GP-GPU PROGRAMMING WITH CUDA

Larry Brown Ph.D.
larryb@nvidia.com

AGENDA

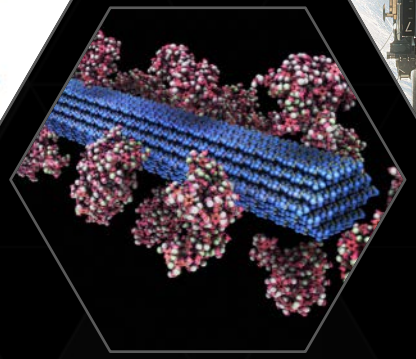
1 NVIDIA Intro
2 GP-GPU Intro
3 Accelerating Code with CUDA
4 Optimizations
5 Key Applications

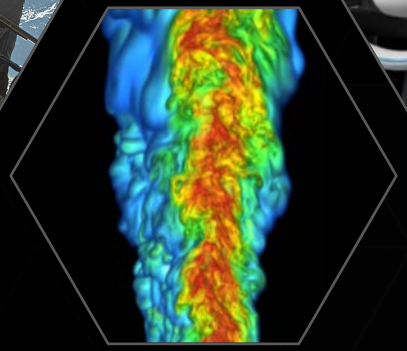
NVIDIA - INVENTOR OF THE GPU

NVIDIA Invented the GPU in 1999, with over 1 Billion shipped to date.

Initially a dedicated a graphics processor for PCs, the GPU's computational power and energy efficiency led to it quickly being adopted for professional and scientific simulation and visualization.

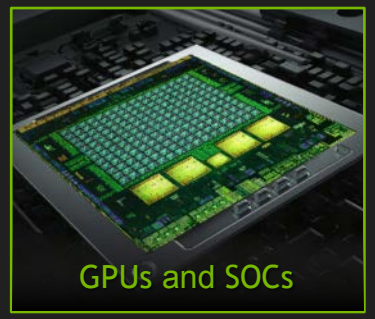
In 2007, NVIDIA launched the **CUDA®** programming platform, and opened up the general purpose parallel processing capabilities of the GPU.

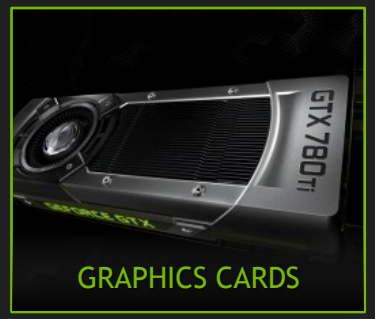




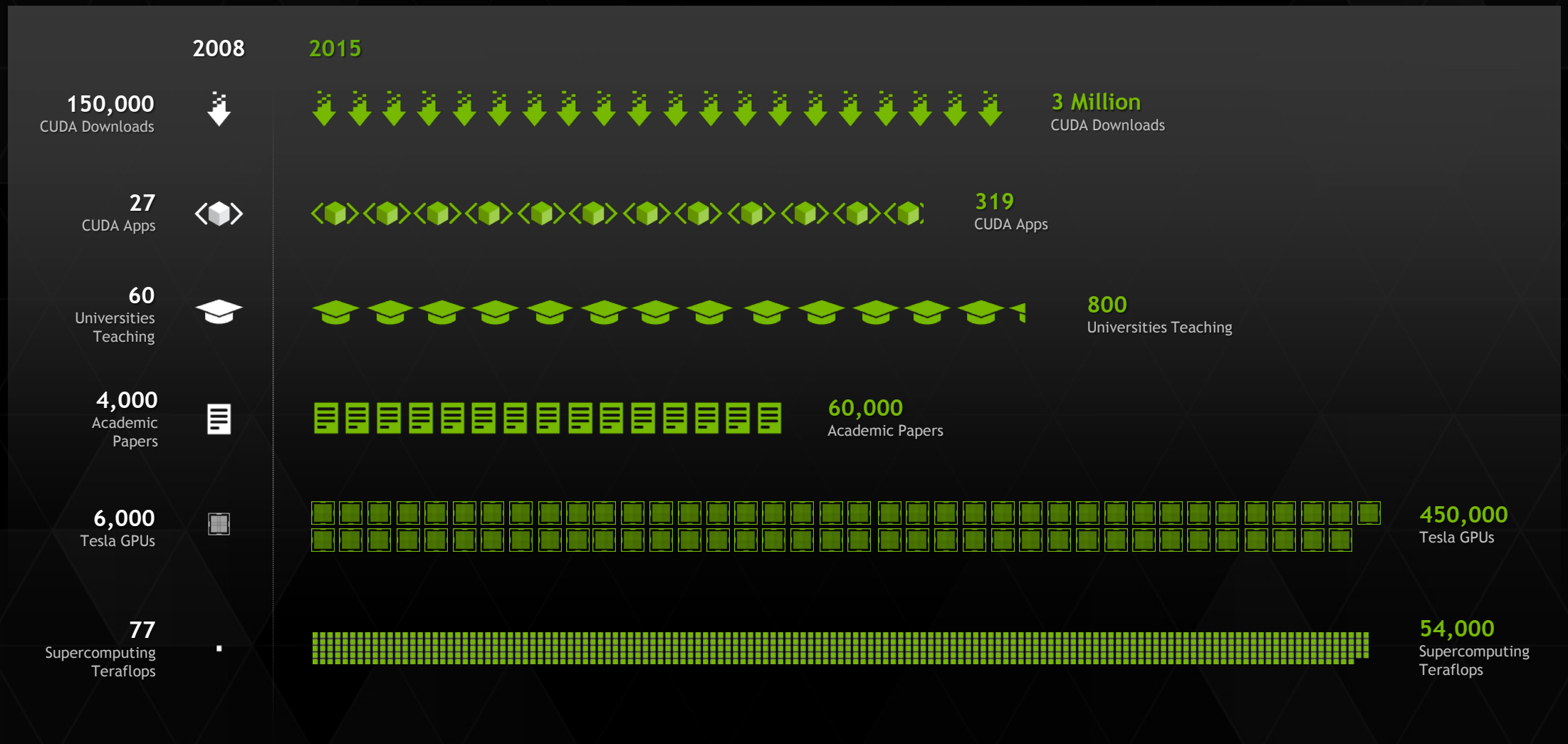
THE GREEN 500™

NVIDIA PLATFORM

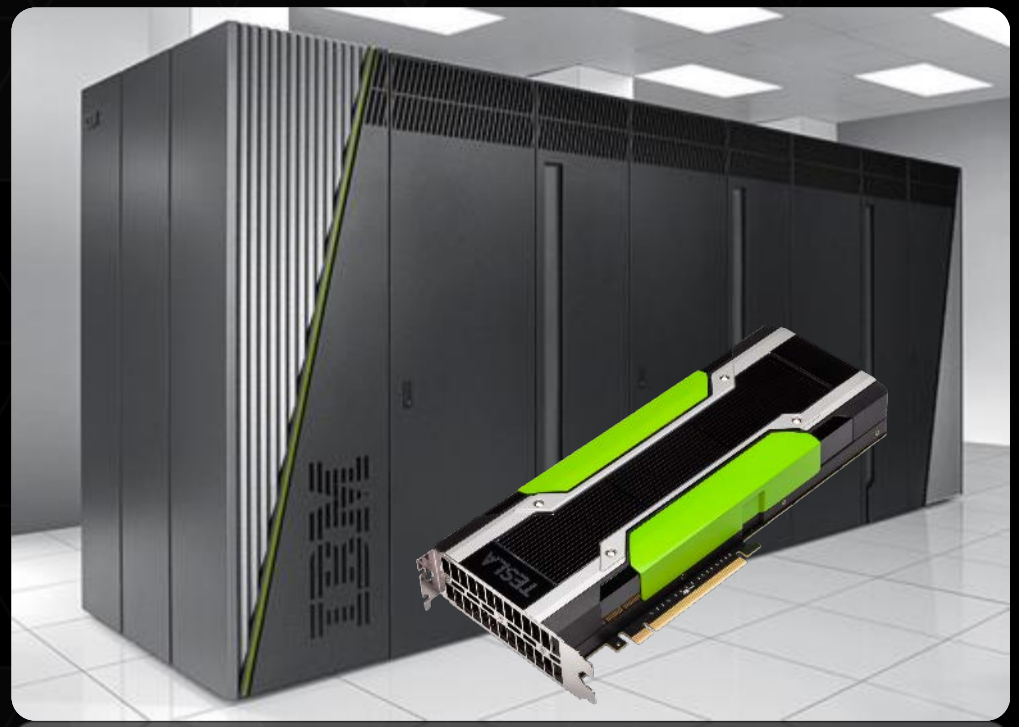




10X GROWTH IN GPU COMPUTING



USA - TWO FLAGSHIP SUPERCOMPUTERS



Oak Ridge National Laboratory

SUMMIT

150-300 PFLOPS
Peak Performance

Lawrence Livermore National Laboratory

SIERRA

> 100 PFLOPS
Peak Performance

IBM POWER9 CPU + NVIDIA Volta GPU

NVLink High Speed Interconnect

>40 TFLOPS per Node, >3,400 Nodes

2017

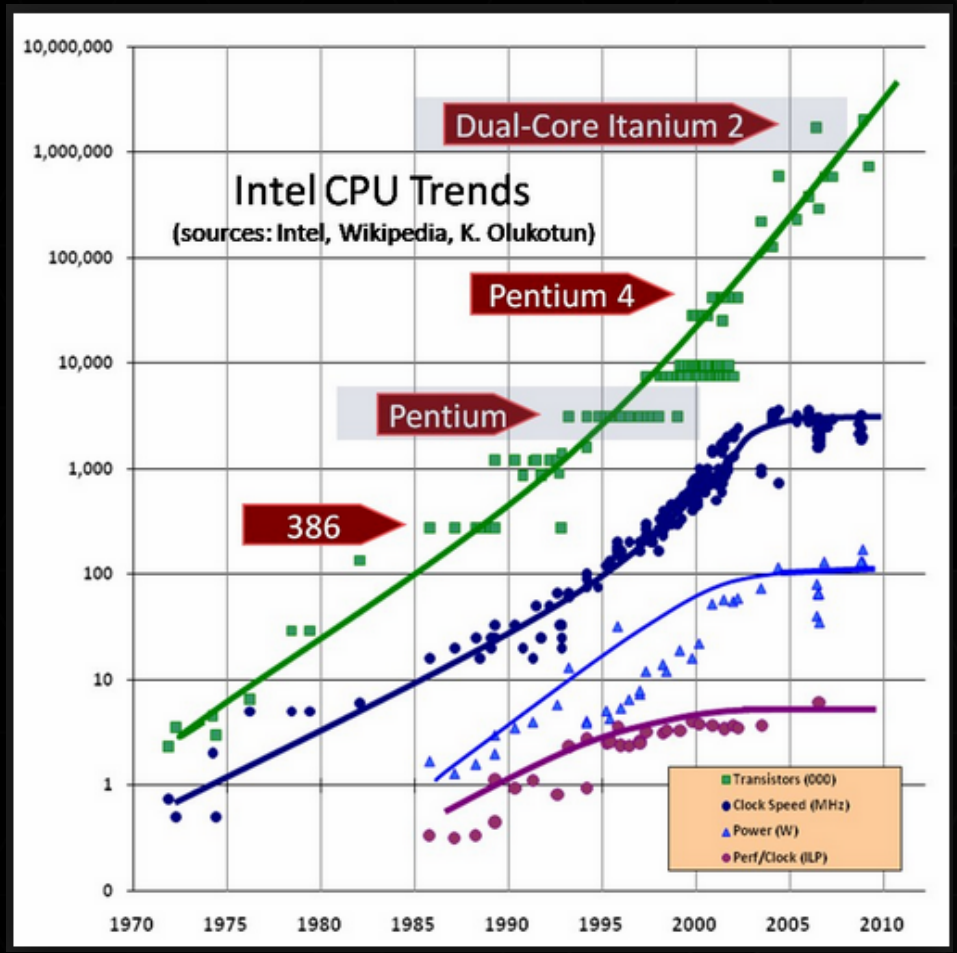
Powered by the NVIDIA GPU

END OF TRADITIONAL FREQUENCY SCALING

The Free Lunch is Over:
A Fundamental Turn towards
Concurrency in Software

Herb Sutter

The breakdown is in Dennard scaling
…Moores law marches on.

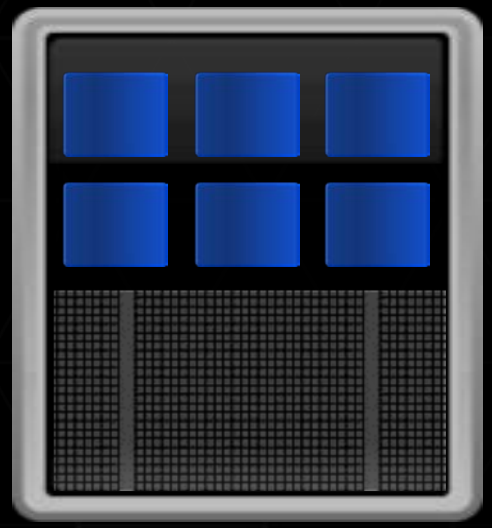


GP-GPU Programming

HETEROGENEOUS COMPUTING

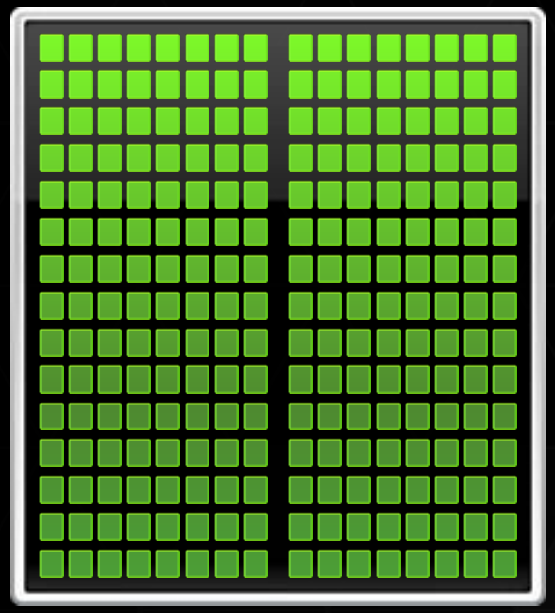
10X PERFORMANCE 5X ENERGY EFFICIENCY

CPU
Optimized for
Serial Tasks

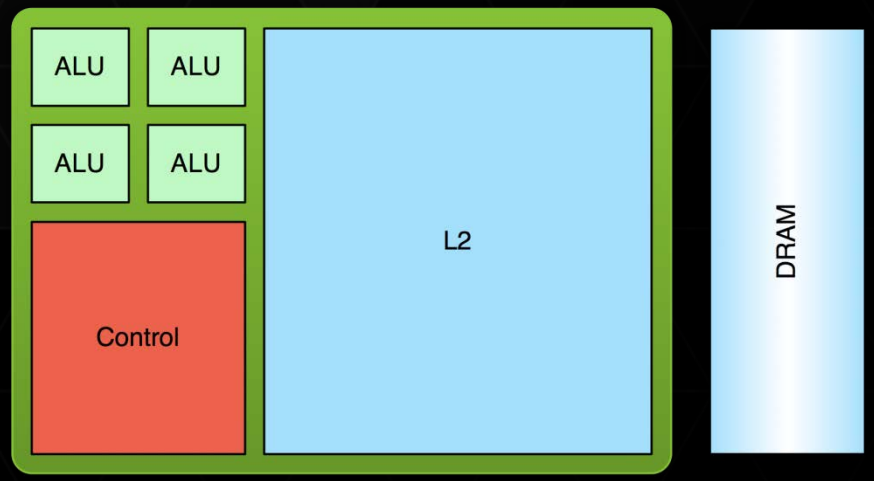


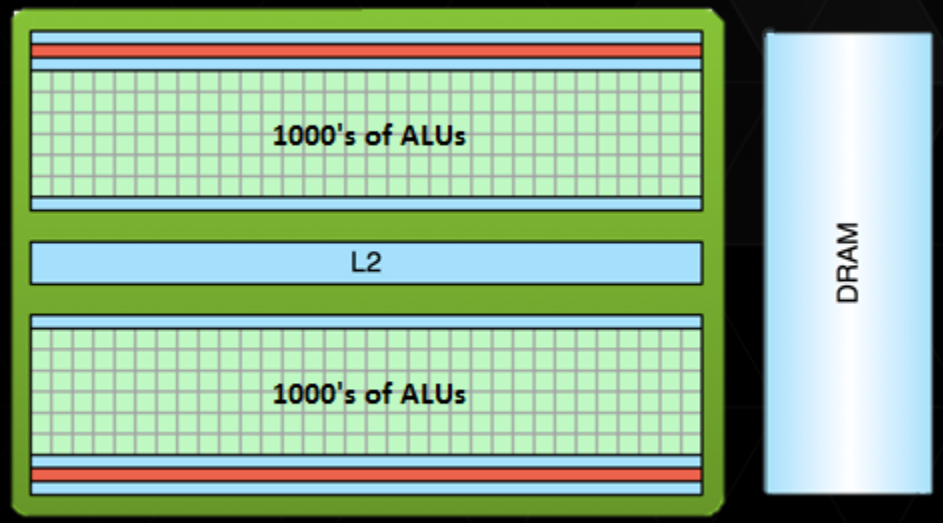
+

GPU Accelerator
Optimized for
Parallel Tasks



LOW LATENCY OR HIGH THROUGHPUT?





