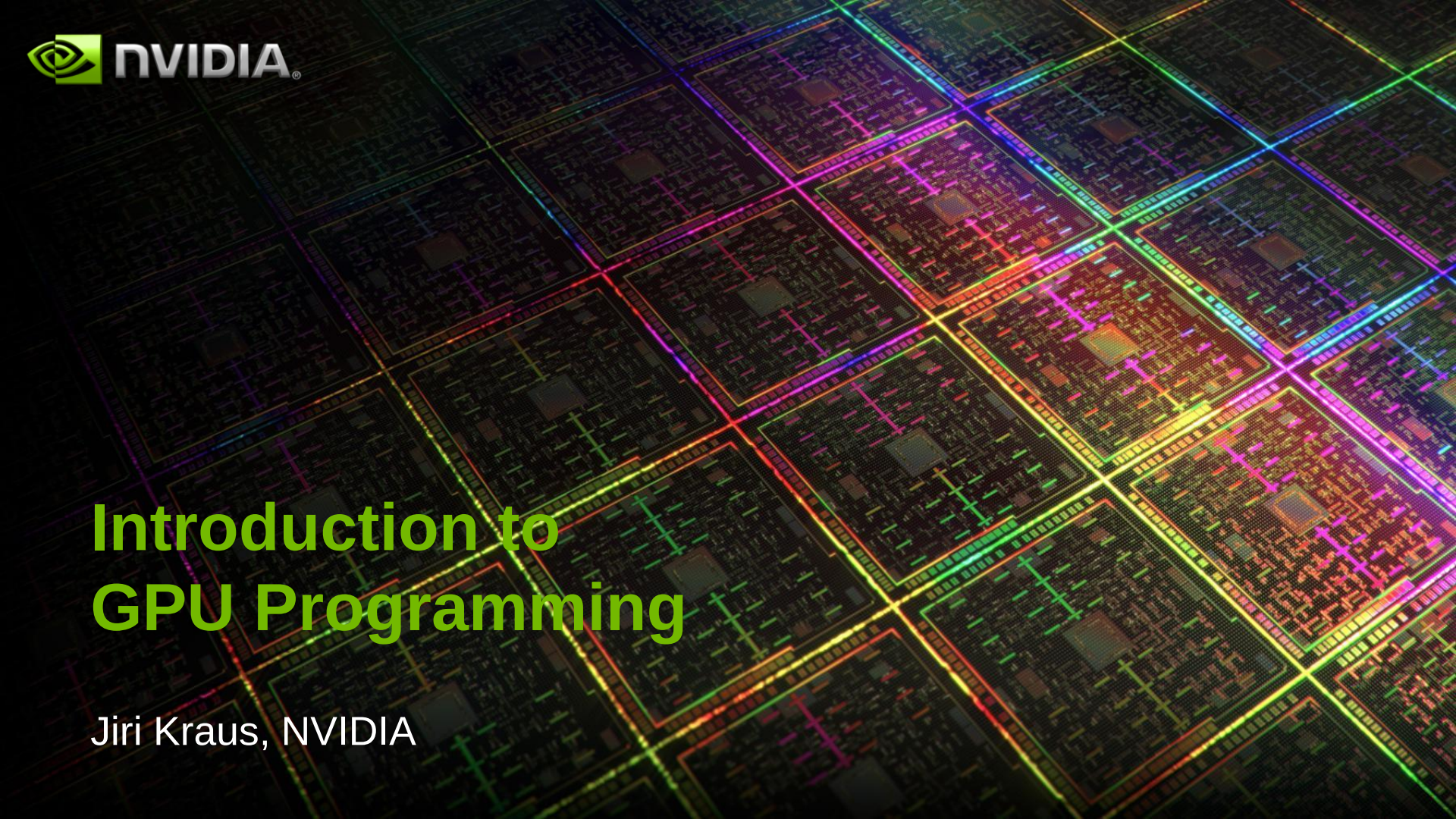

Hands-on with CUDA: C/C++ and Fortran
ISC13 Tutorial
NVIDIA

Goals

- Learn how to start programming with CUDA
- Understand how to use NVIDIA tools to identify performance issues and learn how to fix them
- Learn how CUDA can be combined with CUDA enabled libraries

Schedule

0900	Introduction to GPU computing and CUDA Basic Programming with CUDA
1100	Break
1130	Optimize a CUDA application with NVIDIA tools
1300	Close

