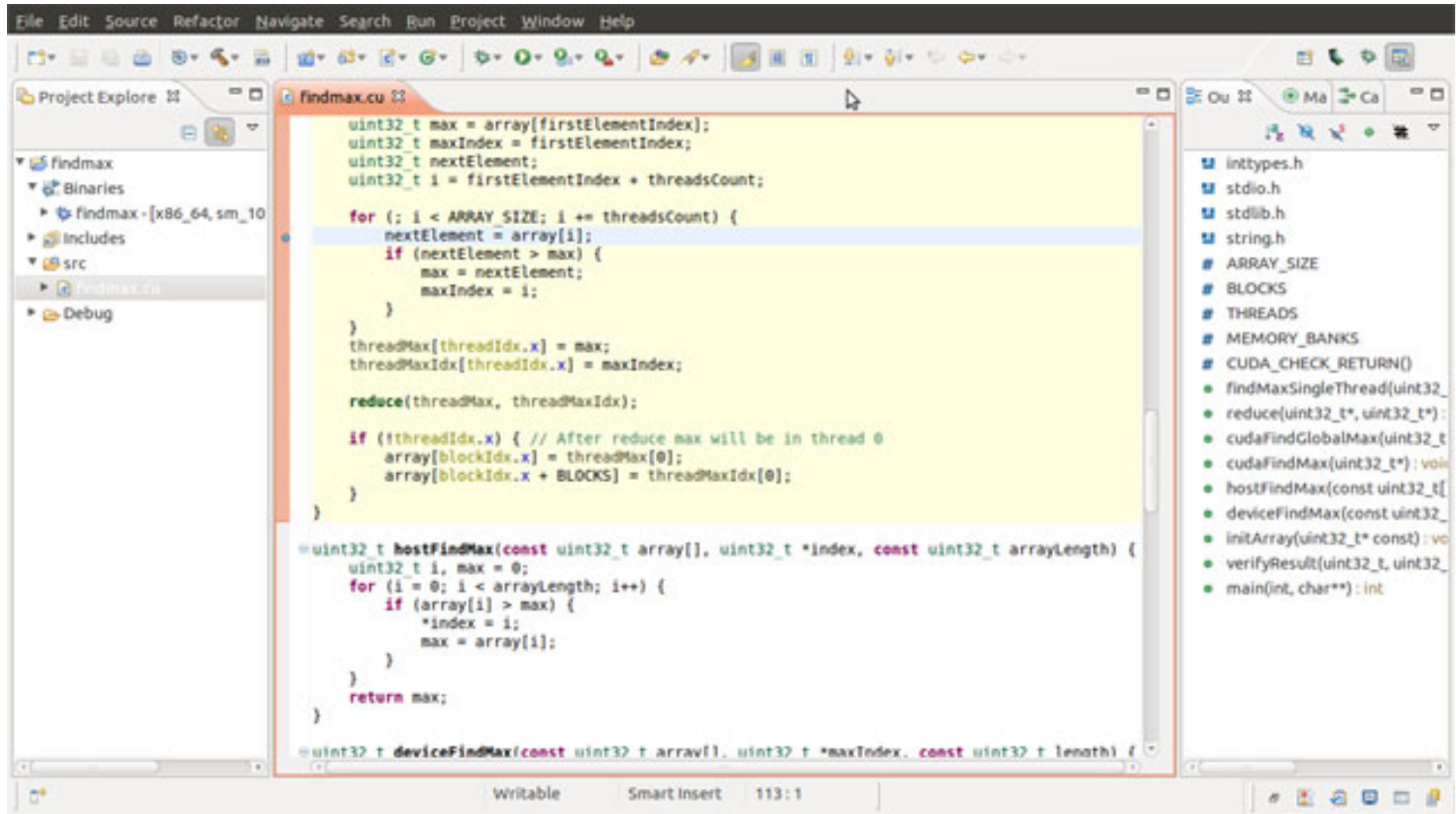
# Learning from CUDA Samples: vectorADD

```
__global__ void vectorAdd(const float *A,  
const float *B, float *C, int numElements)  
{  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if (i < numElements)  
    {  
        C[i] = A[i] + B[i];  
    }  
}
```

**GPU has dedicated cache for constants. Use it.**



# More on CUDA: Nsight editor



```
File Edit Source Refactor Navigate Search Run Project Window Help
Project Explorer
findmax
  Binaries
    findmax-[x86_64,sm_10]
  Includes
  src
    findmax.cu
  Debug
findmax.cu
uint32_t max = array[firstElementIndex];
uint32_t maxIndex = firstElementIndex;
uint32_t nextElement;
uint32_t i = firstElementIndex + threadsCount;

for (; i < ARRAY_SIZE; i += threadsCount) {
    nextElement = array[i];
    if (nextElement > max) {
        max = nextElement;
        maxIndex = i;
    }
}
threadMax[threadIdx.x] = max;
threadMaxIdx[threadIdx.x] = maxIndex;

reduce(threadMax, threadMaxIdx);

if (!threadIdx.x) { // After reduce max will be in thread 0
    array[blockIdx.x] = threadMax[0];
    array[blockIdx.x + BLOCKS] = threadMaxIdx[0];
}

uint32_t hostFindMax(const uint32_t array[], uint32_t *index, const uint32_t arrayLength) {
    uint32_t i, max = 0;
    for (i = 0; i < arrayLength; i++) {
        if (array[i] > max) {
            *index = i;
            max = array[i];
        }
    }
    return max;
}

uint32_t deviceFindMax(const uint32_t array[], uint32_t *maxIndex, const uint32_t length) {
    inttypes.h
    stdio.h
    stdlib.h
    string.h
    # ARRAY_SIZE
    # BLOCKS
    # THREADS
    # MEMORY_BANKS
    # CUDA_CHECK_RETURN()
    • findMaxSingleThread(uint32_t, uint32_t*)
    • reduce(uint32_t*, uint32_t*)
    • cudaFindGlobalMax(uint32_t, uint32_t*)
    • cudaFindMax(uint32_t, uint32_t*)
    • hostFindMax(const uint32_t, uint32_t*)
    • deviceFindMax(const uint32_t, uint32_t*)
    • initArray(uint32_t, const uint32_t)
    • verifyResult(uint32_t, uint32_t)
    • main(int, char**) : int
Writable SmartInsert 113:1
```

# More on CUDA: Libraries



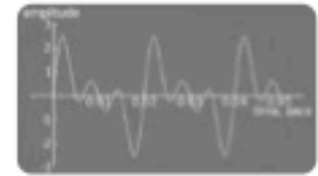
NVIDIA cuBLAS



NVIDIA cuSPARSE



NVIDIA NPP



NVIDIA cuFFT



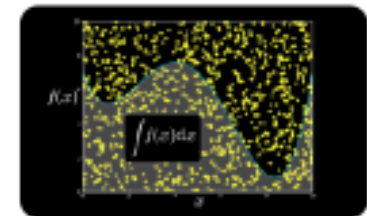
Matrix Algebra on GPU and Multicore



GPU Accelerated Linear Algebra



Vector Signal Image Processing



NVIDIA cuRAND



IMSL Library



CenterSpace NMath

ArrayFire



Building-block Algorithms



C++ Templated Parallel Algorithms

Credit: Lecture contains NVIDIA material available at <https://developer.nvidia.com/cuda-zone> 42