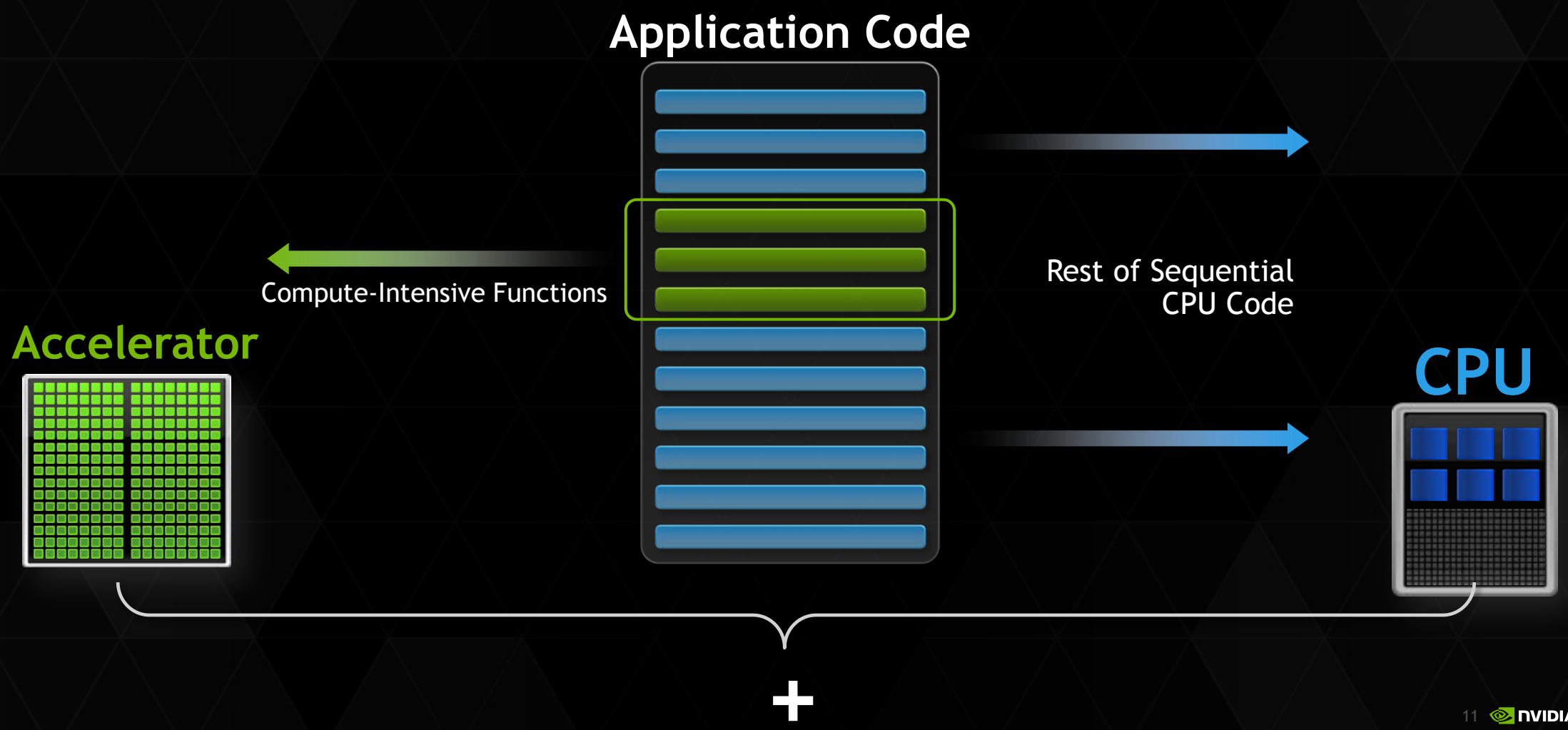
CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

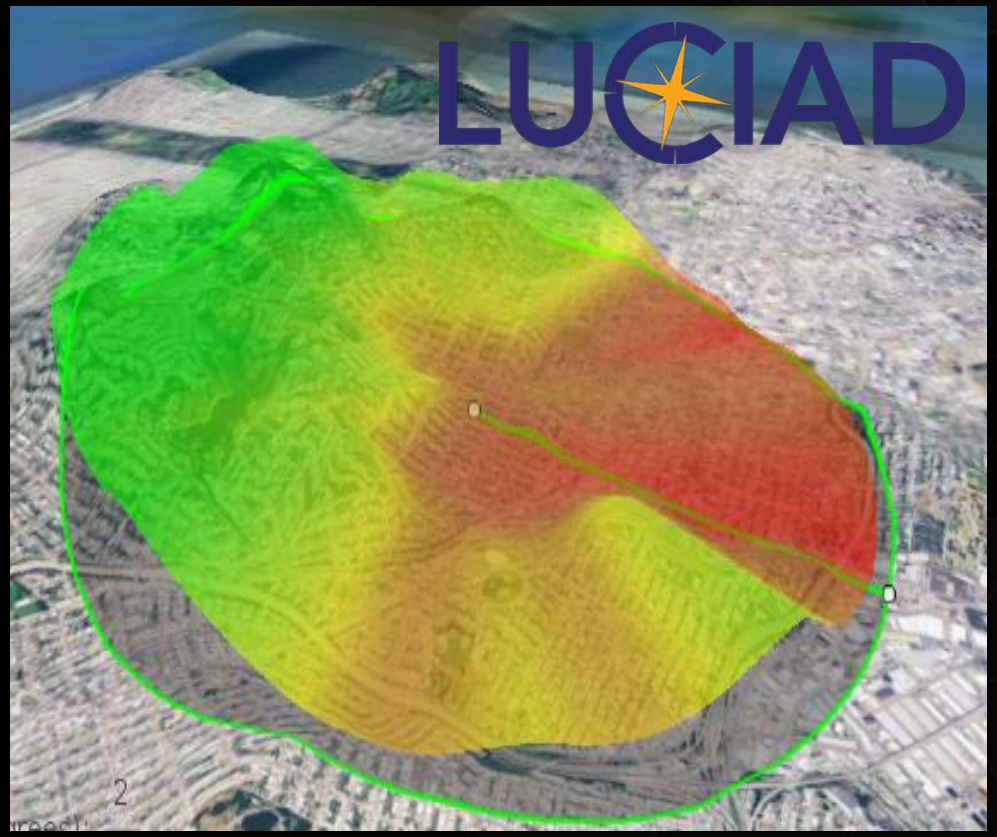
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

GPGPU ACCELERATION - THE BASIC IDEA



GPGPU IMPACT – REALTIME LINE-OF-SIGHT

Task	Line-of-sight at 60 fps
GPUs	100x faster than CPUs
Result	Instant analysis for mission preparation



http://www.brainshark.com/nvidia/demo-video-luciadlightspeed

http://www.luciad.com/

GPGPU IMPACT – FIND OBJECTS IN IMAGES

Task	Automatic objects search in multiple video/image formats
GPUs	20x faster than CPUs delivering 20,000 FPS
Result	Live and forensic search with ranking and alerts

http://www.nervvetechnologies.com/

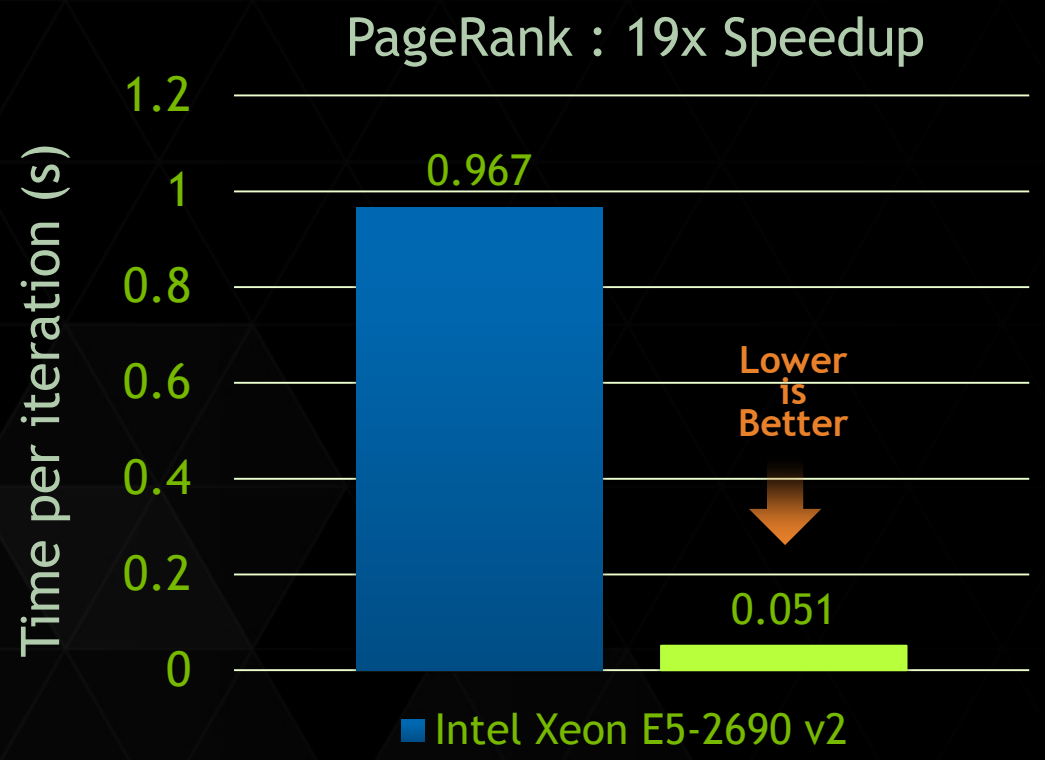
One hour of video searched in 5 seconds

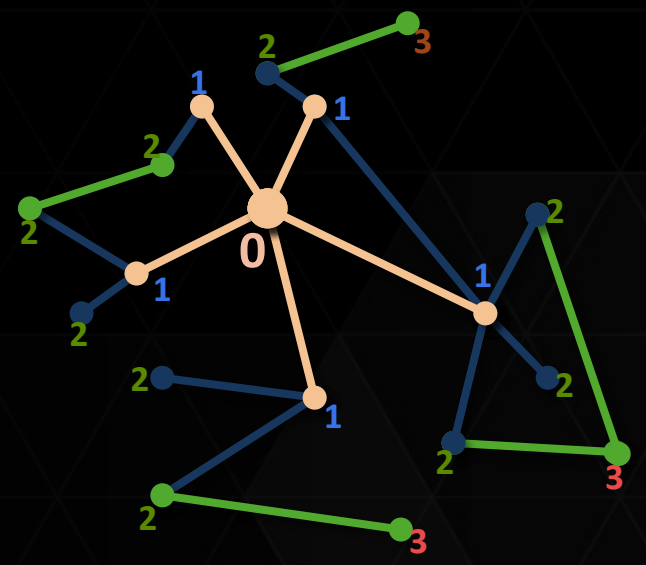
GPGPU IMPACT – GRAPH ANALYTICS

- Comms & Social networks
- Cyber pattern recognition
- Shortest path finding

1 GPU vs 60 Nodes

280x vs optimized Spark
1440x vs Spark
3420x vs Hadoop





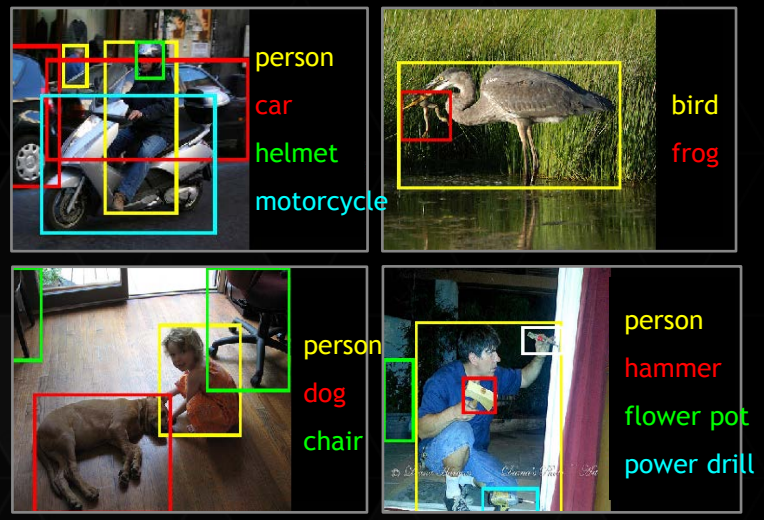
GPGPU IMPACT– DEEP LEARNING

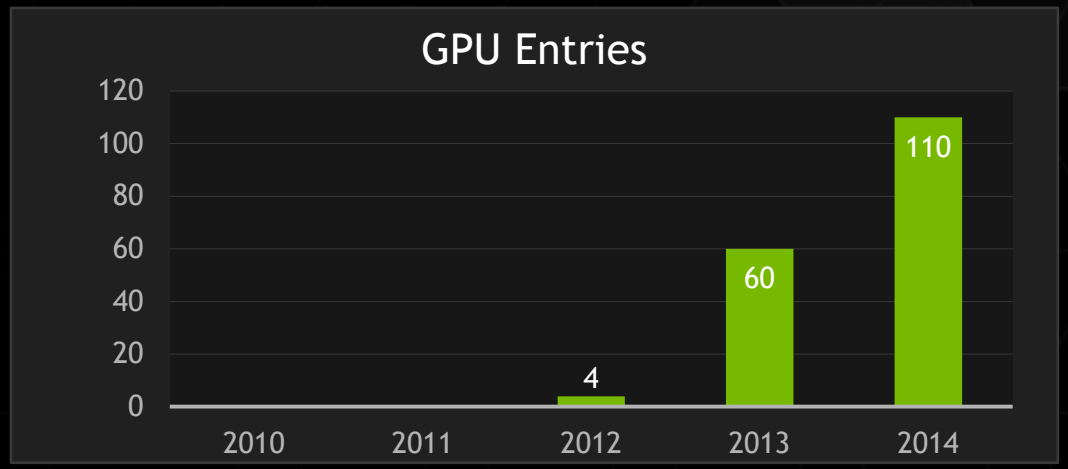
Image Recognition Challenge

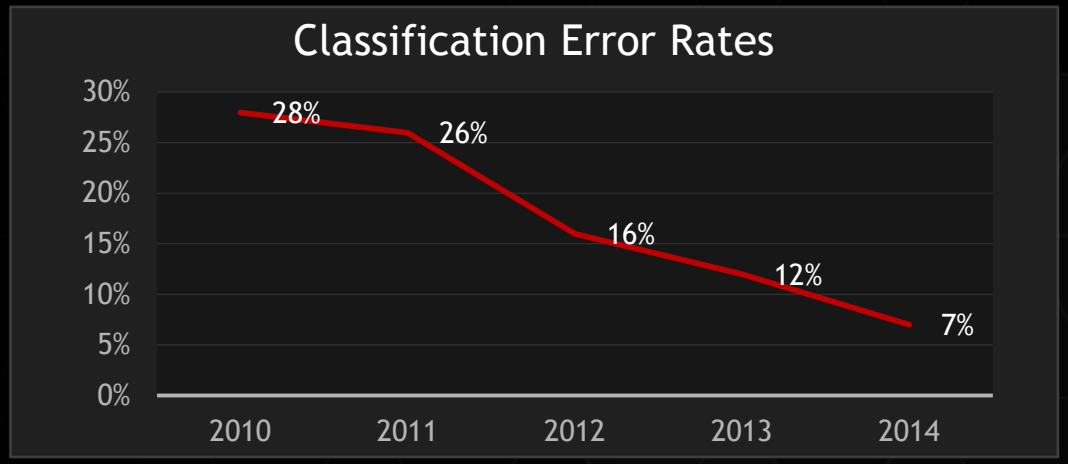
1.2M training images • 1000 object categories

Hosted by

IMAGENET







HOW DO I USE GPUS?

CUDA PARALLEL DEVELOPMENT PLATFORM

developer.nvidia.com

Programming Approaches

- **Libraries** — "Drop-in" Acceleration
- **Directives** — Easily Accelerate Apps
- **Programming Languages** — Maximum Flexibility

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

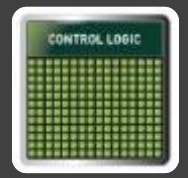
Open Compiler Tool Chain

LLVM COMPILER INFRASTRUCTURE

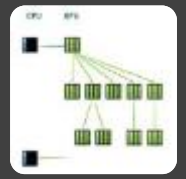
Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

Hardware Capabilities

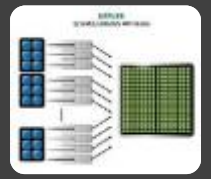
SMX



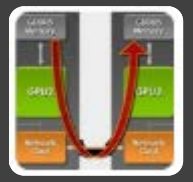
Dynamic Parallelism



HyperQ



GPUDirect



3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

ACCELERATING YOUR APPLICATIONS

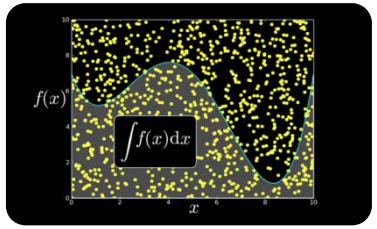
Simple & Portable Parallel Software Development

Applications

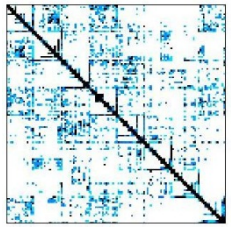
Libraries	Directives	Languages
"Drop-in" Acceleration	Annotate code with compiler hints	Modern language features (unified memory, for_each, lambda)

GPU Accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal
Image Processing

CULA tools

GPU Accelerated
Linear Algebra

MAGMA

ICL UT

Matrix Algebra on
GPU and Multicore

open source initiative

NVIDIA cuFFT

ROGUE WAVE
SOFTWARE

IMSL Library

ArrayFire Matrix
Computations

CUSP

Sparse Linear
Algebra

open source initiative

Thrust

C++ STL Features
for CUDA

open source initiative

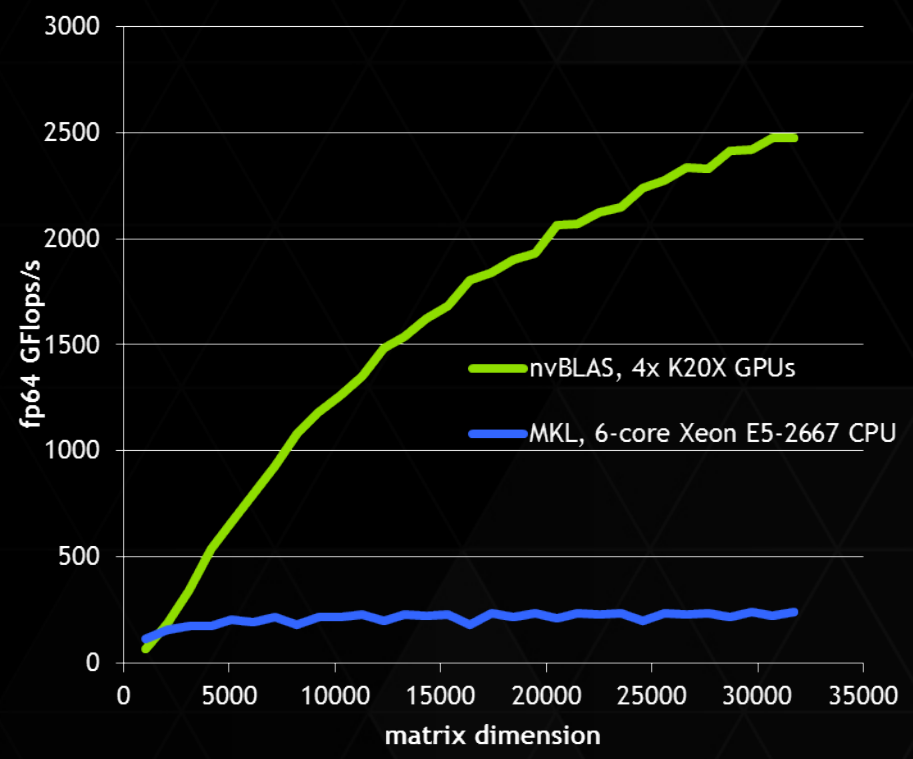
“DROP-IN” LIBRARY EXAMPLE: NVBLAS

- Automatic Speedup for “R” application

```
> LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so R

> A <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> B <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> system.time(C <- A %*% B)

   user  system elapsed

   0.348  0.142   0.289
```

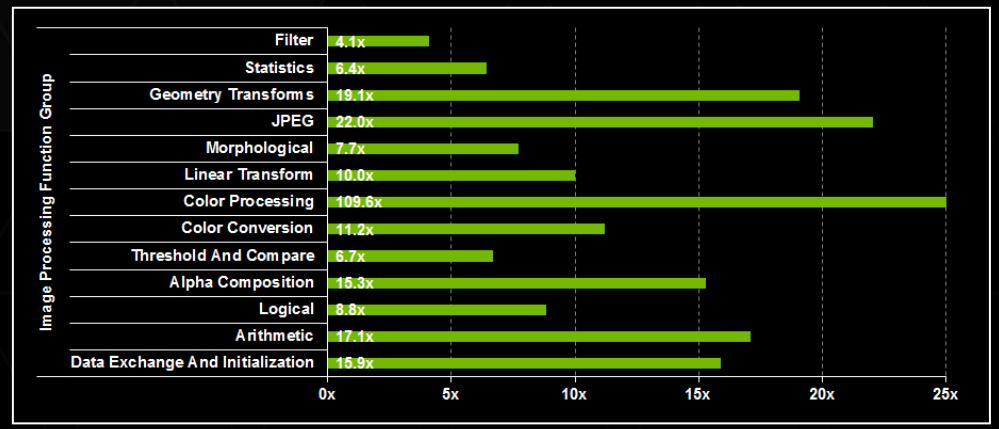
NO CODE CHANGE REQUIRED

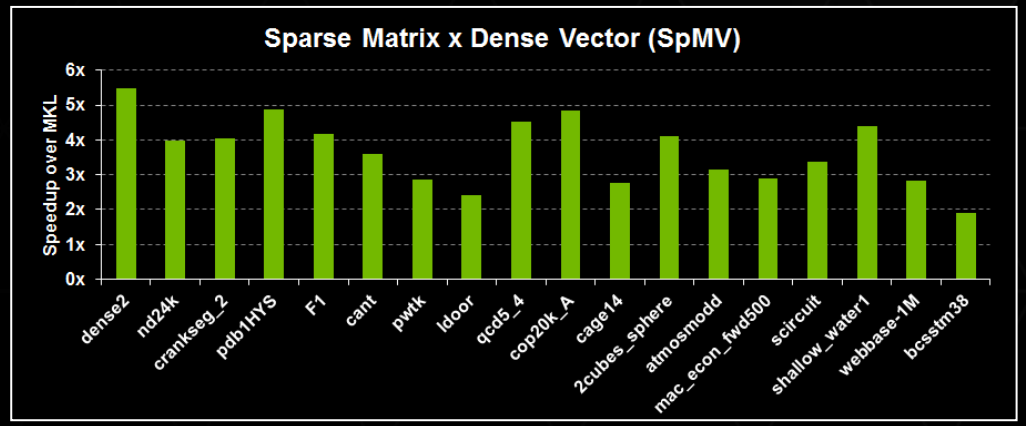
- Use in any app that uses standard BLAS3
  - R, Octave, Scilab, etc.