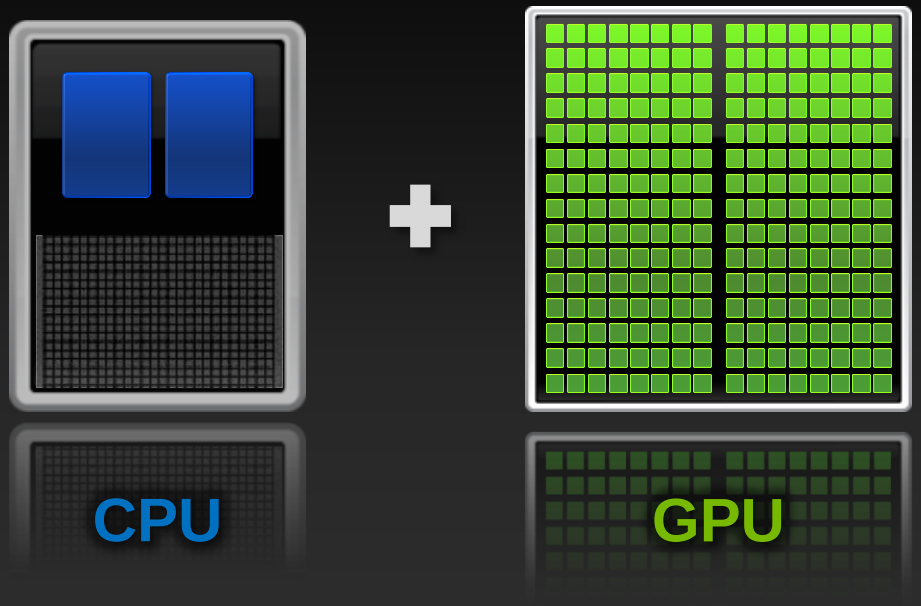


Introduction to GPU Programming

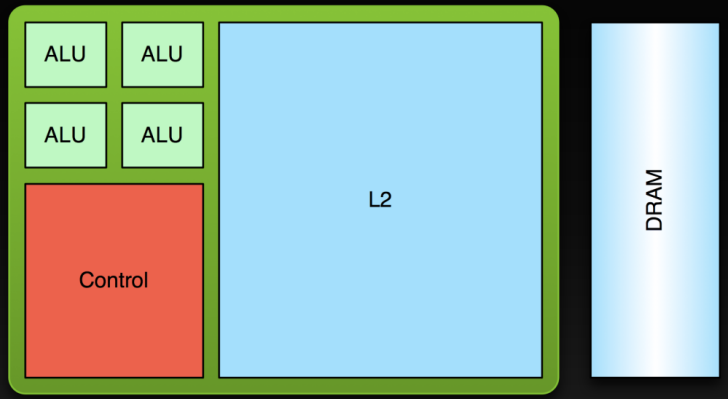
Jiri Kraus, NVIDIA

GPGPU Revolutionizes Computing

Latency Processor + Throughput processor

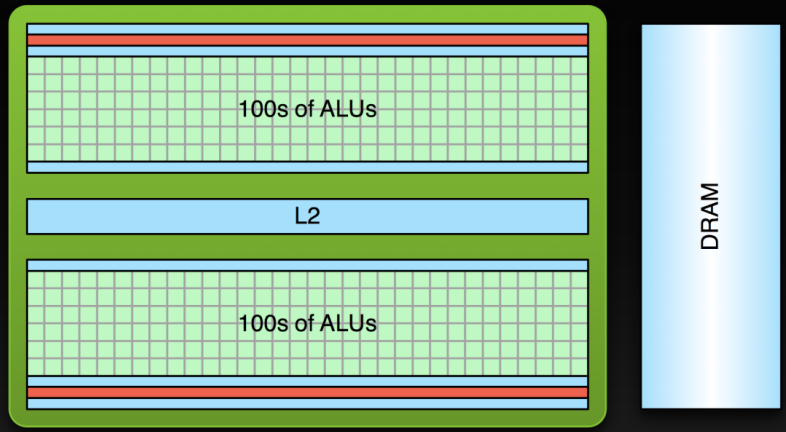


Low Latency or High Throughput?



CPU

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

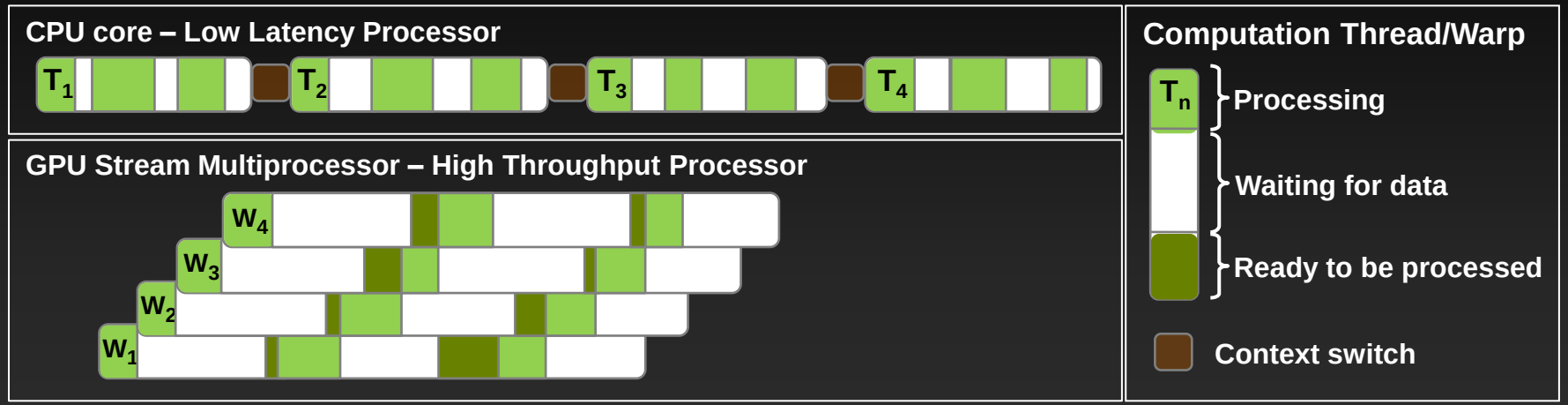


GPU

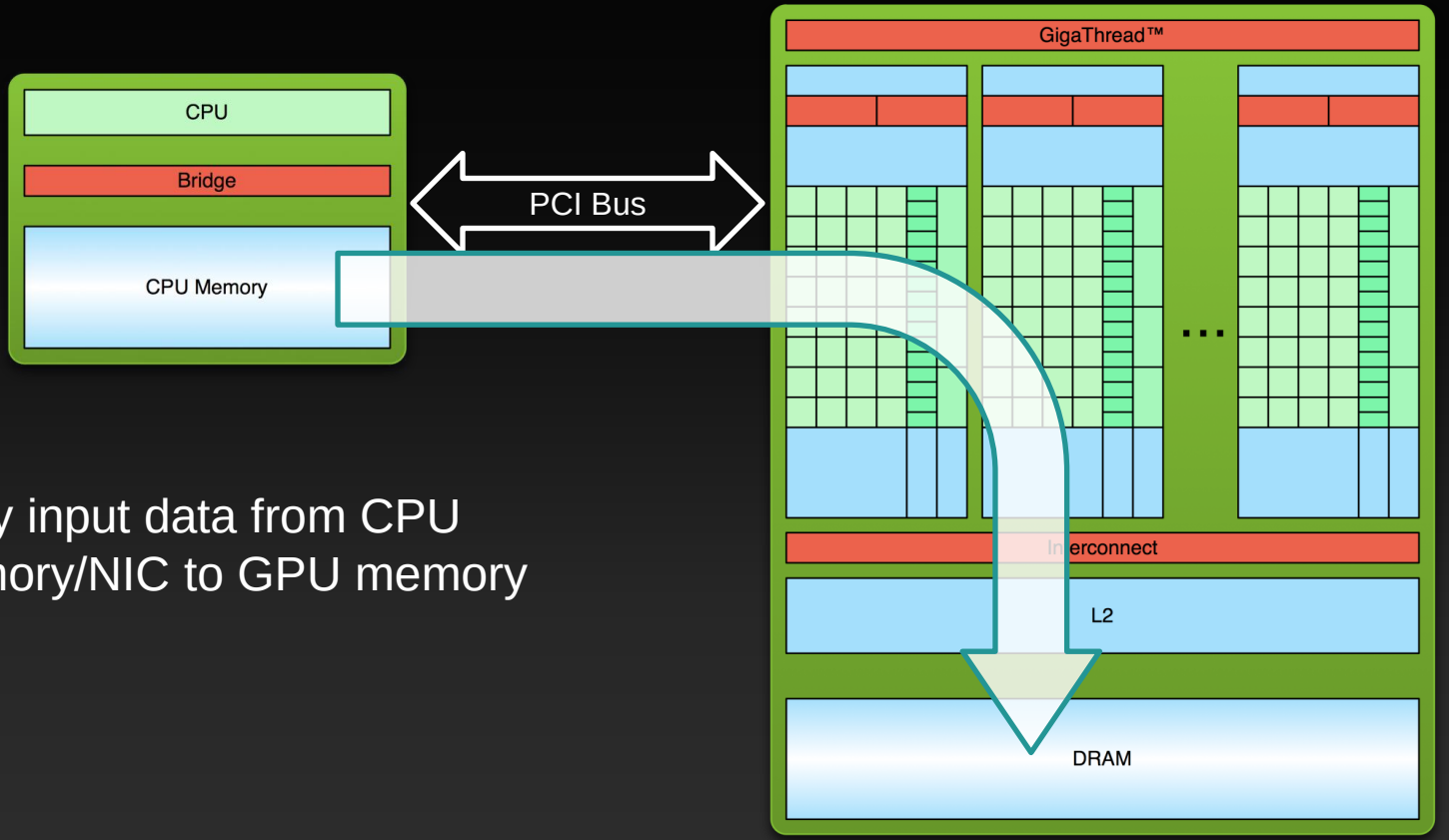
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

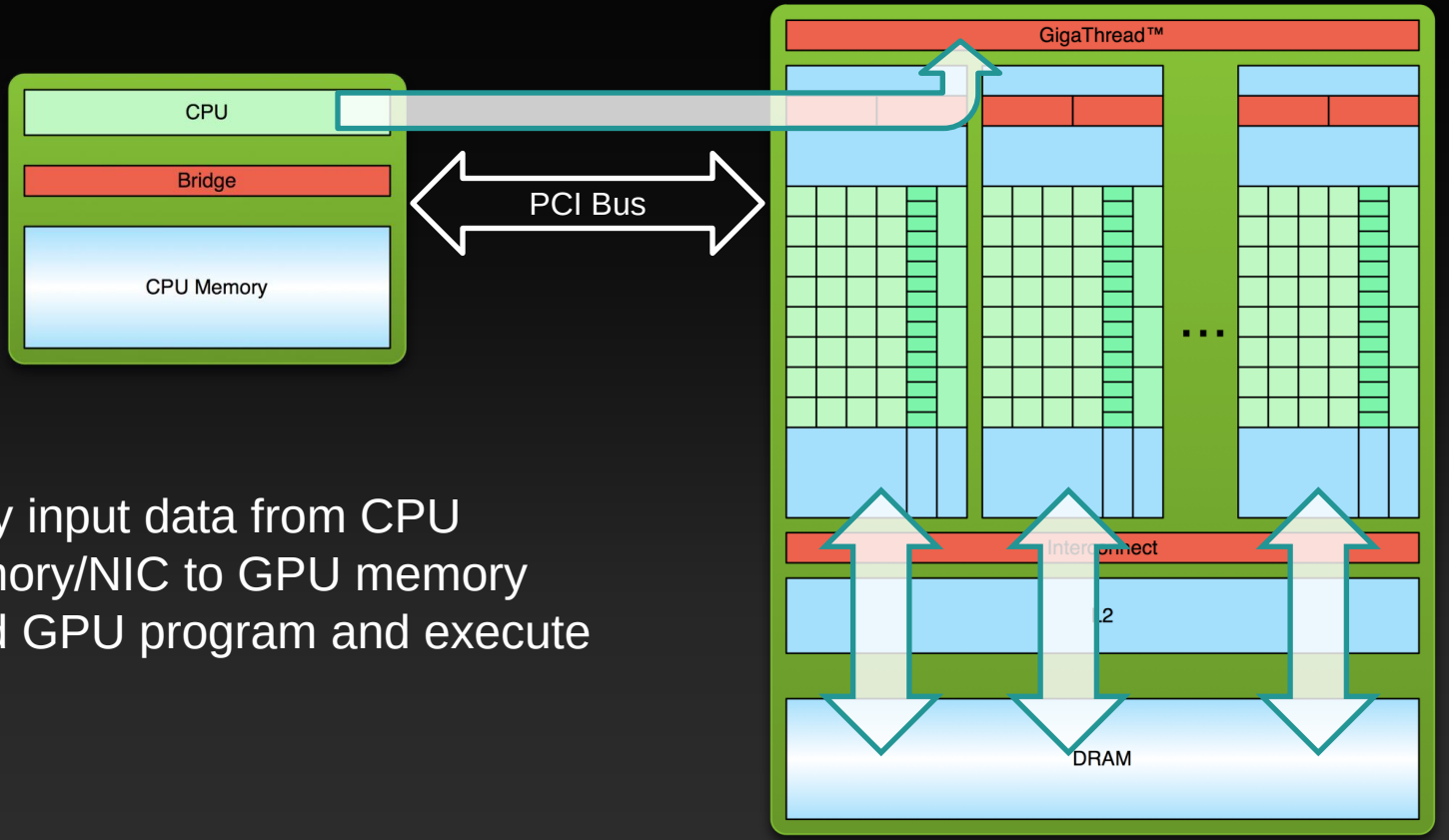


Simple Processing Flow



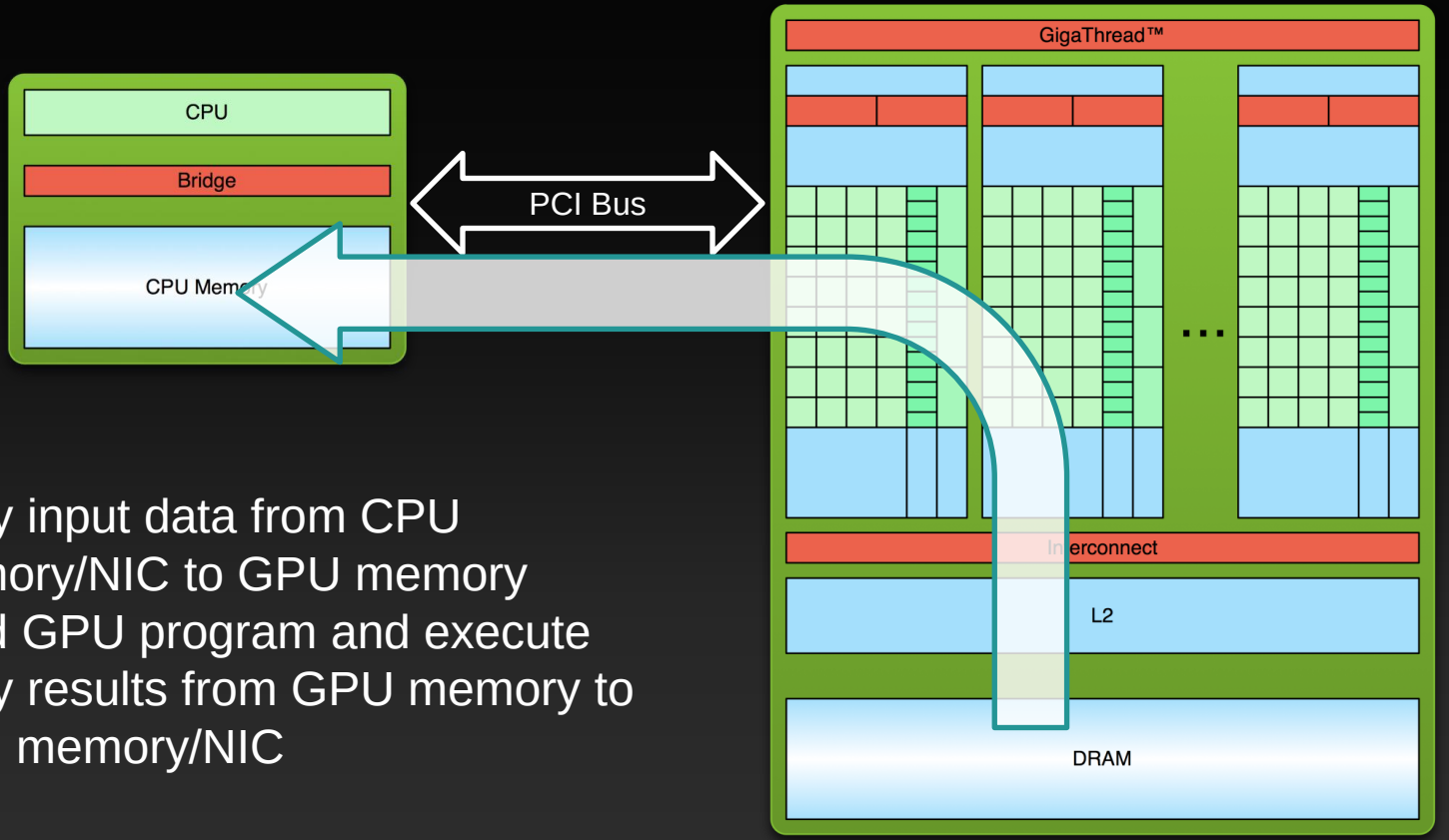
1. Copy input data from CPU memory/NIC to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute

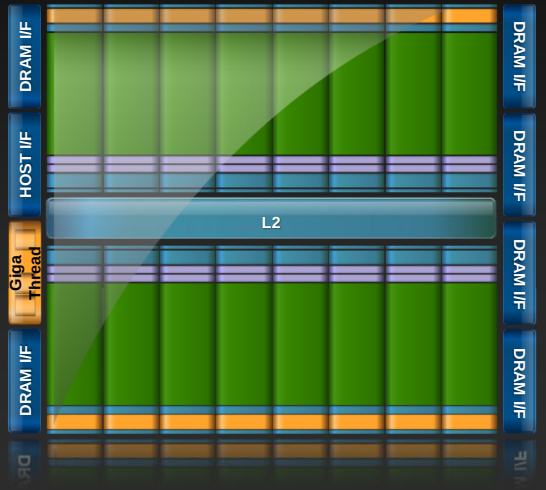
Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC

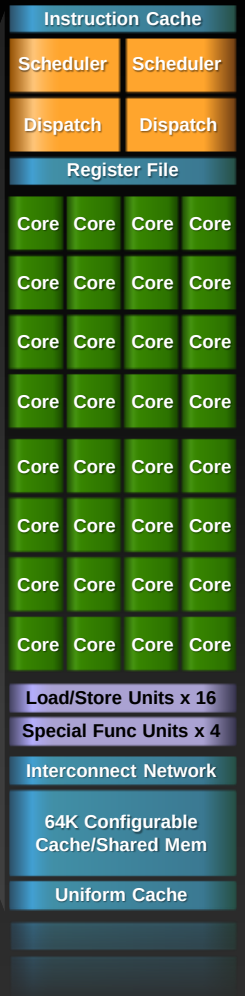
GPU Architecture: Two Main Components

- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB per GPU
  - Bandwidth currently up to ~250 GB/s (Tesla products)
  - ECC on/off (Quadro and Tesla products)
- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



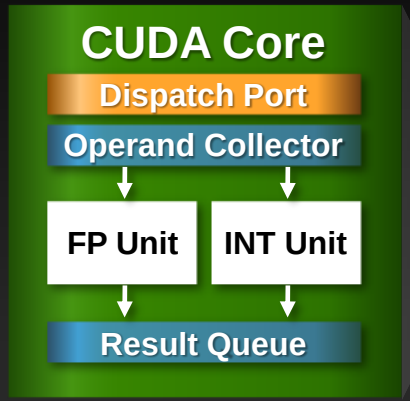
GPU Architecture – Fermi: Streaming Multiprocessor (SM)

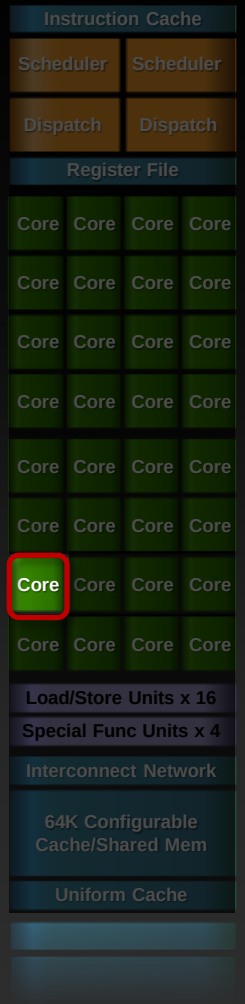
- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