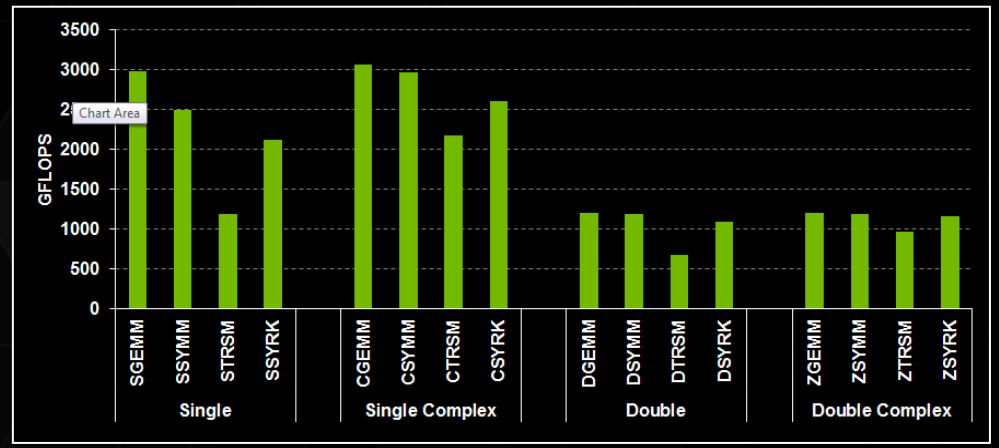


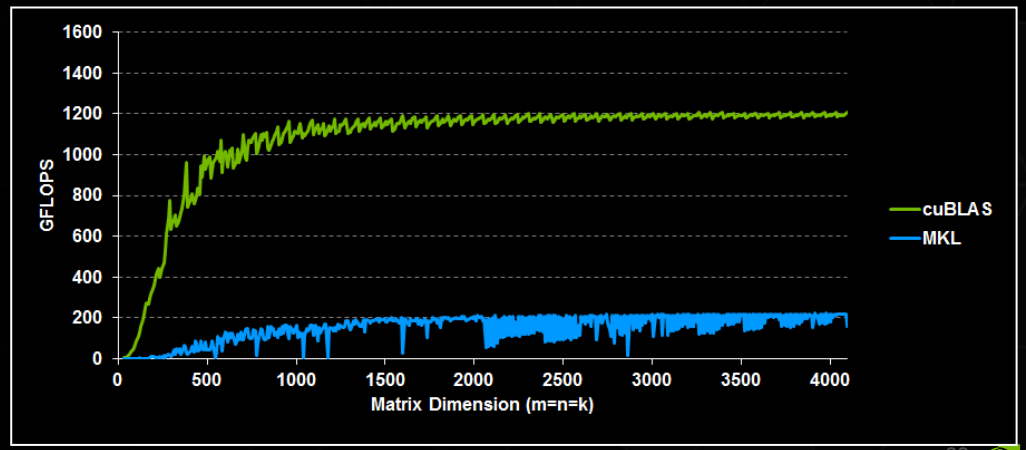
5X–10X SPEEDUP USING NVIDIA LIBRARIES

BLAS | LAPACK | SPARSE | FFT | Math | Deep Learning | Image Processing









WHAT IS A *DROP-IN* LIBRARY?

1. Uses standard or near-standard interface

2. No code change required
   - Recompile might not even be required

3. Data lives in CPU memory at the beginning and end of library routines.
   - Completely invisible to the program

ACCELERATING YOUR APPLICATIONS

Simple & Portable Parallel Software Development

Applications

Libraries	Directives	Languages
"Drop-in" Acceleration	Annotate code with compiler hints	Modern language features (unified memory, for_each, lambda)

OPENACC DIRECTIVES

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          z[i] = x[i] + y[i];
          ...
      }
  }
  ...
}
```

OpenACC
Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Simple compiler hints
- CPU, GPU, MIC
- Compiler parallelizes code

OpenACC

Simple | Powerful | Portable

```
main()
{
  <serial code>
  #pragma acc kernels
  //automatically runs on GPU
  {
    <parallel code>
  }
}
```

University of Illinois
PowerGrid- MRI Reconstruction

70x Speed-Up
2 Days of Effort

Fueling the Next Wave of Scientific Discoveries in HPC

RIKEN Japan
NICAM- Climate Modeling

7-8x Speed-Up
5% of Code Modified

8000+

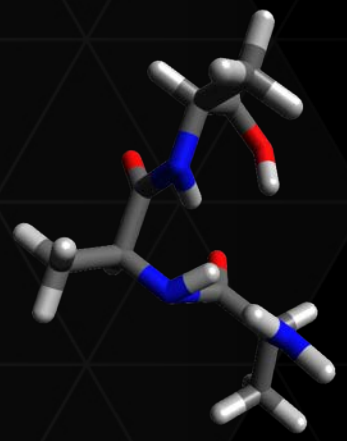
Developers

using OpenACC

http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.hpcwire.com/off-the-wire/first-round-of-2015-hackathons-gets-underway
http://on-demand.gputechconf.com/gtc/2015/presentation/S5297-Hisashi-Yashiro.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7

LS-DALTON

LARGE-SCALE APPLICATION FOR CALCULATING HIGH-ACCURACY MOLECULAR ENERGIES



"*OpenACC makes GPU computing approachable for domain scientists. Initial OpenACC implementation required only minor effort, and more importantly, no modifications of our existing CPU implementation.*"

Janus Juul Eriksen, PhD Fellow
qLEAP Center for Theoretical Chemistry, Aarhus University

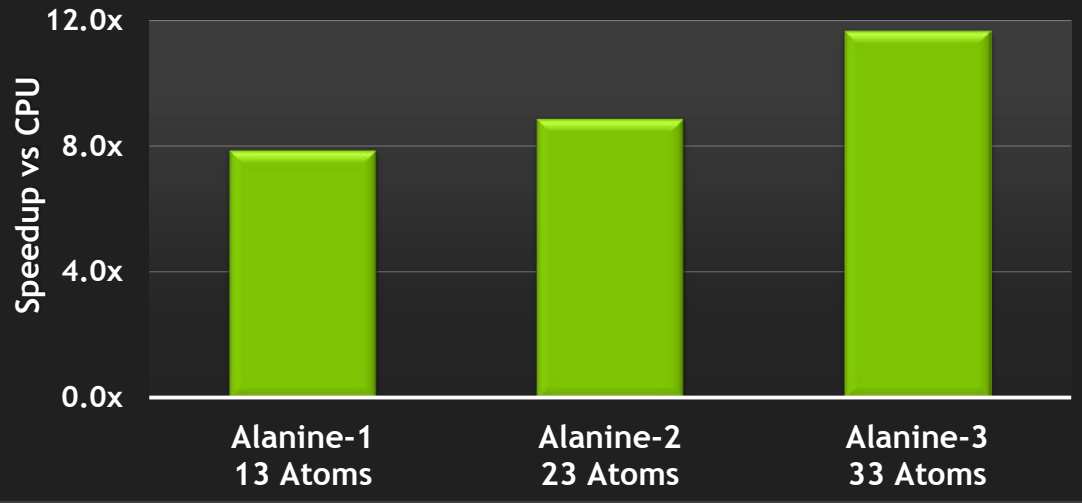
Minimal Effort

Lines of Code Modified	# of Weeks Required	# of Codes to Maintain
<100 Lines	1 Week	1 Source

Big Performance

LS-DALTON CCSD(T) Module

Benchmarked on Titan Supercomputer (AMD CPU vs Tesla K20X)



INTRODUCING THE NEW OPENACC TOOLKIT

Free Toolkit Offers Simple & Powerful Path to Accelerated Computing

http://developer.nvidia.com/openacc

PGI Compiler
Free OpenACC compiler for academia

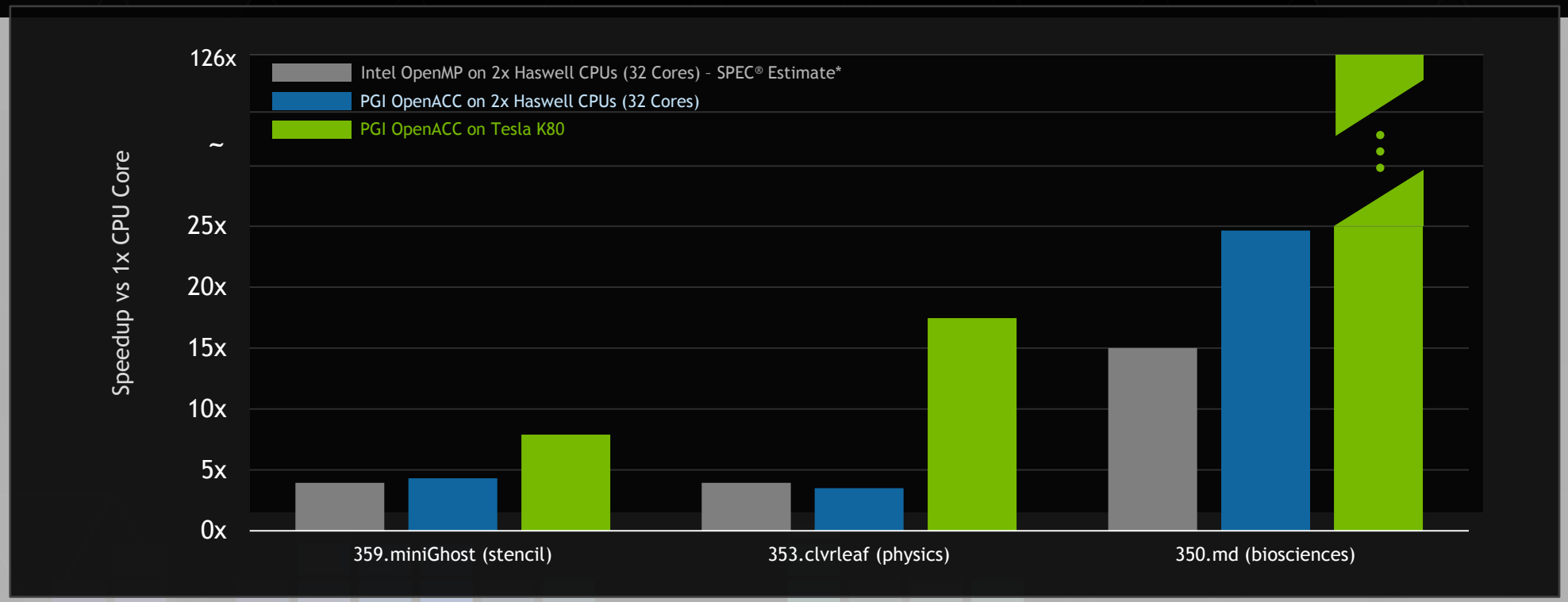
NVProf Profiler
Easily find where to add compiler directives

GPU Wizard
Identify which GPU libraries can jumpstart code

Code Samples
Learn from examples of real-world algorithms

Documentation
Quick start guide, Best practices, Forums

OPENACC DELIVERS PERFORMANCE PORTABILITY



Supermicro SYS-2028GR-TRT, Intel Xeon E5-2698 v3, 32 cores, NVIDIA Tesla K80, 256GB of System Memory
PGI 15.7 Beta OpenACC Multicore and K80 results from SPEC ACCEL™ measured June 2015.
* Intel 15.0.90 OpenMP results use Cloverleaf reference application and SPEC OMP®2012 using workloads from SPEC ACCEL.
SPEC® and the benchmark names SPEC ACCEL™ and SPEC OMP® are registered trademarks of the Standard Performance Evaluation Corporation.
More info: SPEC ACCEL www.spec.org/accel, OMP2012 www.spec.org/omp2012, CloverLeaf OpenMP uk-mac.github.io/CloverLeaf/, miniGhost ref 1.0.1 https://mantevo.org/download/

PGI®

ACCELERATING YOUR APPLICATIONS

Simple & Portable Parallel Software Development

Applications

Libraries	Directives	Languages
"Drop-in" Acceleration	Annotate code with compiler hints	Modern language features (unified memory, for_each, lambda)

VISION: MAINSTREAM PARALLEL PROGRAMMING

- Enable more programmers to write parallel software
- Give programmers the choice of language to use
- Embrace and evolve standards in key languages

COMPILE PYTHON FOR PARALLEL ARCHITECTURES

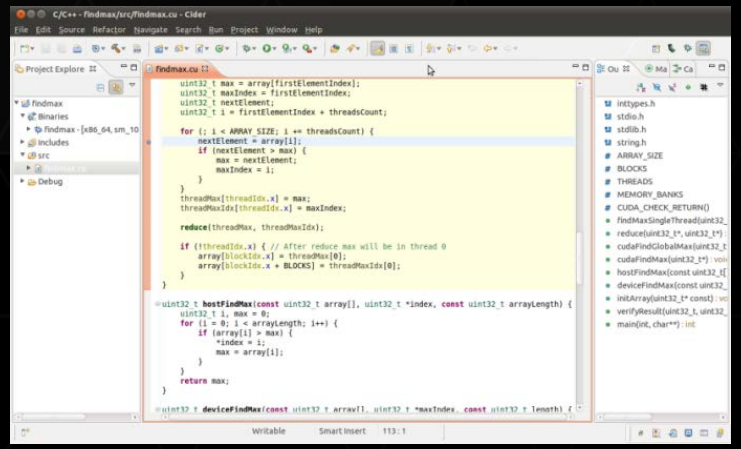
- Anaconda Accelerate from Continuum Analytics
  - NumbaPro array-oriented compiler for Python & NumPy
  - Compile for CPUs or GPUs (uses LLVM + NVIDIA Compiler SDK)

- Fast Development + Fast Execution: Ideal Combination

Free Academic License

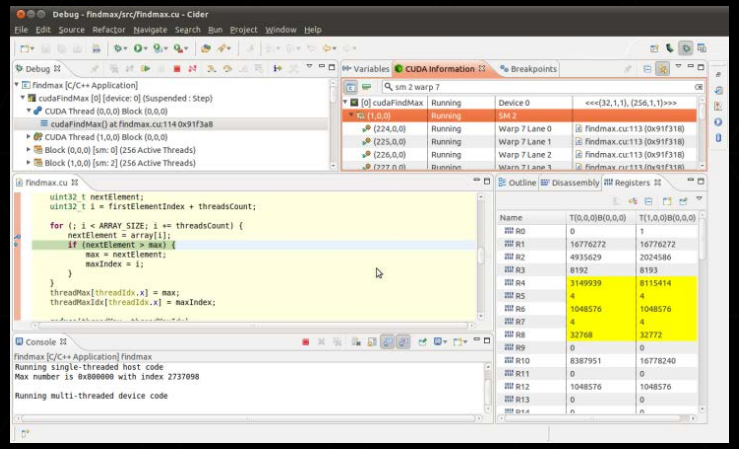
http://continuum.io

NVIDIA® NSIGHT™



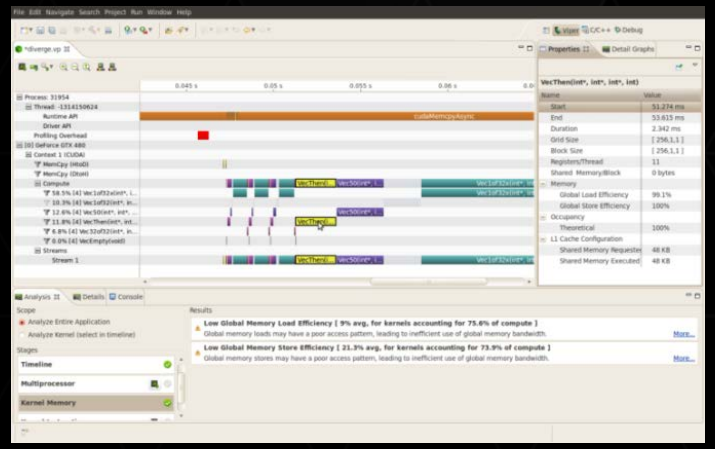
Nsight Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting
- Integrated code samples & documentation
- Cross-compilation for Linux ARM and POWER targets



Nsight Debugger

- Simultaneously debug CPU and GPU code
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging
- Integrated CUDA memory checker



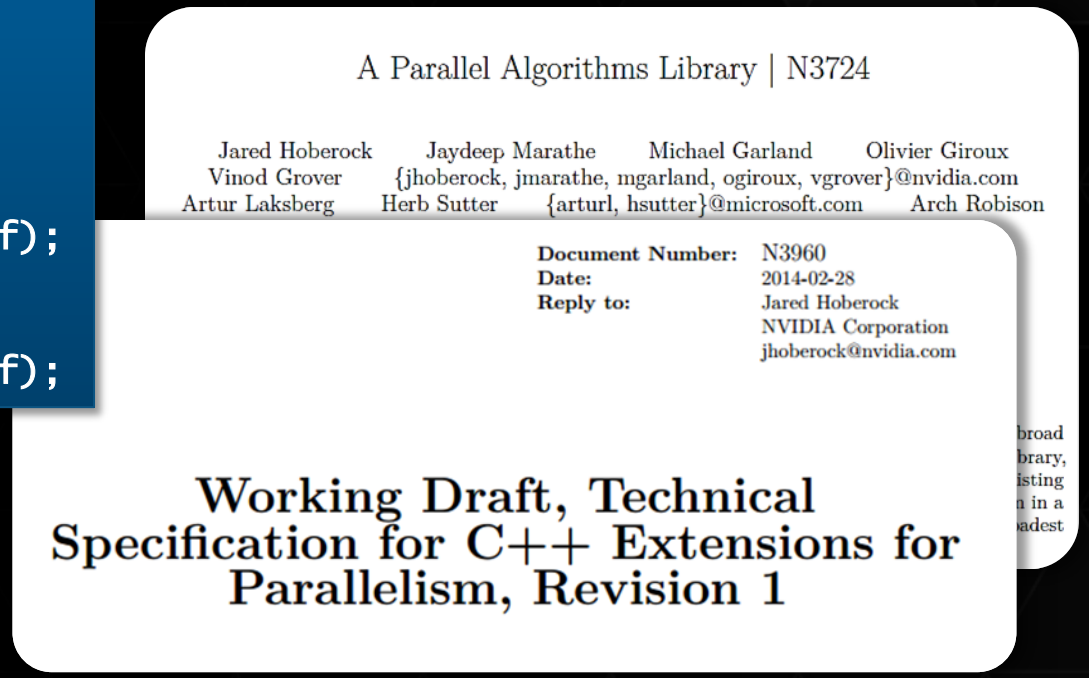
Nsight Profiler

- Quickly identifies performance issues
- Guided expert analysis
- Source line correlation
- PLANNED: Unified CPU and GPU profiling

C++ PARALLEL ALGORITHMS LIBRARY

```
std::vector<int> vec = ...

// previous standard sequential loop
std::for_each(vec.begin(), vec.end(), f);

// explicitly sequential loop
std::for_each(std::seq, vec.begin(), vec.end(), f);

// permitting parallel execution
std::for_each(std::par, vec.begin(), vec.end(), f);
```

- Complete set of parallel primitives: for_each, sort, reduce, scan, etc.
- ISO C++ committee voted unanimously to accept as official tech. specification working draft


A Parallel Algorithms Library | N3724

Jared Hoberock Jaydeep Marathe Michael Garland Olivier Giroux
Vinod Grover {jhoberock, jmarathe, mgarland, ogiroux, vgrover}@nvidia.com
Artur Laksberg Herb Sutter {arturl, hsutter}@microsoft.com Arch Robison

Document Number: N3960
Date: 2014-02-28
Reply to: Jared Hoberock
NVIDIA Corporation
jhoberock@nvidia.com

Working Draft, Technical Specification for C++ Extensions for Parallelism, Revision 1

N3960 Technical Specification Working Draft:
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf
Prototype:
https://github.com/n3554/n3554

CUDA C/C++

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);
```

developer.nvidia.com/cuda-toolkit

UPDATING LEGACY CODE

Libraries

1.

Directives

2.

CUDA C

3.

Accelerating Code with CUDA

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

HELLO WORLD!

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

HETEROGENEOUS COMPUTING

Terminology:

- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)

Host

Device

HETEROGENEOUS COMPUTING

device code

host code

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

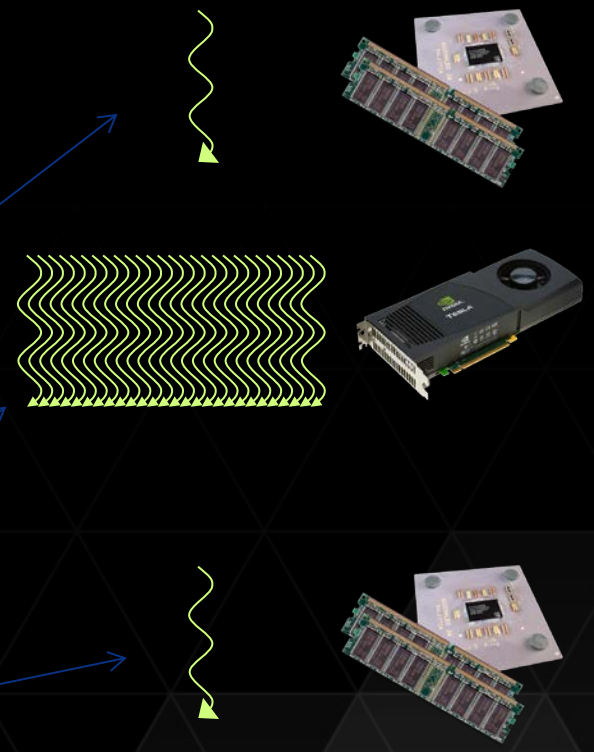
parallel function

serial function

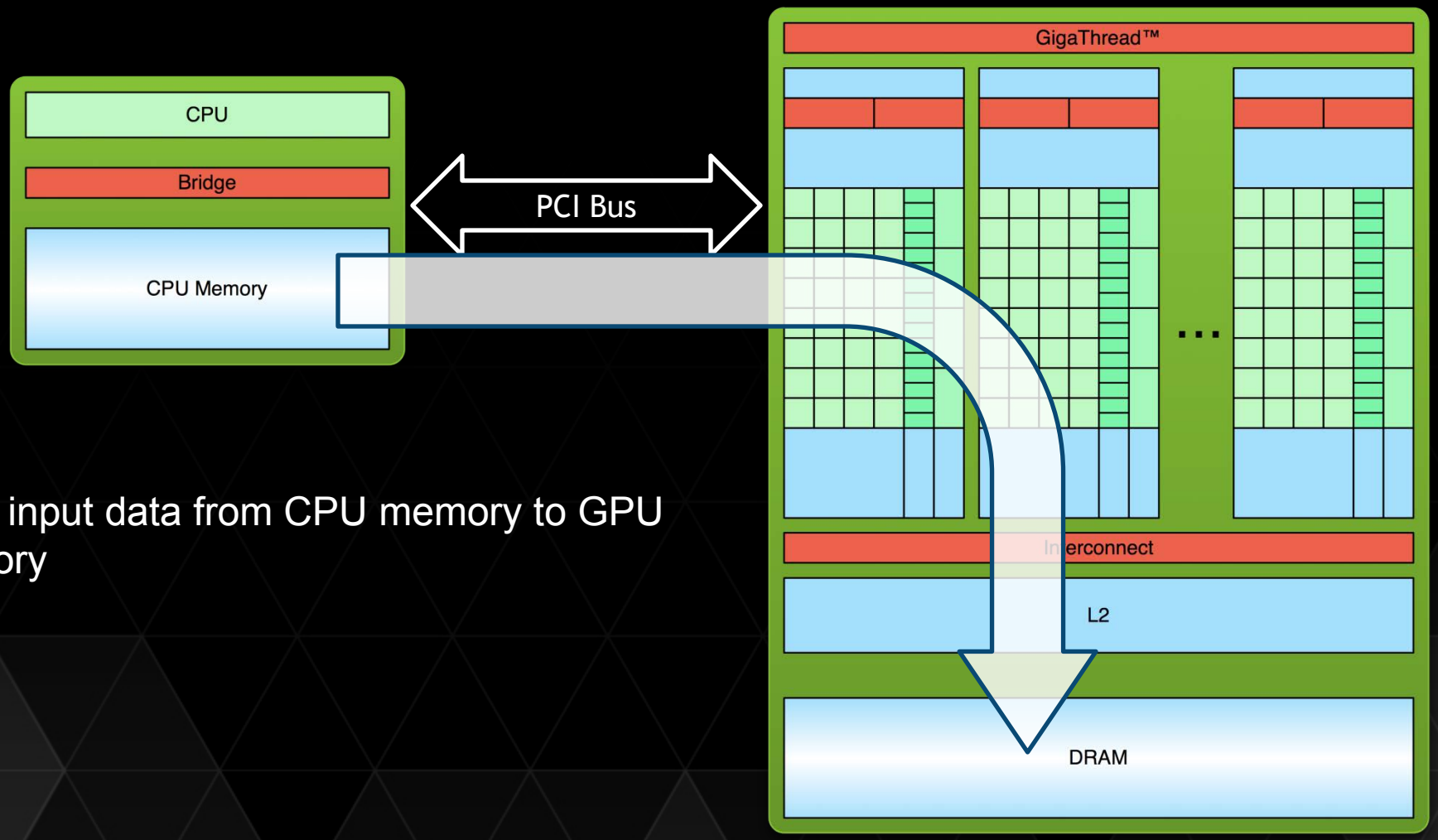
serial code

device function call

serial code

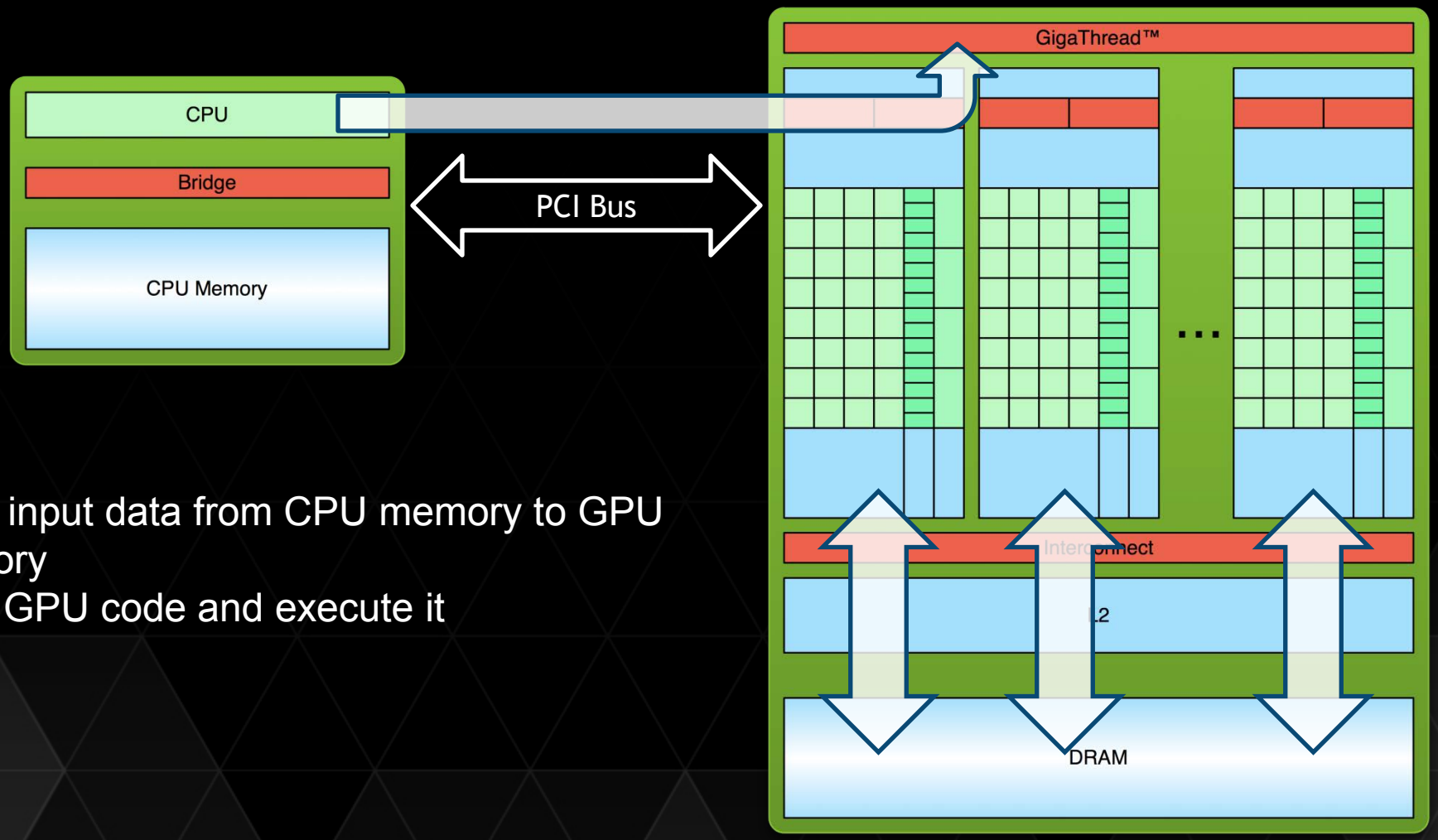


SIMPLE PROCESSING FLOW



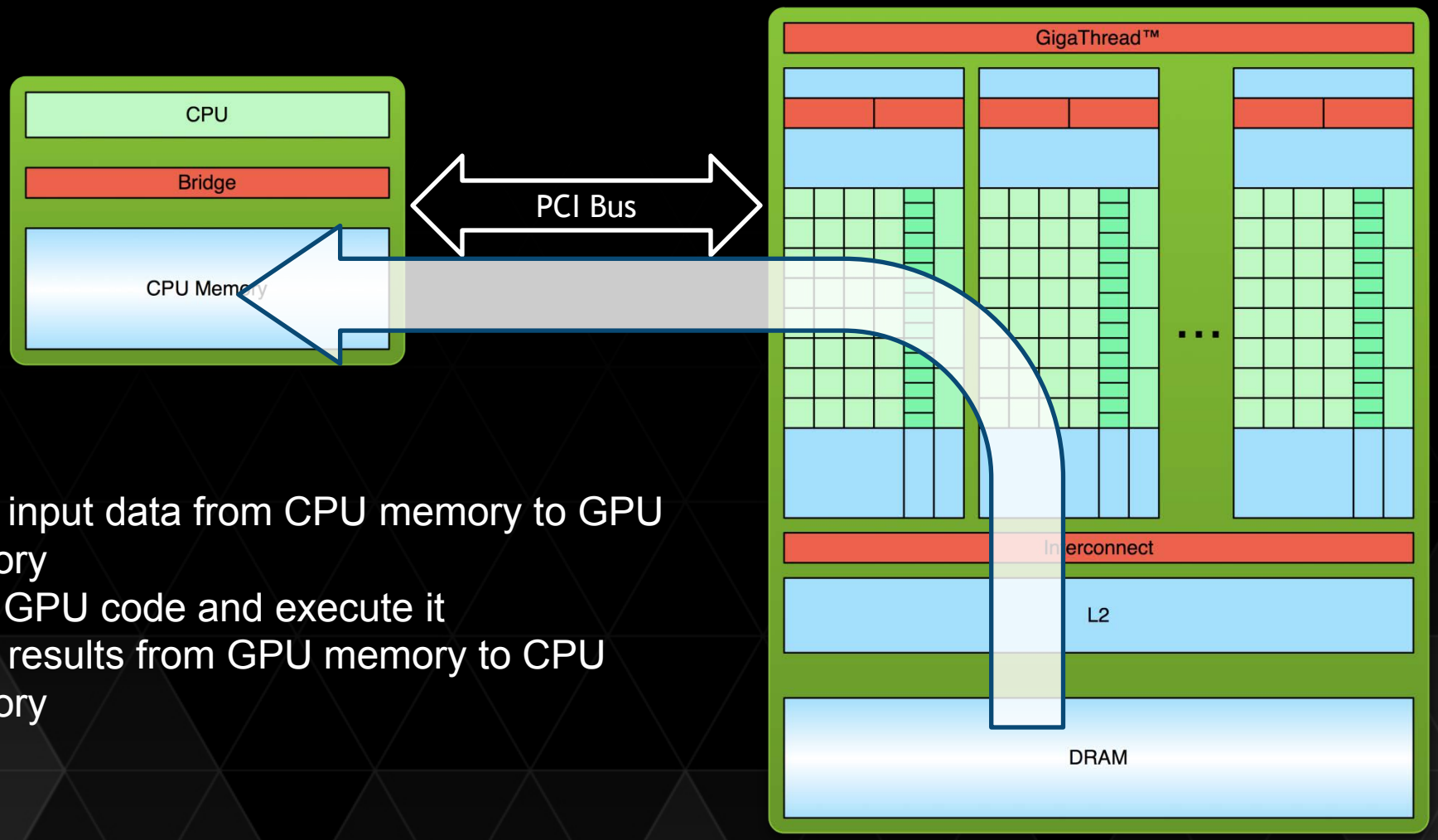
1. Copy input data from CPU memory to GPU memory

SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it

SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
3. Copy results from GPU memory to CPU memory

HELLO WORLD!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code
- `nvcc` separates source code into host and device components

HELLO WORLD! WITH DEVICE CODE

```
__global__ void mykernel(void) {
}


int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

Two new syntactic elements…

HELLO WORLD! WITH DEVICE CODE

```
__global__ void mykernel(void) {

}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler

HELLO WORLD! WITH DEVICE CODE

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

HELLO WORLD! WITH DEVICE CODE

```
__global__ void mykernel(void) {
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

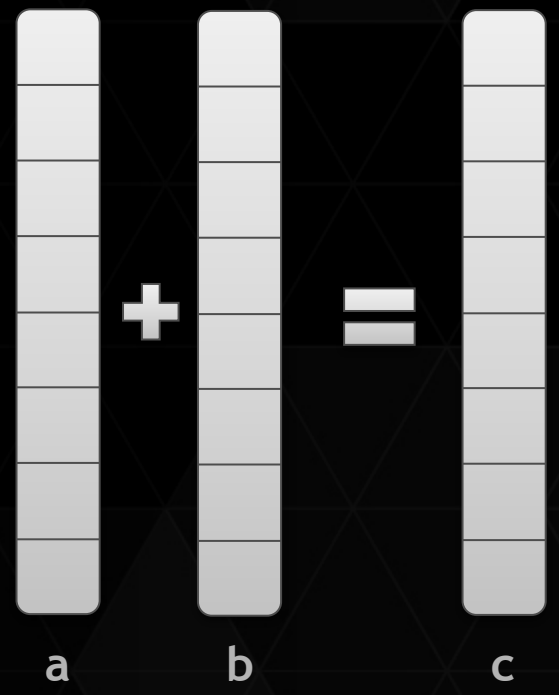
Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
```

- `mykernel()` does nothing, somewhat anticlimactic!

PARALLEL PROGRAMMING IN CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We'll start by adding two integers and build up to vector addition



ADDITION ON THE DEVICE

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

ADDITION ON THE DEVICE

- Note that we use pointers for the variables

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

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

MEMORY MANAGEMENT

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

ADDITION ON THE DEVICE: `ADD()`

- Returning to our `add()` kernel

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

- Let's take a look at main()…

ADDITION ON THE DEVICE: MAIN()

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

ADDITION ON THE DEVICE: MAIN()

```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

REVIEW

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function

RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

MOVING TO PARALLEL

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel
- This first index, is called the BlockIdx…you will see why soon.

VECTOR ADDITION ON THE DEVICE

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

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

- By using `blockIdx.x` to index into the array, each block handles a different element of the array

VECTOR ADDITION ON THE DEVICE

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

- On the device, each block can execute in parallel:

Block 0

```
c[0]  = a[0] + b[0];
```

Block 1

```
c[1]  = a[1] + b[1];
```

Block 2

```
c[2]  = a[2] + b[2];
```

Block 3

```
c[3]  = a[3] + b[3];
```

VECTOR ADDITION ON THE DEVICE: `ADD()`

- Returning to our parallelized `add()` kernel

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

- Let's take a look at main()…

VECTOR ADDITION ON THE DEVICE: MAIN()

```
#define N 512
int main(void) {
        int *a, *b, *c;                 // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

VECTOR ADDITION ON THE DEVICE: MAIN()

```
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU with N blocks
        add<<<N,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

REVIEW

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

CUDA THREADS

- Terminology: a block can be split into parallel threads

- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

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

- We use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`…

VECTOR ADDITION USING THREADS: MAIN()

```
#define N 512
int main(void) {
        int *a, *b, *c;                 // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

VECTOR ADDITION USING THREADS: MAIN()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

COMBINING THREADS AND BLOCKS

CONCEPTS

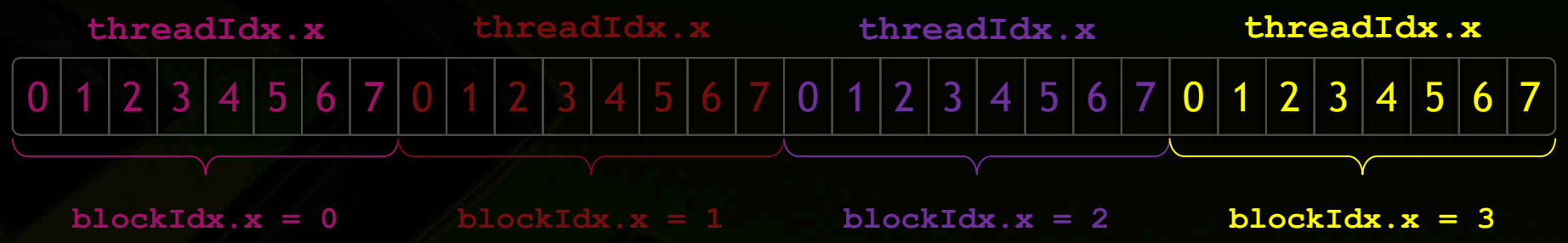
- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

COMBINING BLOCKS *AND* THREADS

- We've seen parallel vector addition using:
  - Several blocks with one thread each
  - One block with several threads
- Let's adapt vector addition to use both *blocks* and *threads*
- Why? We'll come to that…
- First let's discuss data indexing…

INDEXING ARRAYS WITH BLOCKS AND THREADS

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

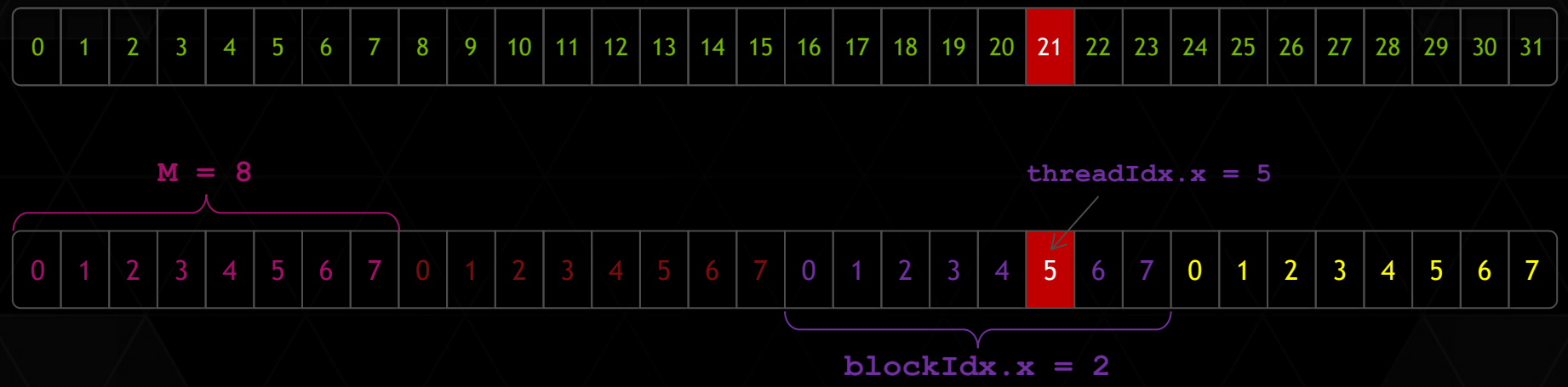


- With M threads per block, a unique index for each thread is given by:

```
int index = blockIdx.x * M + threadIdx.x;
```

INDEXING ARRAYS: EXAMPLE

- Which thread will operate on the red element?




```
int index = blockIdx.x * M + threadIdx.x;
          =     2        * 8 +       5;
          = 21;
```

VECTOR ADDITION WITH BLOCKS AND THREADS

- Use the built-in variable `blockDim.x` for threads per block

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

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

- What changes need to be made in `main()`?