GPU Architecture – Fermi: CUDA Core

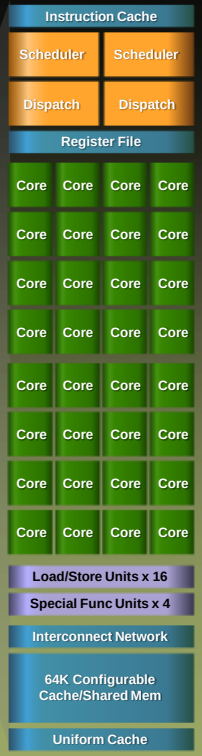
- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



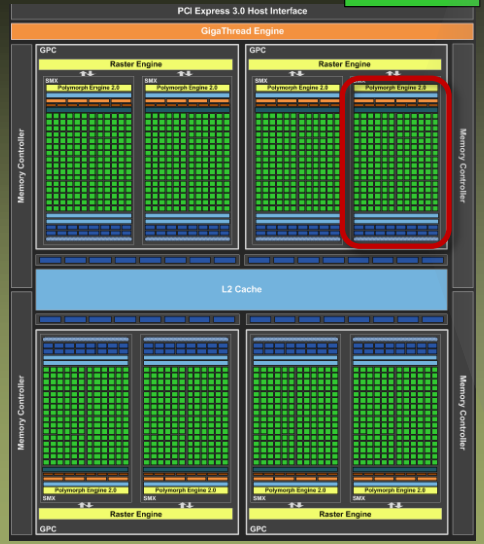


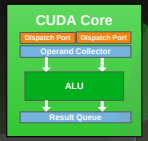
Kepler

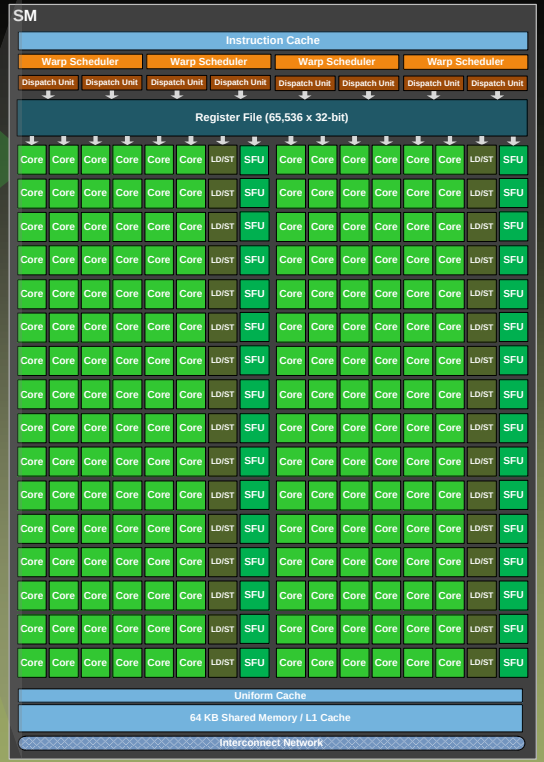
Fermi



Kepler







CUDA Review

PROGRAMMING MODEL

Anatomy of a CUDA Application

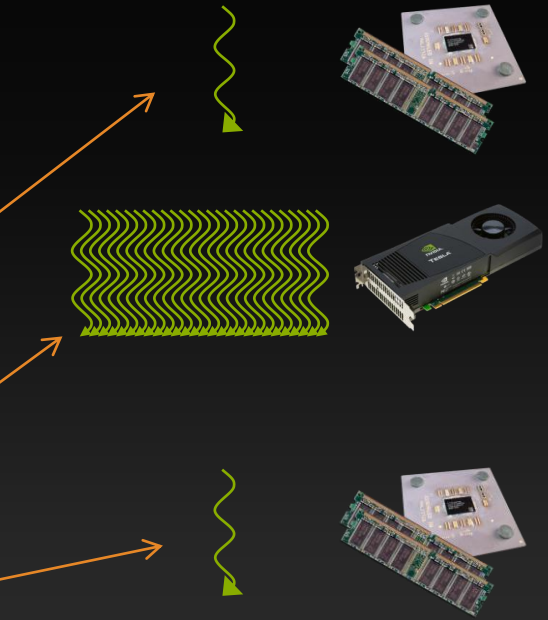
```
#include<iostream>
#include<algorithm>

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 fn

serial code

parallel code

serial code



CUDA Kernels

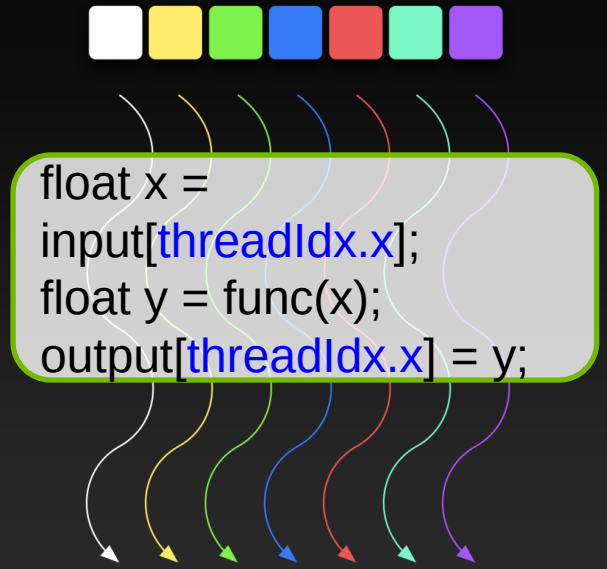
- **Parallel portion of application: execute as a kernel**