ADDITION WITH BLOCKS AND THREADS:

MAIN()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
        int *a, *b, *c;                 // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

ADDITION WITH BLOCKS AND THREADS:

MAIN()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

HANDLING ARBITRARY VECTOR SIZES

- Typical problems are not even multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

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

- Update the kernel launch:

```
add<<<(N/M + 1),M>>>(d_a, d_b, d_c, N);
```

WHY BOTHER WITH THREADS?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to efficiently:
  - Communicate
  - Synchronize
- To look closer, we need a new example…

COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

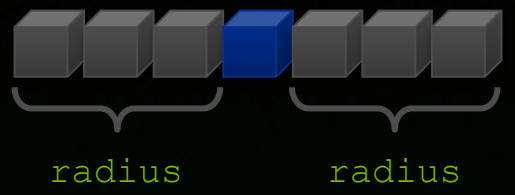
1D STENCIL

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:

IMPLEMENTING WITHIN A BLOCK

- Each thread processes one output element
  - `blockDim.x` elements per block



- Input elements are read several times
  - With radius 3, each input element is read seven times

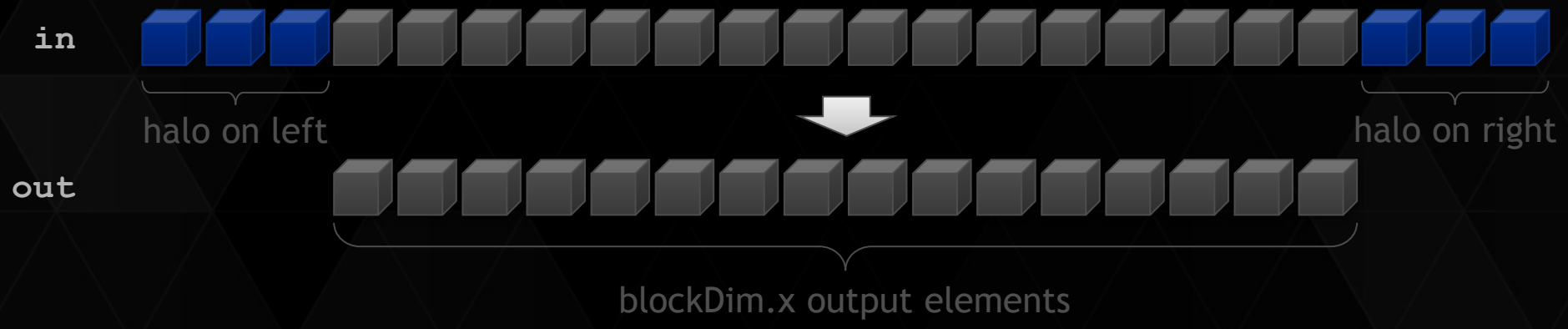
SHARING DATA BETWEEN THREADS

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory
  - By opposition to device memory, referred to as global memory
  - Like a user-managed cache
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

IMPLEMENTING WITH SHARED MEMORY

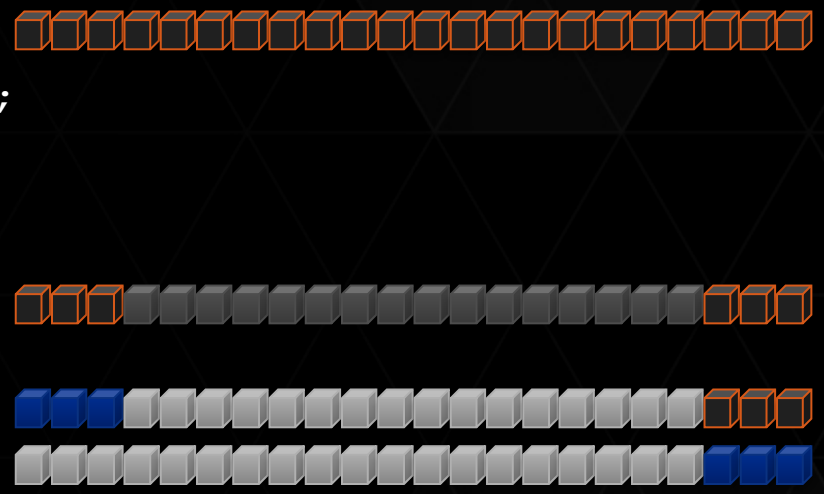
- Cache data in shared memory
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory

- Each block needs a halo of `radius` elements at each boundary



STENCIL KERNEL (1 OF 2)

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
```



STENCIL KERNEL (2 OF 2)

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

DATA RACE!

- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];

if (threadIdx.x < RADIUS) {

   temp[lindex - RADIUS] = in[gindex - RADIUS];

   temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];

}

int result = 0;

for (int offset = -RADIUS ; offset <= RADIUS ; offset++)

   result += temp[lindex + offset];
```

Store at temp[18]

Skipped since threadId.x > RADIUS

Load from temp[19]

__SYNCTHREADS()

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

STENCIL KERNEL

```
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
```

STENCIL KERNEL

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

REVIEW (1 OF 2)

- Launching parallel threads
  - Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block

- Assign elements to threads:

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

REVIEW (2 OF 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

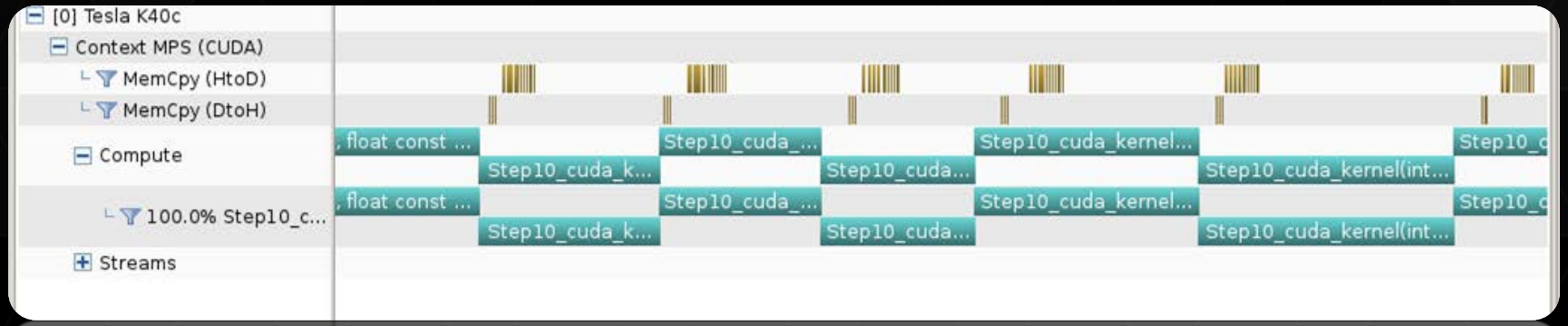
Optimizing CUDA Kernels

OPTIMIZATION TOOLS

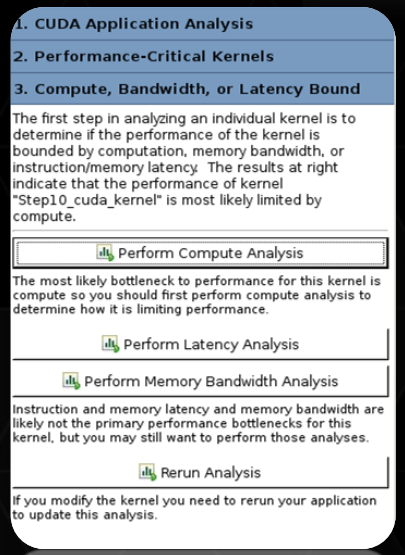
1. NVIDIA’s Visual Profiler (NVVP)
2. NVPROF
3. Nsight Eclipse Edition

NVIDIA'S VISUAL PROFILER

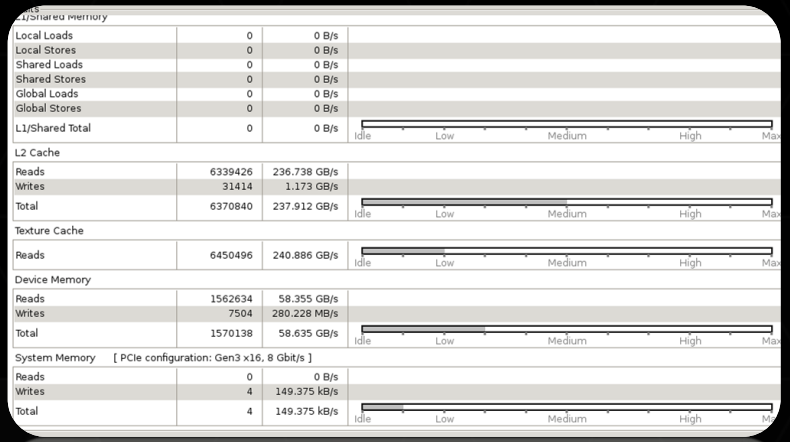
Timeline



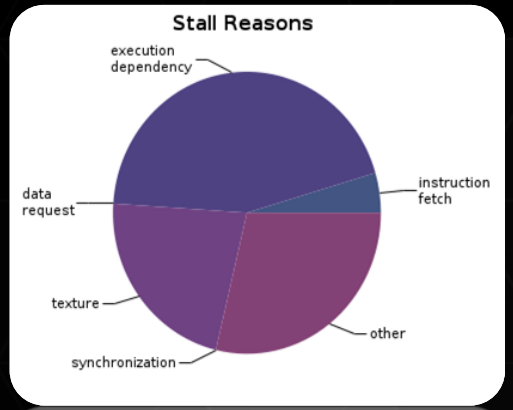
Guided System



Analysis



Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	0	0 B/s	
Global Stores	0	0 B/s	
L1/Shared Total	0	0 B/s	Idle Low Medium High Max
L2 Cache			
Reads	6339426	236.738 GB/s	
Writes	31414	1.173 GB/s	
Total	6370840	237.912 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	6450496	240.886 GB/s	Idle Low Medium High Max
Device Memory			
Reads	1562634	58.355 GB/s	
Writes	7504	280.228 MB/s	
Total	1570138	58.635 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	4	149.375 kB/s	
Total	4	149.375 kB/s	Idle Low Medium High Max



NVPROF

- Command line profiler
  - Collect profiles for NVVP

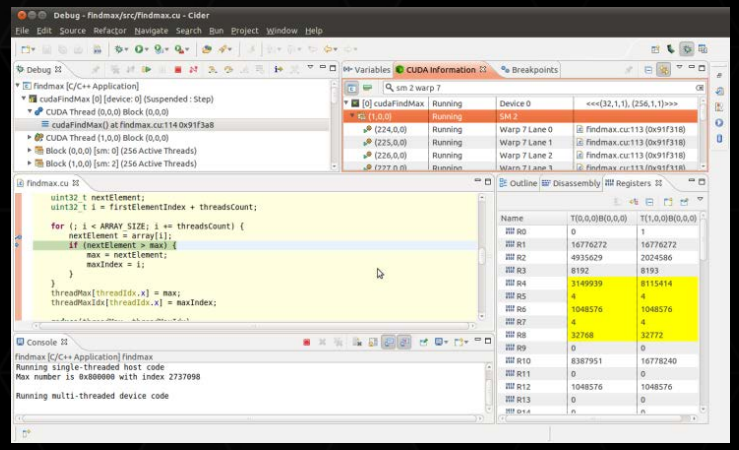
    %> nvprof --analysis-metrics -o profile.out ./HACCmk
  - Collect for MPI processes

    %> mpirun –np 2 nvprof --analysis-metrics -o profile.%p.out ./HACCmk
  - Collect profiles for complex process hierarchies
    - --profile-child-processes, --profile-all-processes
  - Collect key events and metrics

    %> nvprof --metrics flops_sp ./HACCmk
    - --query-metrics --query-events

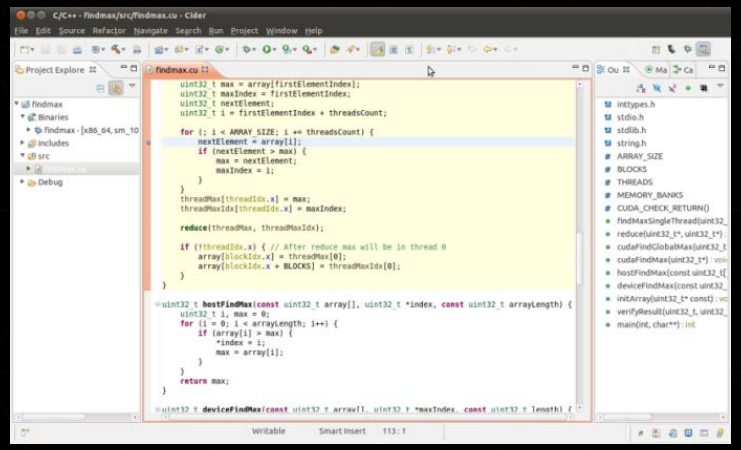
NVIDIA® NSIGHT™, ECLIPSE EDITION

Editor



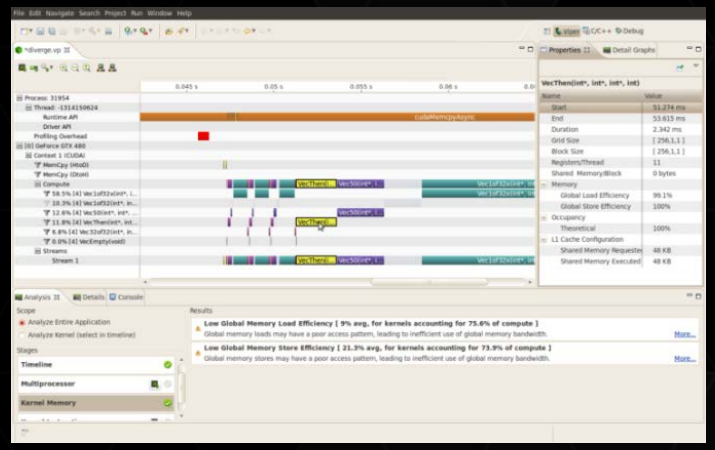
- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Code completion & inline help
- Integrated CUDA samples

Debugger



- Simultaneously debug of CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging

Profiler

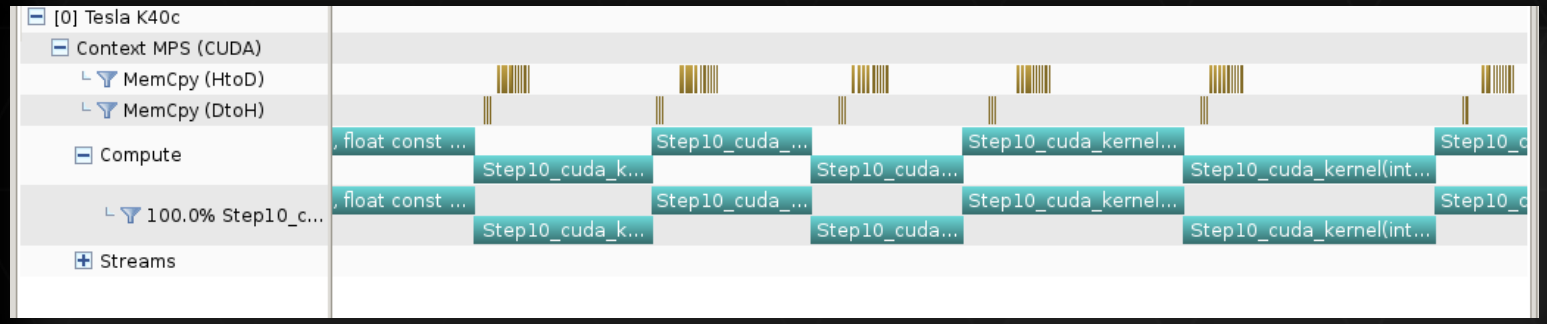


- Quickly identifies performance issues
- Automated analysis
- Source line correlation

Available for Linux and Mac OS

BEFORE OPTIMIZING YOUR KERNELS

- Always use NVVP to determine if the kernel is the limiter
- Kernels may not always be the limiter
  - Memory Allocation
  - Data Transfer
  - Synchronization
  - Host



KEY KERNEL OPTIMIZATIONS

Latency

- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth

- Coalesced Memory Access
- Texture Memory
- Shared memory

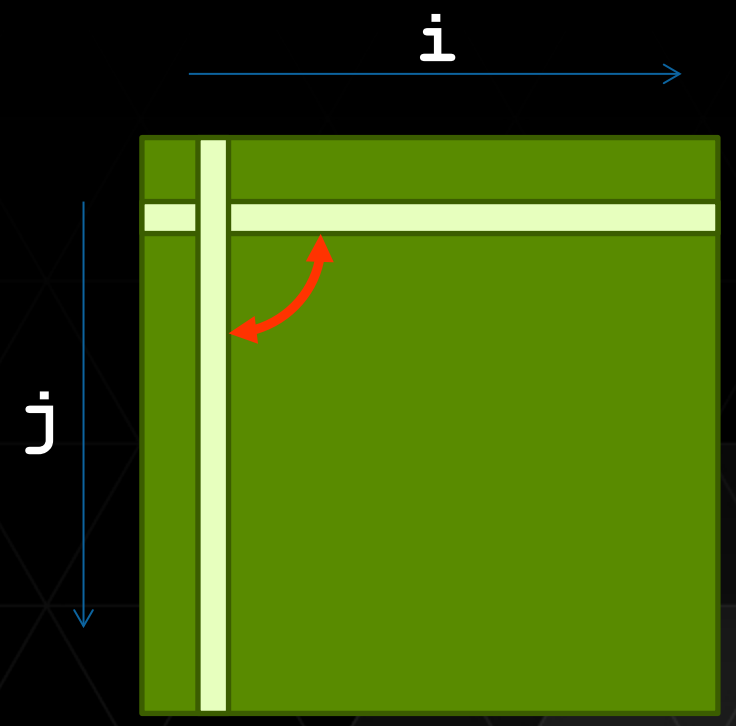
Compute

- Reducing Divergence

CASE STUDY: MATRIX TRANSPOSE

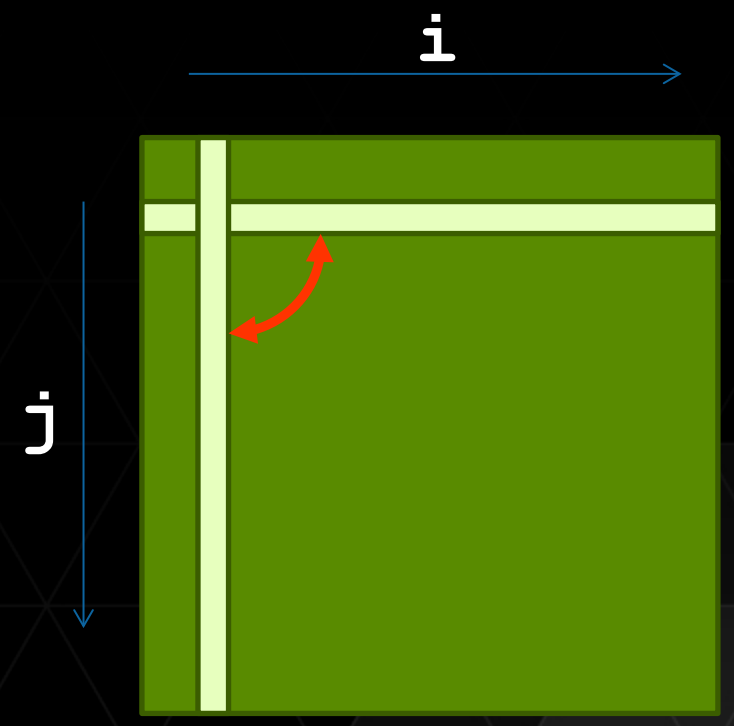
```
void transpose(float in[][], float out[][], int N)
{
  for(int j=0; j < N; j++)
    for(int i=0; i < N; i++)
      out[j][i] = in[i][j];
}
```

- Commonly used in applications
  - BLAS and FFT
- Stresses memory systems
  - Strided reads or writes



2D TO 1D INDEXING

```
void transpose(float in[], float out[], int N)
{
  for(int j=0; j < N; j++)
    for(int i=0; i < N; i++)
      out[j*N+i] = in[i*N+j];
}
```



- This indexing is often used in numerical codes
- We will use this indexing during this presentation

PARALLELIZATION FOR CPU

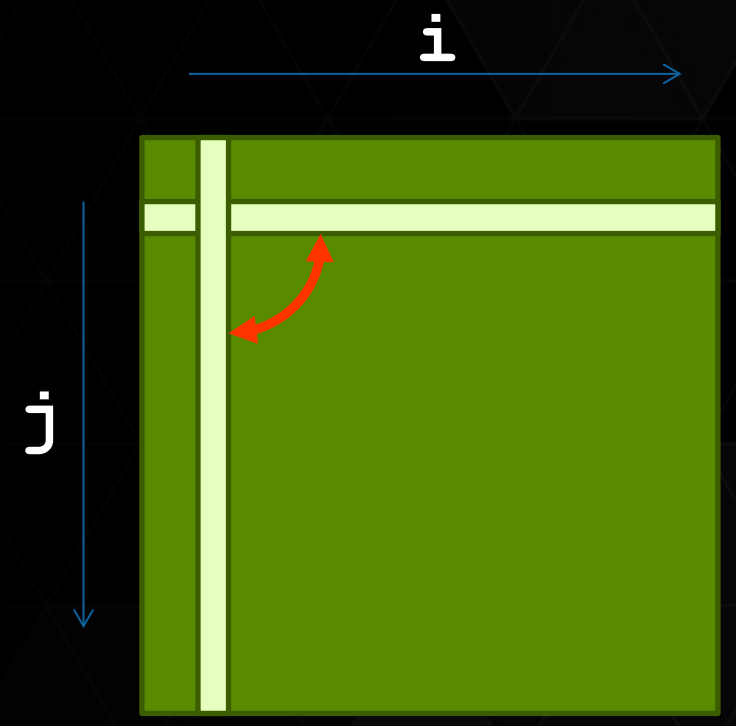
```
void transpose(float in[], float out[], int N)
{
  #pragma omp parallel for
  for(int j=0; j < N; j++)
    #pragma omp parallel for
    for(int i=0; i < N; i++)
      out[j*N+i] = in[i*N+j];
}
```

Kernel	Throughput
CPU+OMP	4.9 GB/s

2X Intel Xeon E5-2650 (16 threads)

INITIAL CUDA IMPLEMENTATION

- Parallelize outer loop



CPU KERNEL

```
void

gpuTranspose_kernel(int rows, int cols, float *in, float *out)

{

  int i, j;

  for ( i=0; i<rows; i++)

    for ( j=0; j<cols; j++)

      out [ j * rows + i ] = in [ i * cols + j ];

}
```

INITIAL GPU KERNEL

```
__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  for ( j=0; j<cols; j++)
    out [ j * rows + i ] = in [ i * cols + j ];
}
```

RESULTS

- Initial implementation slightly slower than dual sockets
- Peak 288 GB/s
- Low percent of peak
  - Why?

Kernel	Throughput	Speedup
CPU+OMP	13.8 GB/s	--
CUDA-1D	9.6 GB/s	0.7x

GPU: K40

NVIDIA.

INITIAL PROFILE

- Always look at occupancy first!
- Each block is scheduled on an SM
  - There are 15 SMs on K40
  - Only 4 blocks!
- Bottleneck
  - Grid size
  - Most of the GPU is idle
- Solution
  - Express more parallelism

Properties

gpuTranspose1_kernel(int, int, float*, float*)

Start	214.366 ms
End	215.231 ms
Duration	865.184 µs
Grid Size	[4,1,1]
Block Size	[256,1,1]
Registers/Thread	17
Shared Memory/Block	0 bytes
Efficiency	
Global Load Efficiency	12.5%
Global Store Efficiency	99.9%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution	100%
Occupancy	
Achieved	12.5%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB
Shared Memory Bank Size	4 bytes

KEY KERNEL OPTIMIZATIONS

Latency

- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth

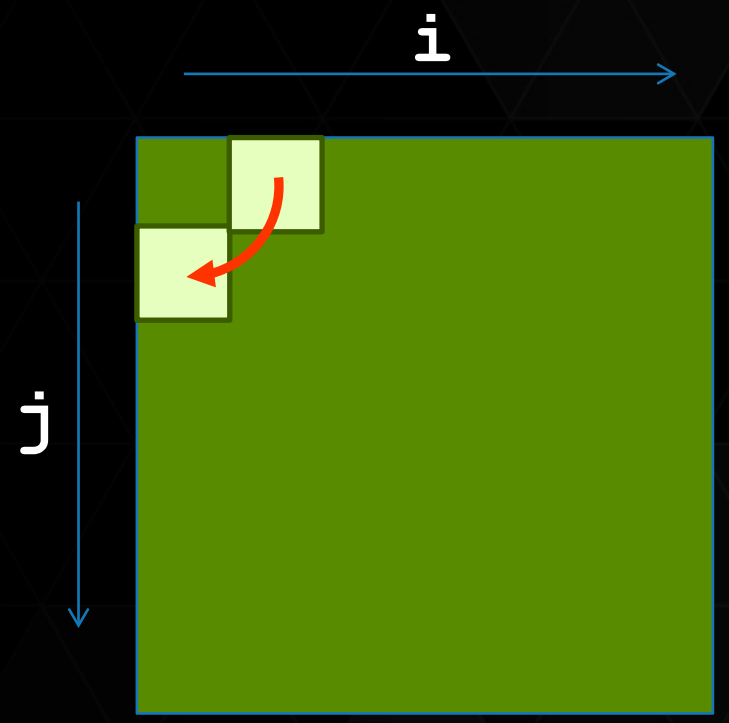
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute

- Reducing Divergence

OPTIMIZATION: EXPRESS MORE PARALLELISM

- The CPU version parallelizes over rows and columns
- Lets do the same on the GPU
  - Replace columns loop with an index calculation
  - Change launch configuration to 2D
    - blockSize = 32x32



1D SOLUTION

```
__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  for ( j=0; j<cols; j++)
    out [ j * rows + i ] = in [ i * cols + j ];
}
```

2D SOLUTION

```
__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  out [ j * rows + i ] = in [ i * cols + j ];
}
```

RESULTS

- We are now at a 3x speedup over 2 sockets
- But how are we doing overall?
  - Peak for K40 is 288 GB/s
  - ~14% of peak
- Why is bandwidth utilization low?
  - Back to NVVP

Kernel	Throughput	Speedup
CPU+OMP	13.9 GB/s	--
GPU-1D	9.7 GB/s	0.7x
GPU-2D	41.6 GB/s	3.0x

2D PROFILE

- Occupancy is now much better
- All SMs have work
- Global load efficiency is low
- Bottleneck
  - Uncoalesced loads

Properties

gpuTranspose2_kernel(int, int, float*, float*)

Start	261.05 ms
End	261.25 ms
Duration	199.744 µs
Grid Size	[32,32,1]
Block Size	[32,32,1]
Registers/Thread	8
Shared Memory/Block	0 bytes
Efficiency	
Global Load Efficiency	12.5%
Global Store Efficiency	100%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution	100%
Occupancy	
Achieved	77.4%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB
Shared Memory Bank Size	4 bytes

KEY KERNEL OPTIMIZATIONS

Latency

- Expressing Concurrency
- Instruction Level Parallelism

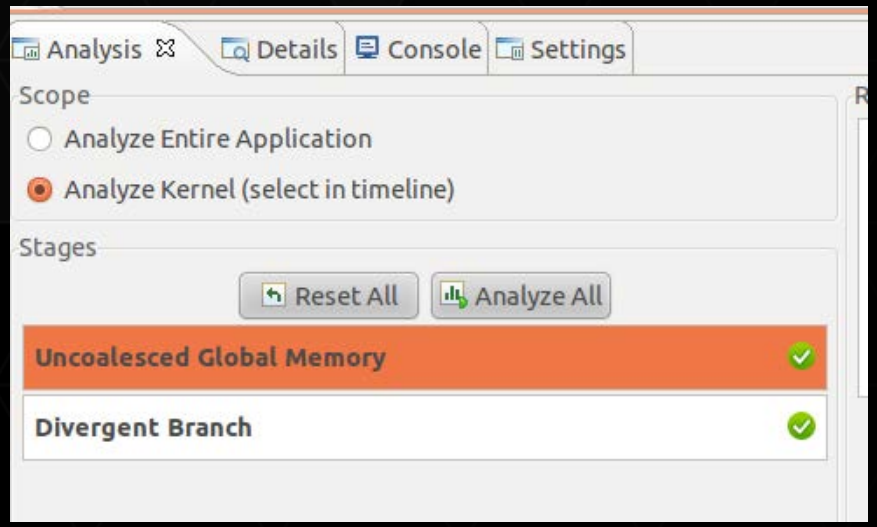
Bandwidth

- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute

- Reducing Divergence

USE NVVP TO FIND COALESCING PROBLEMS



- Compile with -lineinfo

```
__global__ void gpuTranspose_kernel(int rows, int co
{
  int i; int j;

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  out[i*cols + j] = in[j*cols + i];
}
```

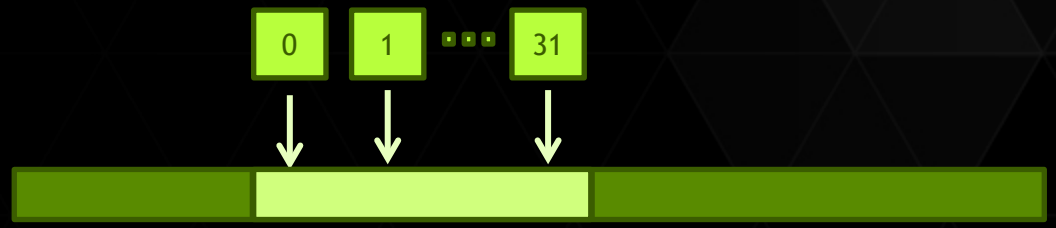
Uncoalesced Global Memory Accesses

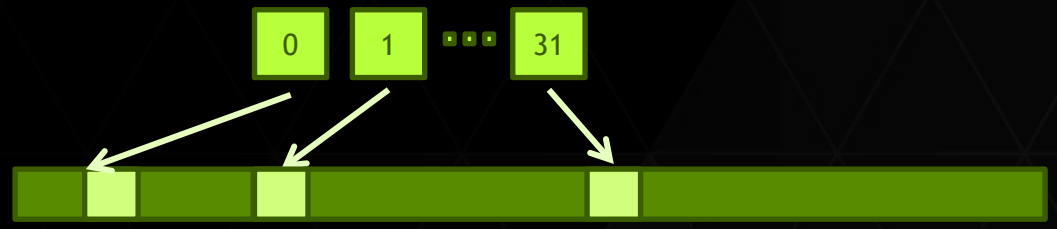
Global memory loads and stores have poor access patterns, leading to inefficient use of global memory bandwidth. Select from the table below to see the source code which generates the inefficient global loads and stores.

Location	Description
File: main.c	
Line: 41	Global Store L2 Transactions/Access = 32.0 [1048576 L2 transactions for 32768 total executions]

WHAT IS AN UNCOALESCED GLOBAL LOAD?

- Global memory access happens in transactions of 32 or 128 bytes
- *Coalesced* access:
  - A group of 32 contiguous threads ("warp") accessing adjacent words
  - Few transactions and high utilization
- *Uncoalesced* access:
  - A warp of 32 threads accessing scattered words
  - Many transactions and low utilization





KEY KERNEL OPTIMIZATIONS

Latency

- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth

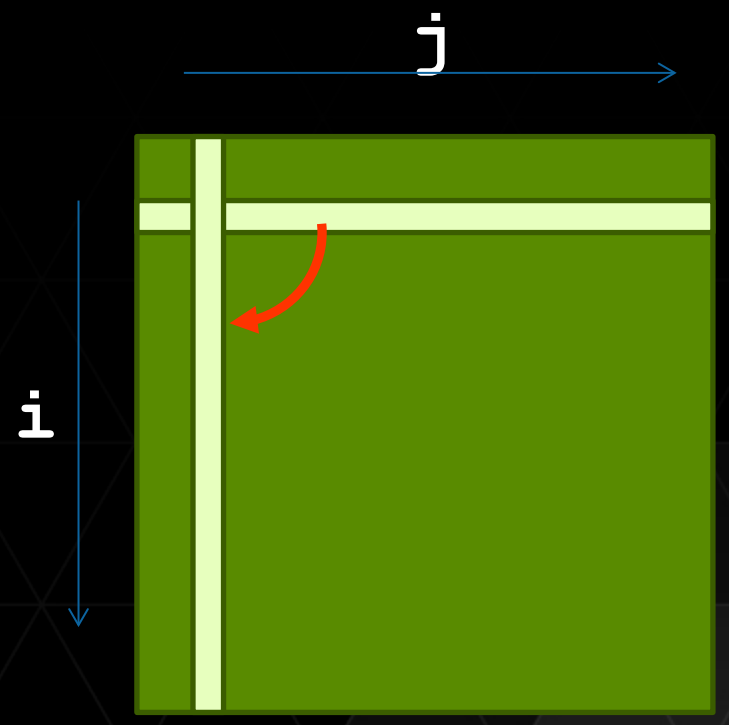
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute

- Reducing Divergence

TRANSPOSE MEMORY COALESCING

- **Either the read or the write will be uncoleased**
- **Two options**
  1. **Use texture for loads**
  2. **Modify indexing to eliminate uncoleased accesses**
     - Fastest changing dimension should be threadIdx.x
     - Need to stage through shared memory



TEXTURE CACHE

- Texture has an on-chip cache (Think L1)
- Two ways to use it (SM_35 or greater)
  - Implicit
    - Add const __restrict__
  - Explict
    - Use the __ldg(ptr) intrinsic

TEXTURE CACHE SOLUTION

```
__global__ void
gpuTranspose_kernel(int rows, int cols,
                    float const * __restrict__ in, float *out)
{
  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  out [ j * rows + i ] = in [ i * cols + j ];
}
```

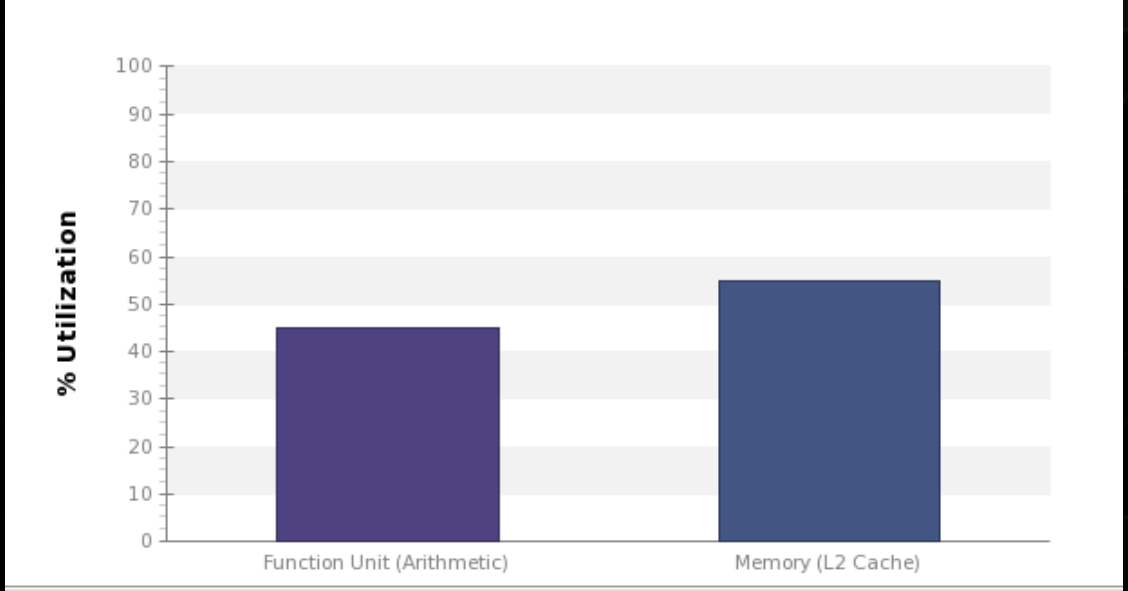
RESULTS

- Now at 48% of peak with very little effort
- About a 10x speedup over 2 sockets

Kernel	Throughput	Speedup
CPU+OMP	13.9 GB/s	--
GPU-1D	9.7 GB/s	0.7x
GPU-2D	41.6 GB/s	3.0x
GPU-Tex	137.4 GB/s	9.9x

CAN WE DO BETTER?

- Not compute bound
- Not memory bound
- Assume latency bound



- What if we did the shared memory approach?