  - **Entire GPU executes kernel, many threads**

- **CUDA threads:**
  - **Lightweight**
  - **Fast switching**
  - **1000s execute simultaneously**

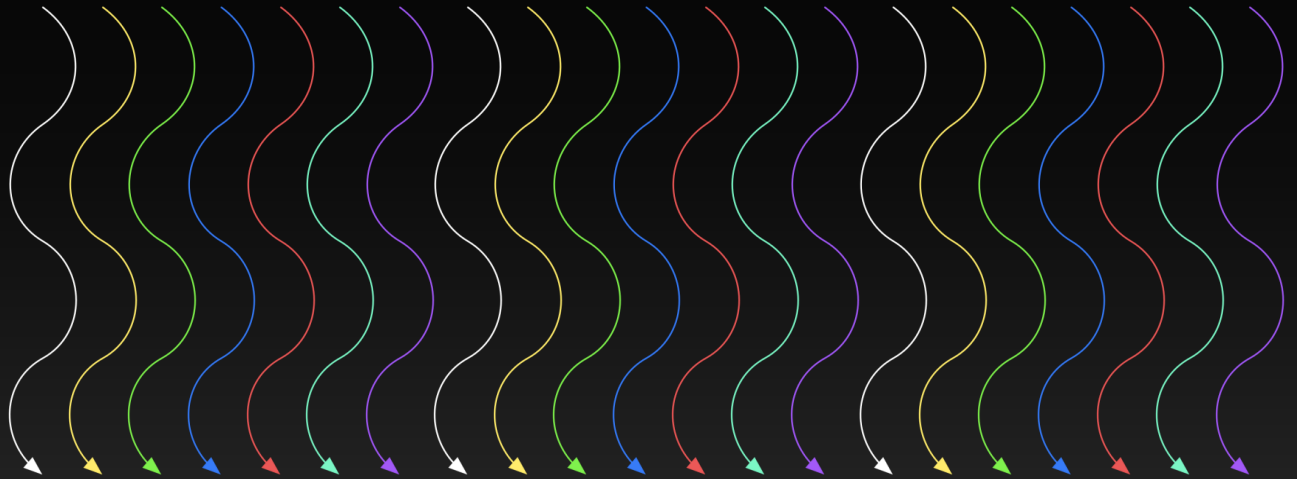
CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA Kernels: Parallel Threads

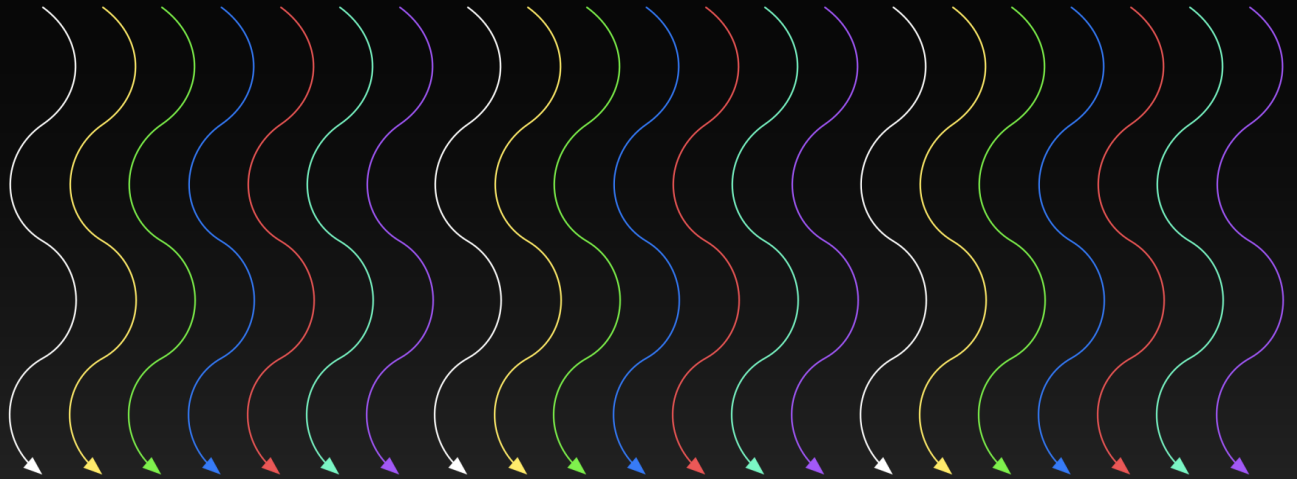
- A **kernel** is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions



CUDA Kernels: Subdivide into Blocks

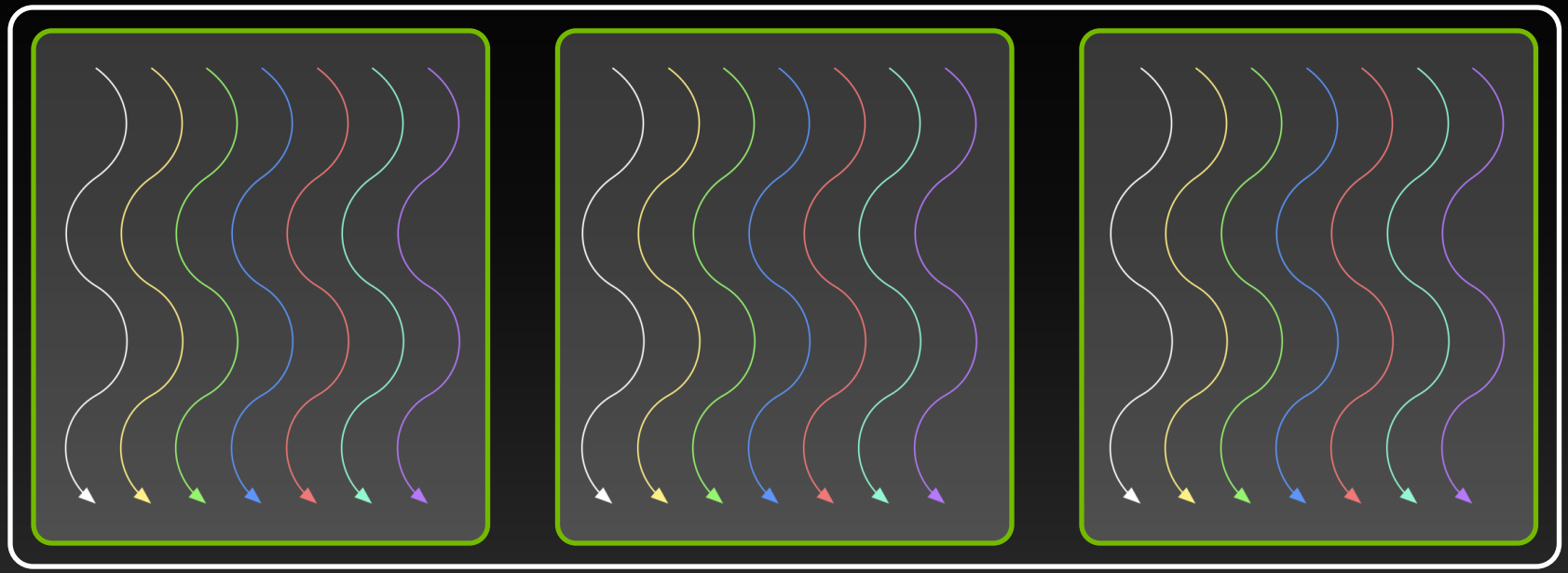


CUDA Kernels: Subdivide into Blocks



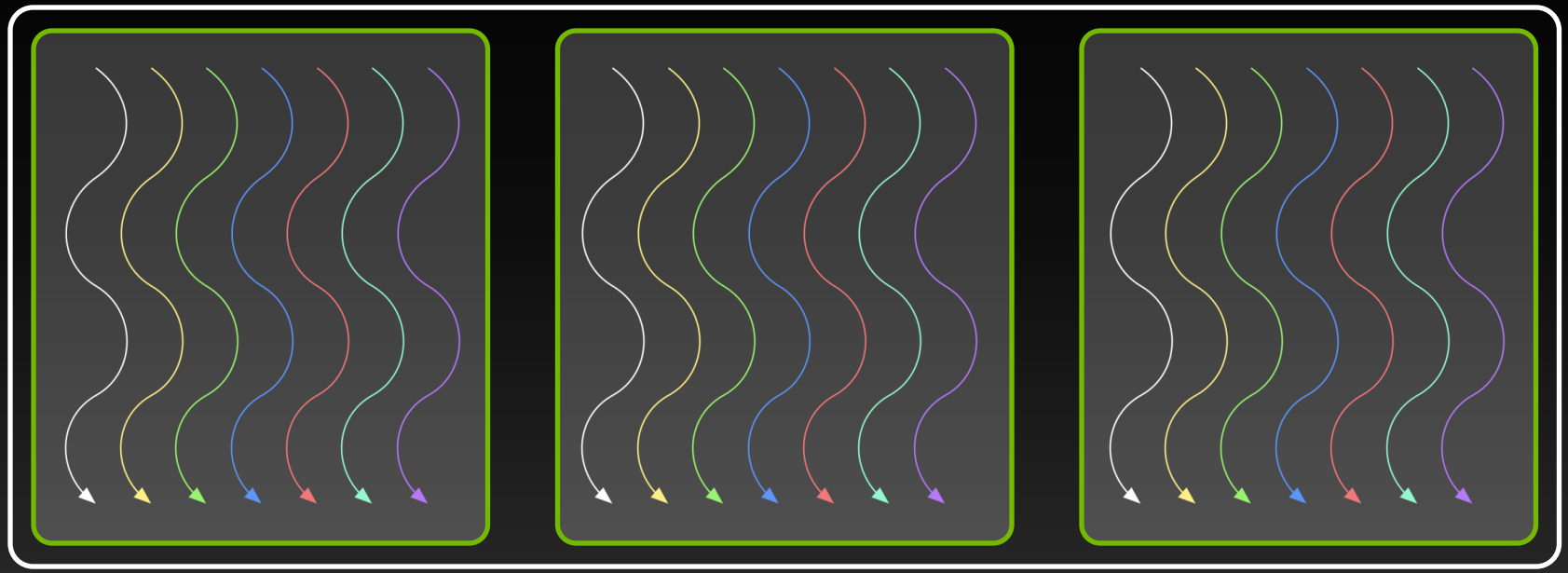
- Threads are grouped into blocks