KEY KERNEL OPTIMIZATIONS

Latency

- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth

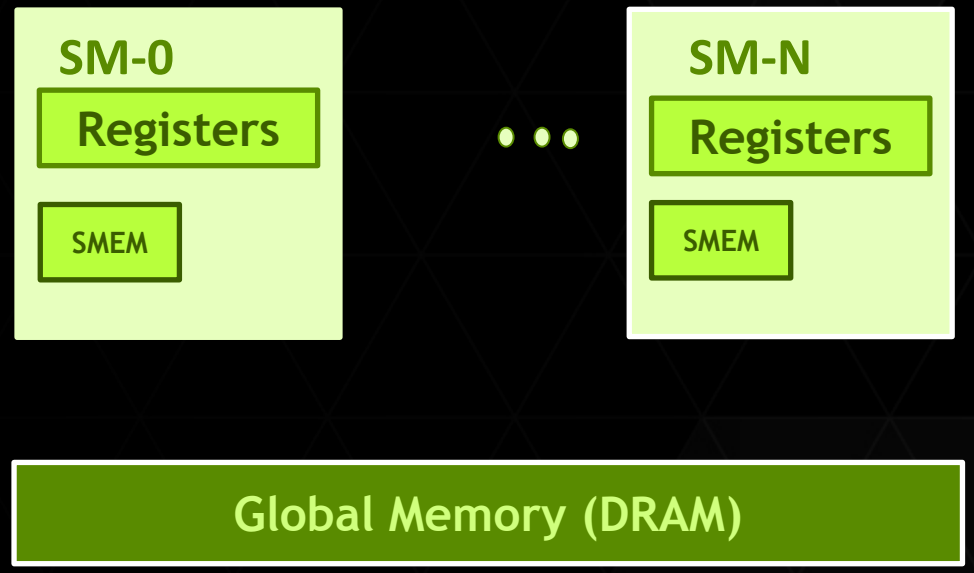
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute

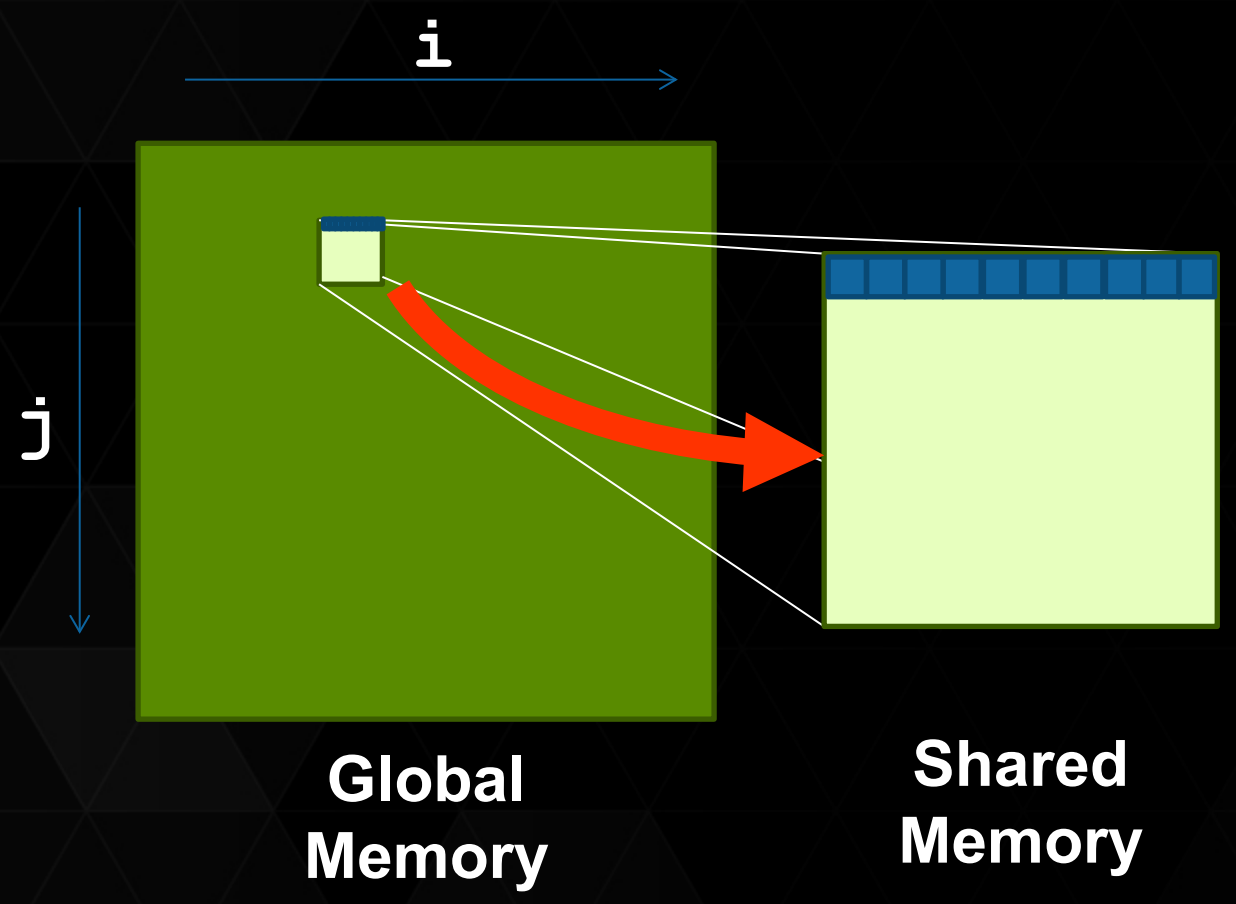
- Reducing Divergence

SHARED MEMORY

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion

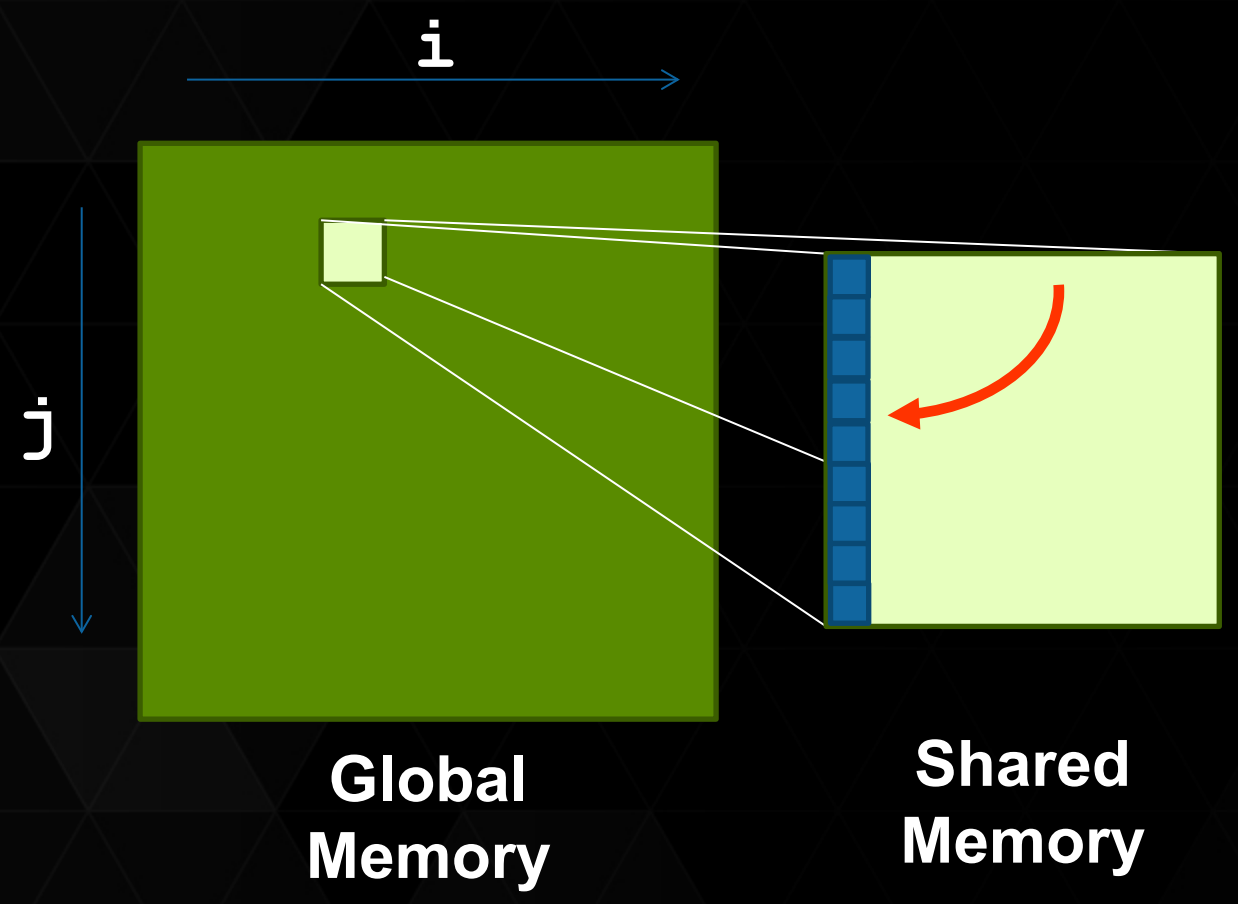


TRANSPOSING WITH SHARED MEMORY



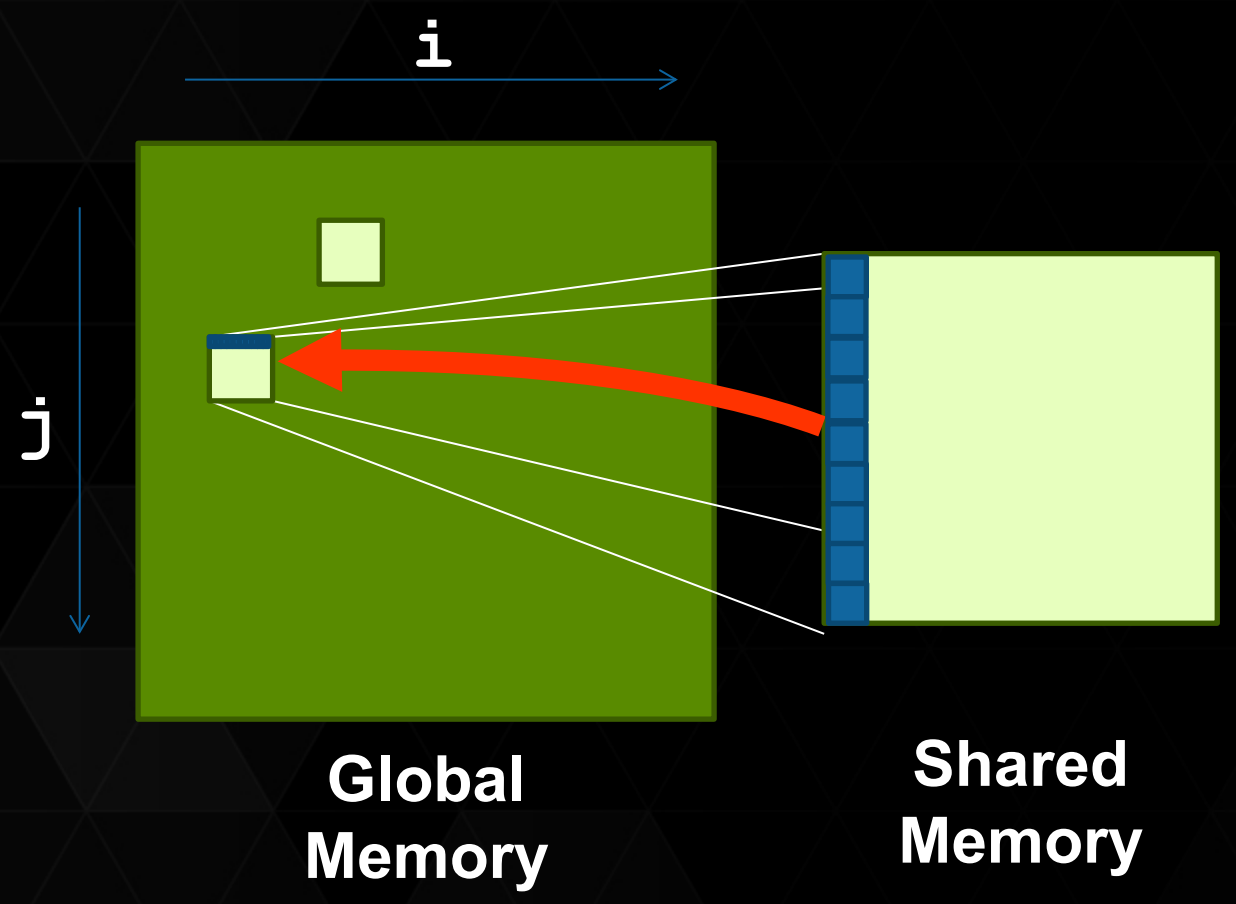
- Read block coalesced into shared memory

TRANSPOSING WITH SHARED MEMORY



- Read block coalesced into shared memory
- Transpose shared memory indices

TRANSPOSING WITH SHARED MEMORY



- Read block_ij coalesced into shared memory
- Transpose shared memory indices
- Write transposed block to global memory

ALLOCATE SHARED MEMORY

```
#define TILE_DIM 32

__global__ void

gpuTranspose_kernel(int rows, int cols, float *in, float *out)

{

  int i, j;

  __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];

  ...

}
```

READ & WRITE COALESCED

```
#define TILE_DIM 32

__global__ void

gpuTranspose_kernel(int rows, int cols, float *in, float *out)

{

  int i, j;
  __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];

  ...
  ... = in [ j * cols + i ];
  ...

  out[ j * rows + i ] = ...
}
```

STAGE THROUGH SHARED MEMORY

```
#define TILE_DIM 32

__global__ void

gpuTranspose_kernel(int rows, int cols, float *in, float *out)

{

  int i, j;
  __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];
  ...
  tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];
  ...

  out[ j * rows + i ] = tile[ threadIdx.y ] [ threadIdx.x ];
}
```

TRANSPOSE SHARED MEMORY

```
#define TILE_DIM 32

_global__ void

gpuTranspose_kernel(int rows, int cols, float *in, float *out)

{

  int i, j;
  __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];


  ...
  tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];
  ...

  out[ j * rows + i ] = tile[ threadIdx.x ] [ threadIdx.y ];
}
```

TRANSPOSE BLOCK INDICES

```
#define TILE_DIM 32

_global__ void

gpuTranspose_kernel(int rows, int cols, float *in, float *out)

{

  int i, j;
  __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];

  i = blockIdx.y * blockDim.y + threadIdx.x;
  j = blockIdx.x * blockDim.x + threadIdx.y;
  out[ j * rows + i ] = tile[ threadIdx.x ] [ threadIdx.y ];

}
```

SYNCHRONIZE

```
#define TILE_DIM 32

__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
  int i, j;
  __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];
  __syncthreads();

  i = blockIdx.y * blockDim.y + threadIdx.x;
  j = blockIdx.x * blockDim.x + threadIdx.y;
  out[ j * rows + i ] = tile[ threadIdx.x ] [ threadIdx.y ];
}
```

RESULTS

- Performance is worse
- Shared memory should be faster than texture
  - What went wrong?

Kernel	Throughput	Speedup
CPU+OMP	13.9 GB/s	--
GPU-1D	9.7 GB/s	0.7x
GPU-2D	41.6 GB/s	3.0x
GPU-Tex	137.4 GB/s	9.9x
GPU-Shared	116.0 GB/s	8.3x

SHARED MEMORY PROFILE

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	524288	1,746.83 GB/s
Shared Stores	32768	109.18 GB/s
Global Loads	0	0 B/s
Global Stores	32768	54.59 GB/s
L1/Shared Total	589824	1,910.6 GB/s

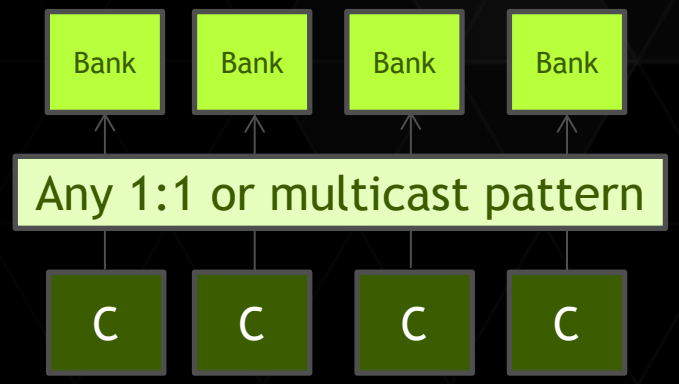
- Shared Memory is achieving good bandwidth
- However, replay overhead is high

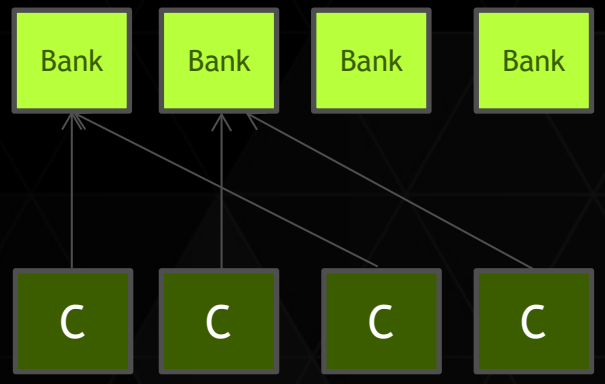
Shared Memory Replay Overhead
0.5

- Bottleneck: Bank Conflicts

SHARED MEMORY ORGANIZATION

- Organized in 32 independent banks
- Optimal access: all words from different banks
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize

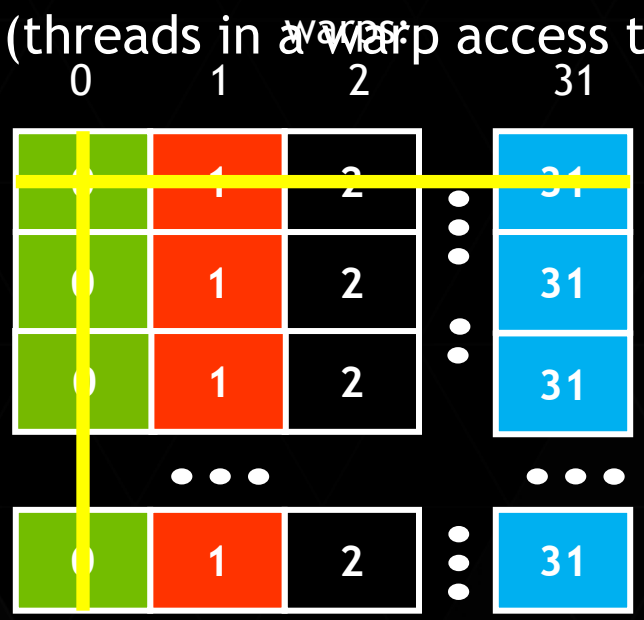




SHARED MEMORY: AVOIDING BANK CONFLICTS

- Example: 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)

Bank 0
Bank 1
…
Bank 31



Accesses along row produces 0 bank conflicts

Accesses along column produces 32 bank conflicts

SHARED MEMORY: AVOIDING BANK CONFLICTS

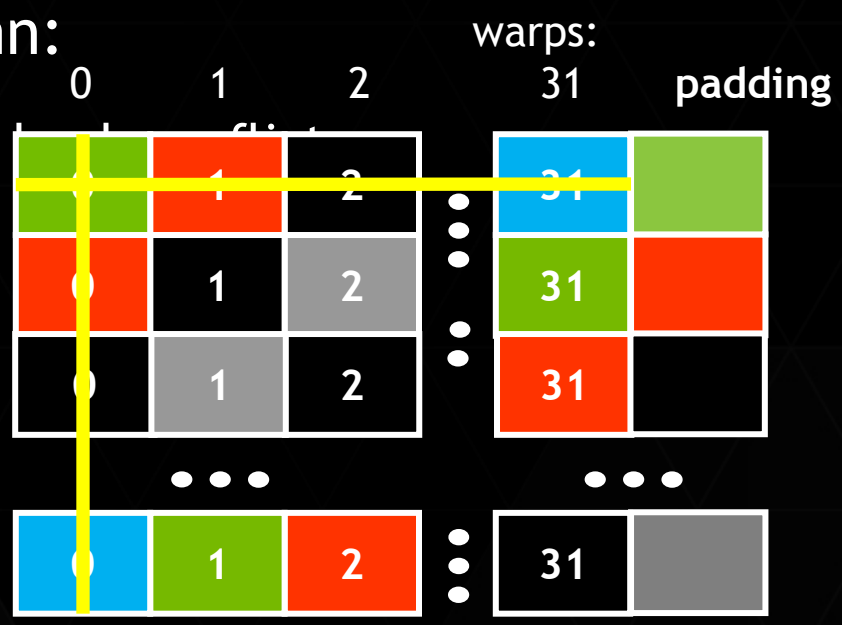
- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts

Bank 0

Bank 1

...

Bank 31



Accesses along row produces 0 bank conflicts

Accesses along column produces 0 bank conflicts

SOLUTION – WITHOUT BANK CONFLICTS

```
#define TILE_DIM 32

__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
  int i, j;
  __shared__ float tile [ TILE_DIM ] [ TILE_DIM + 1 ];

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];
  __syncthreads();

  i = blockIdx.y * blockDim.y + threadIdx.x;
  j = blockIdx.x * blockDim.x + threadIdx.y;
  out[ j * rows + i ] = tile[ threadIdx.x ] [ threadIdx.y ];
}
```

RESULTS

- Performance is better
- Bottleneck:
  - Kepler requires 100+ lines in flight per SM to saturate DRAM
  - 1 line-in-flight per warp @ 100% occupancy = 64 lines in flight
- Solution:
  - Process multiple elements
    - Instruction-level parallelism
    - More lines-in-flight
    - Less __syncthreads overhead
    - Amortize cost of indexing and thread launch

Kernel	Throughput	Speedup
CPU+OMP	13.9 GB/s	--
GPU-1D	9.7 GB/s	0.7x
GPU-2D	41.6 GB/s	3.0x
GPU-Tex	137.4 GB/s	9.9x
GPU-Shared	116.0 GB/s	8.3x
GPU-Padded	150.6 GB/s	10.8x

KEY KERNEL OPTIMIZATIONS

Latency

- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth

- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute

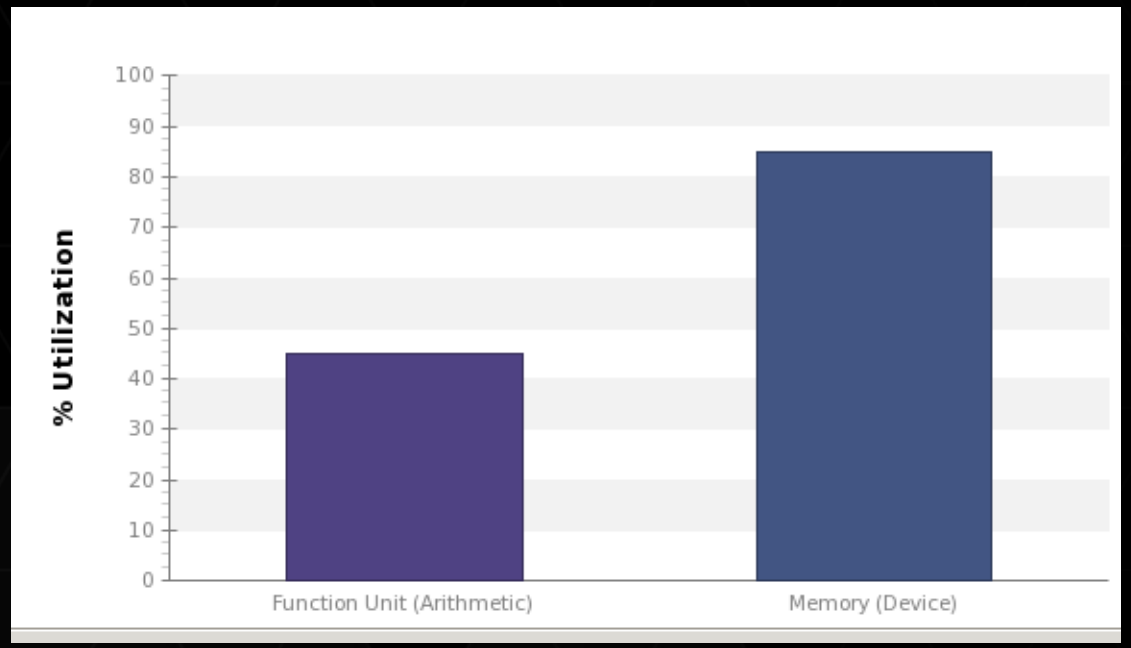
- Reducing Divergence

PROCESS MULTIPLE ELEMENTS PER THREAD

- Increases ILP
  - Reduces Exposed Latency
  - Allows compiler to schedule instructions more efficiently

FINAL PROFILE

- 82% of peak bandwidth
- Approaching memcpy speeds



Kernel	Throughput	Speedup
CPU+OMP	13.9 GB/s	--
GPU-1D	9.7 GB/s	0.7x
GPU-2D	41.6 GB/s	3.0x
GPU-Tex	137.4 GB/s	9.9x
GPU-Shared	116.0 GB/s	8.3x
GPU-Padded	150.6 GB/s	10.8x
GPU-Multi	238.4 GB/s	17.2x

FINAL RESULTS

- Use NVVP to identify bottlenecks
- Use optimization techniques to eliminate bottlenecks
- Refer to GTC archives for complete optimization techniques

Kernel	Throughput	Speedup
CPU+OMP	13.9 GB/s	--
GPU-1D	9.7 GB/s	0.7x
GPU-2D	41.6 GB/s	3.0x
GPU-Tex	137.4 GB/s	9.9x
GPU-Shared	116.0 GB/s	8.3x
GPU-Padded	150.6 GB/s	10.8x
GPU-Multi	238.4 GB/s	17.2x

- www.gputechconf.com/gtcnew/on-demand-gtc.php
- Search “GPU Performance Analysis and Optimization”

KEY KERNEL OPTIMIZATIONS

Latency

- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth

- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute

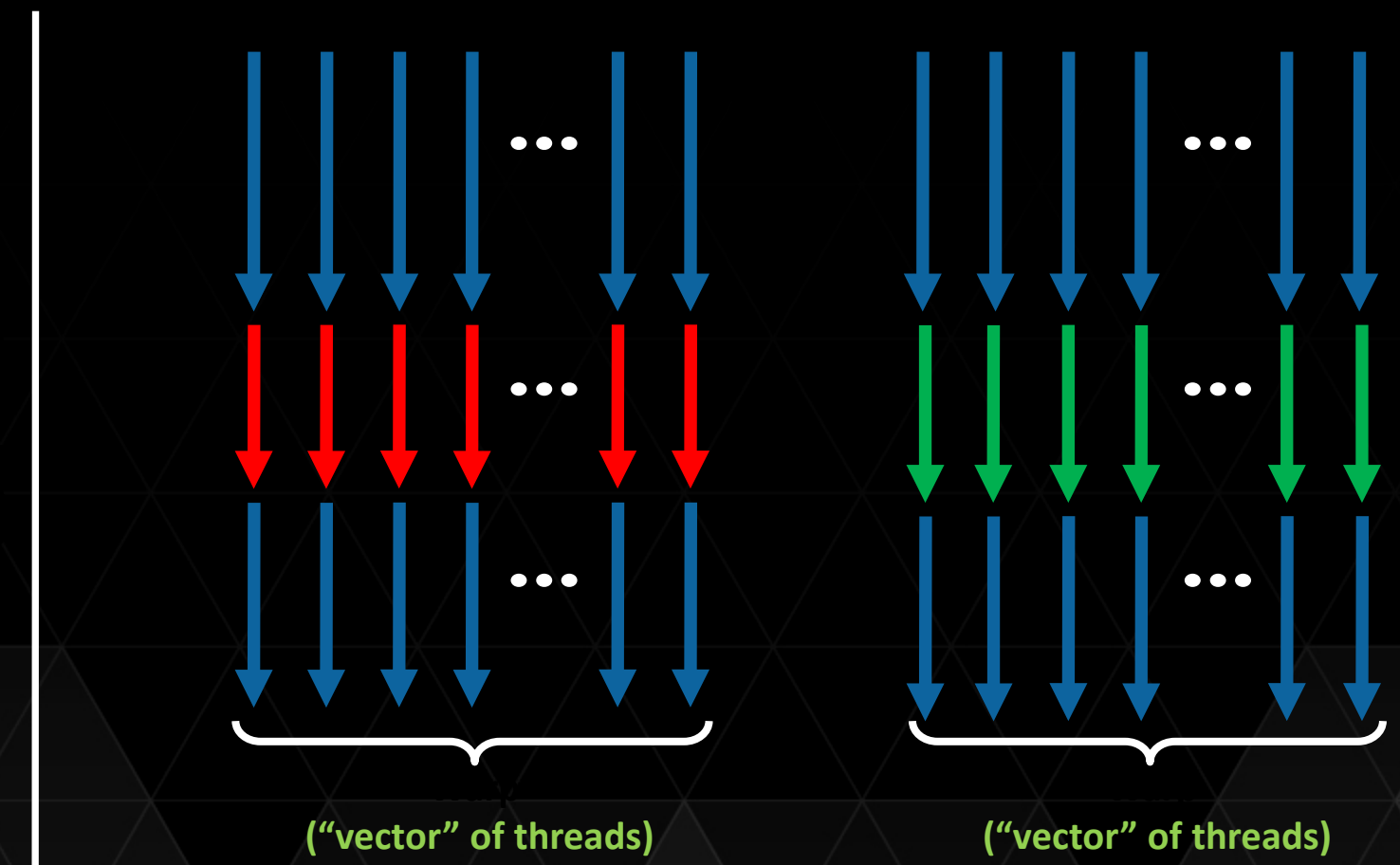
- Reducing Divergence

CONTROL FLOW

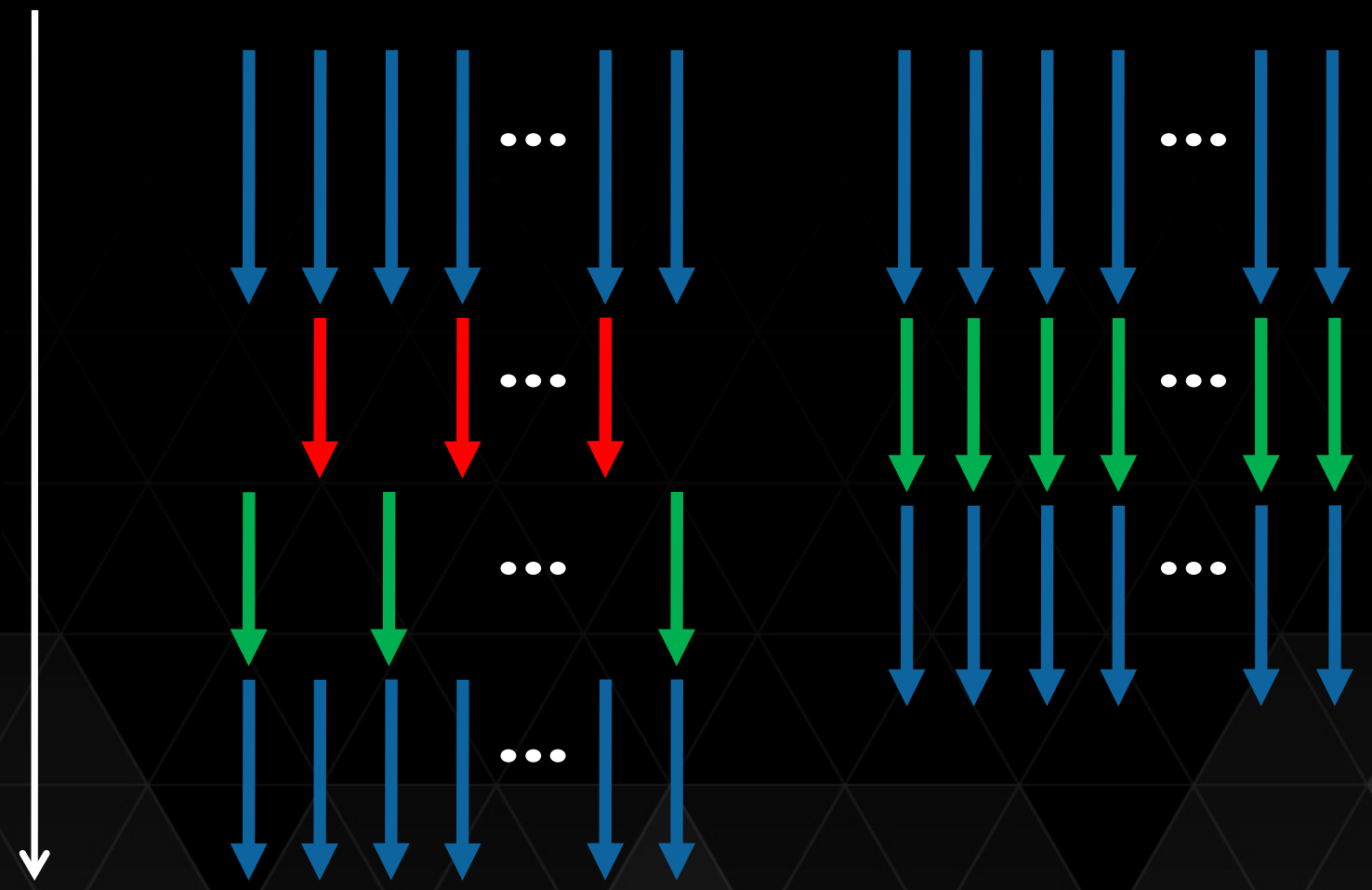
- **Single-Instruction Multiple-Threads (SIMT) model**
  - A single instruction is issued for a warp (thread-vector) at a time
- **SIMT compared to SIMD:**
  - SIMD requires vector code in each thread
  - SIMT allows you to write scalar code per thread
    - Vectorization is handled by hardware
- **Note:**
  - All contemporary processors (CPUs and GPUs) are built by aggregating vector processing units
  - Vectorization is needed to get performance on CPUs and GPUs

CONTROL FLOW

EXECUTION WITHIN WARPS IS COHERENT



EXECUTION DIVERGES WITHIN A WARP



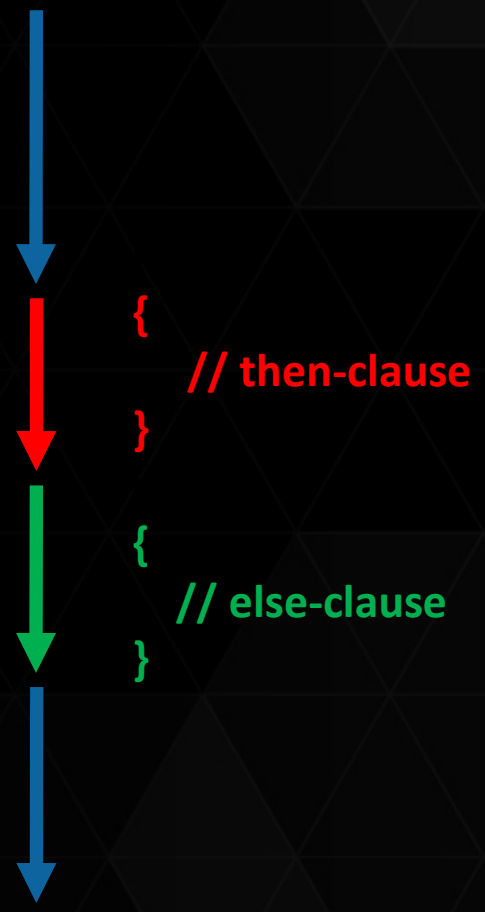
CONTROL FLOW

Don't freak out about this.

Some `if - then` is OK.

Example -

Checking edge cases when you have tiled input…



MATH OPTIMIZATION TIPS

Use “intrinsics” for math operations if you can.

A couple bits lower precision, but much faster.

```
__sin()    __cos()    __exp()
```

Use double-precision when you mean to.

```
float a = b + 2.5f
```

GRAND OPTIMIZATION SUMMARY

APOD = Assess Parallelize Optimize Deploy

- Use profiler
- Deploy early
- Optimizing is increasing effort. Complicates code.

Measure & Improve Memory BW

- Express sufficient parallelism
- Coalesce global memory accesses
- Reduce threads in block when using `synchthreads`

Minimize Thread Divergence

- Avoid branchy code
- Don’t freak out over this

Consider Fast Math

- Intrinsics

Stream & Asynchronous Copies

- Overlap memory transfers and computation

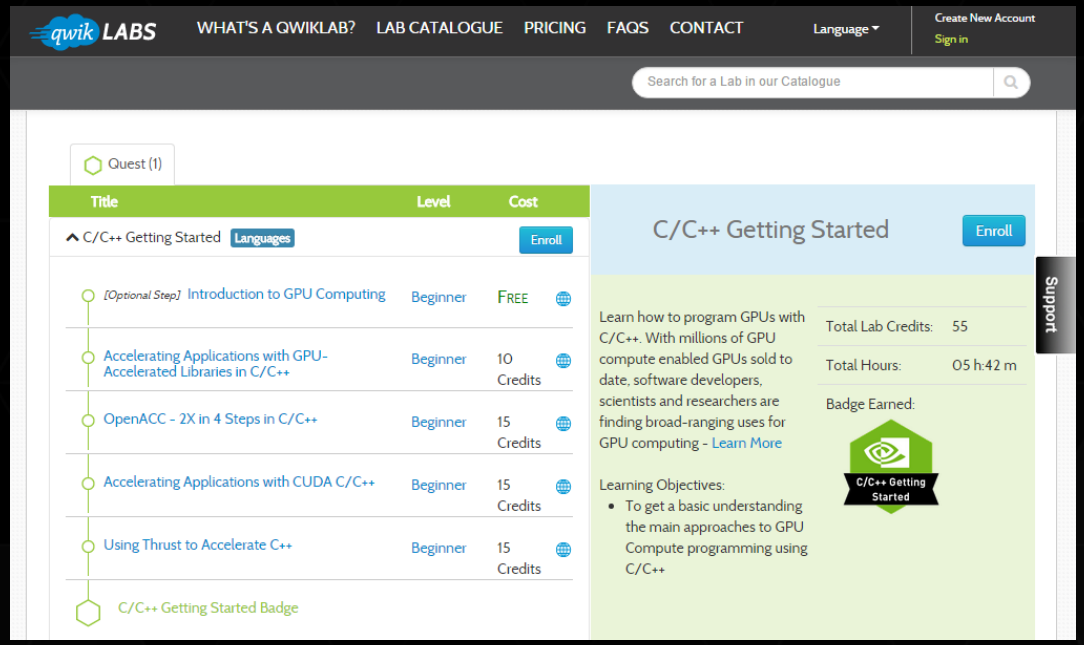
Getting Started on Your Own

LINKS TO GET STARTED

developer.nvidia.com

Self-paced labs: nvidia.qwiklab.com

90-minute labs, simply need a supported web browser



UDACITY PARALLEL PROGRAMMING COURSE

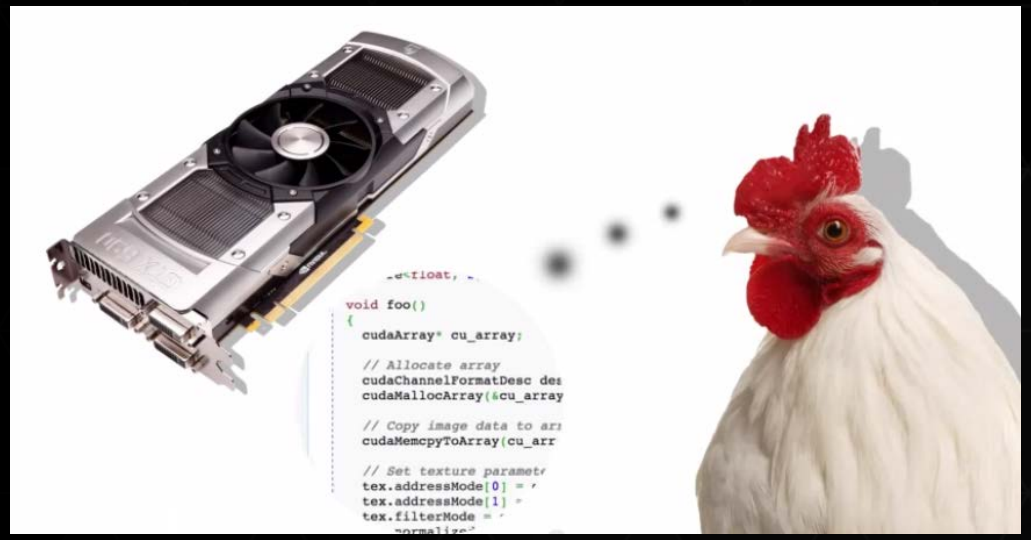
UDACITY

Learn. Think. Do.

Invent your future through free interactive college classes.

10,000+ Students Already Taking the course!

IT'S FREE!



Udacity.com

Thank you!