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks

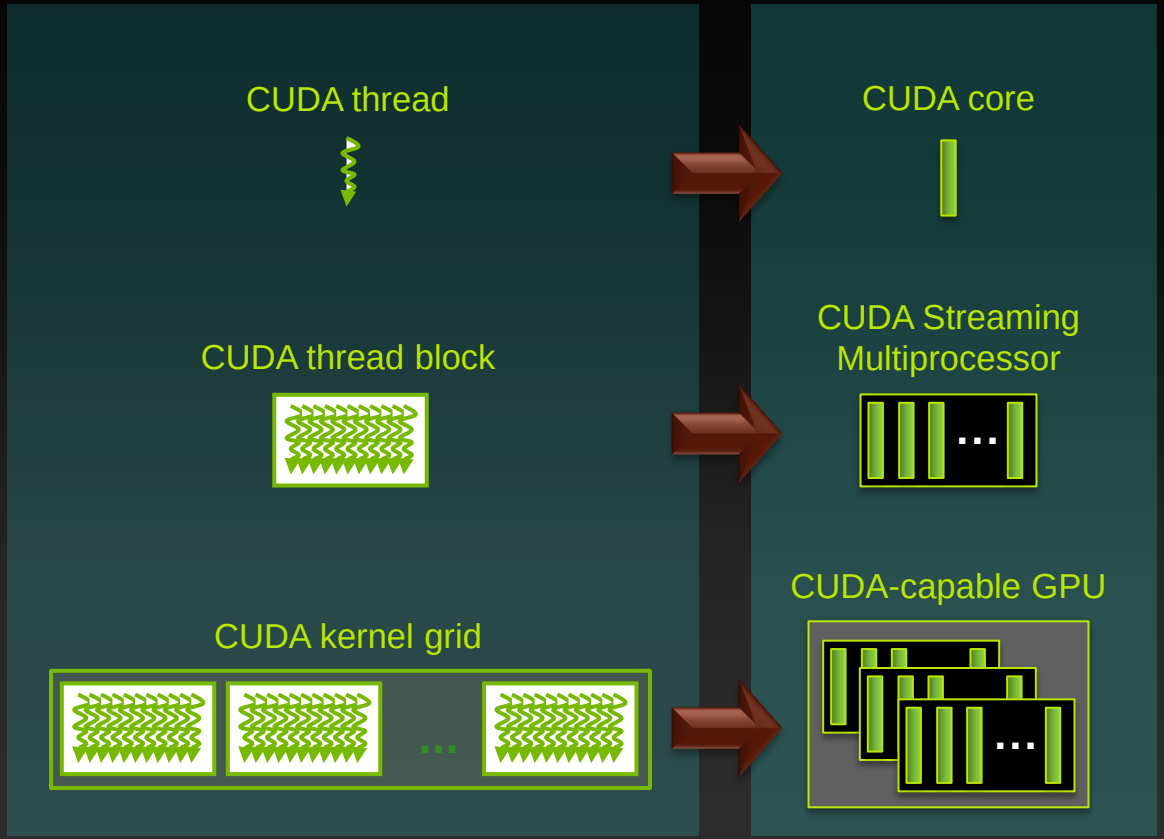


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

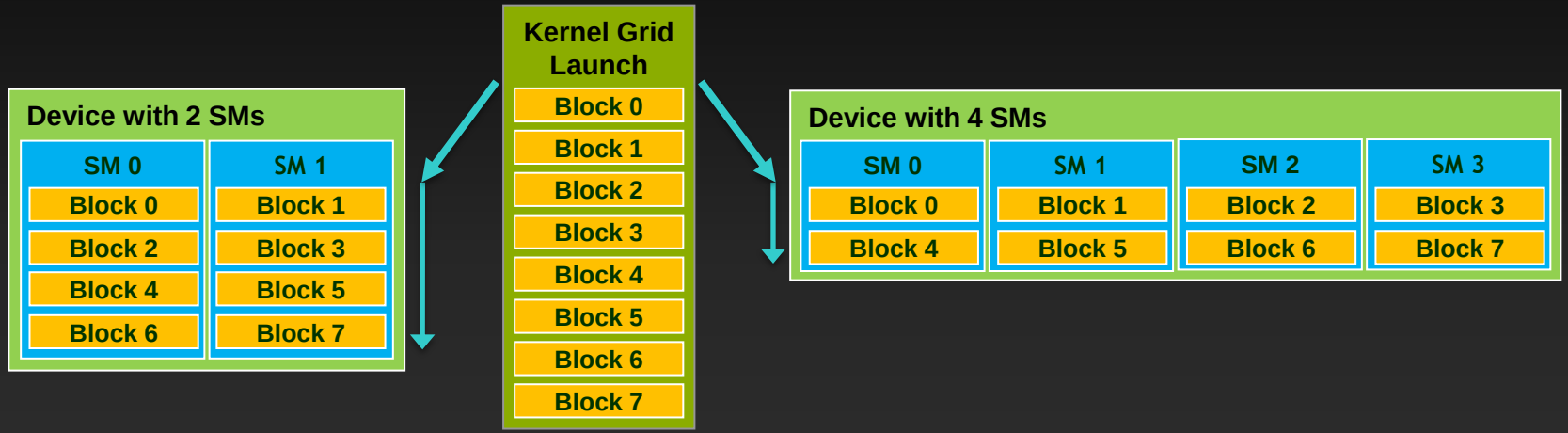
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Communication Within a Block

- Threads may need to cooperate
  - Memory accesses
  - Share results
- Cooperate using shared memory
  - Accessible by all threads within a block
- Restriction to “within a block” permits scalability
  - Fast communication between N threads is not feasible when N large

Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs



CUDA C

Standard C Code

```
void saxpy(int n, float a,
       float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

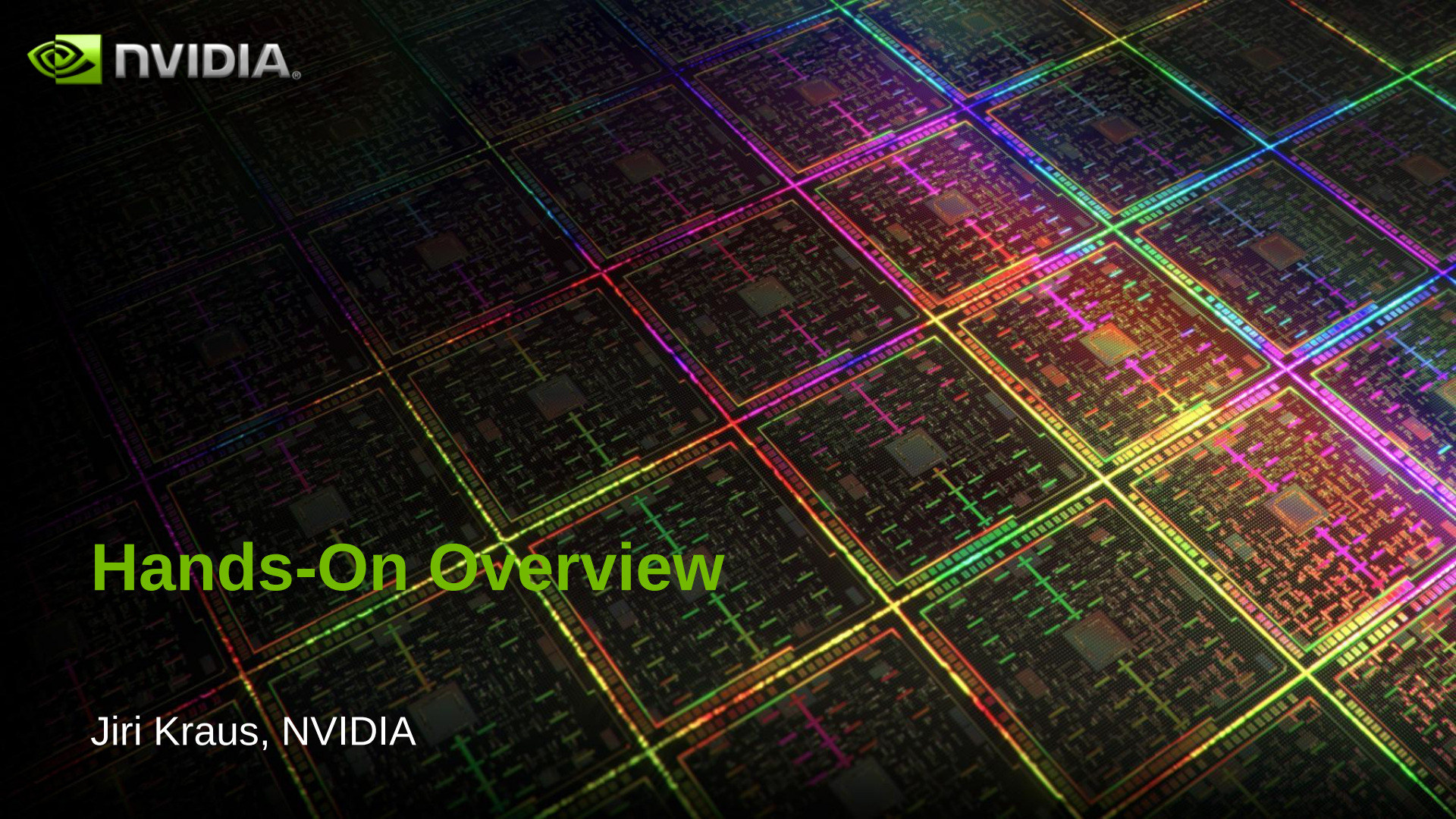
Parallel C Code

```
__global__
void saxpy(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];
}

int N = 1<<20;
cudaMemcpy(x, h_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, h_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(h_y, y, N, cudaMemcpyDeviceToHost);
```

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

What's Next?

- Start learning CUDA
- See how to use NVIDIA tools to identify performance issues and learn how to fix them.



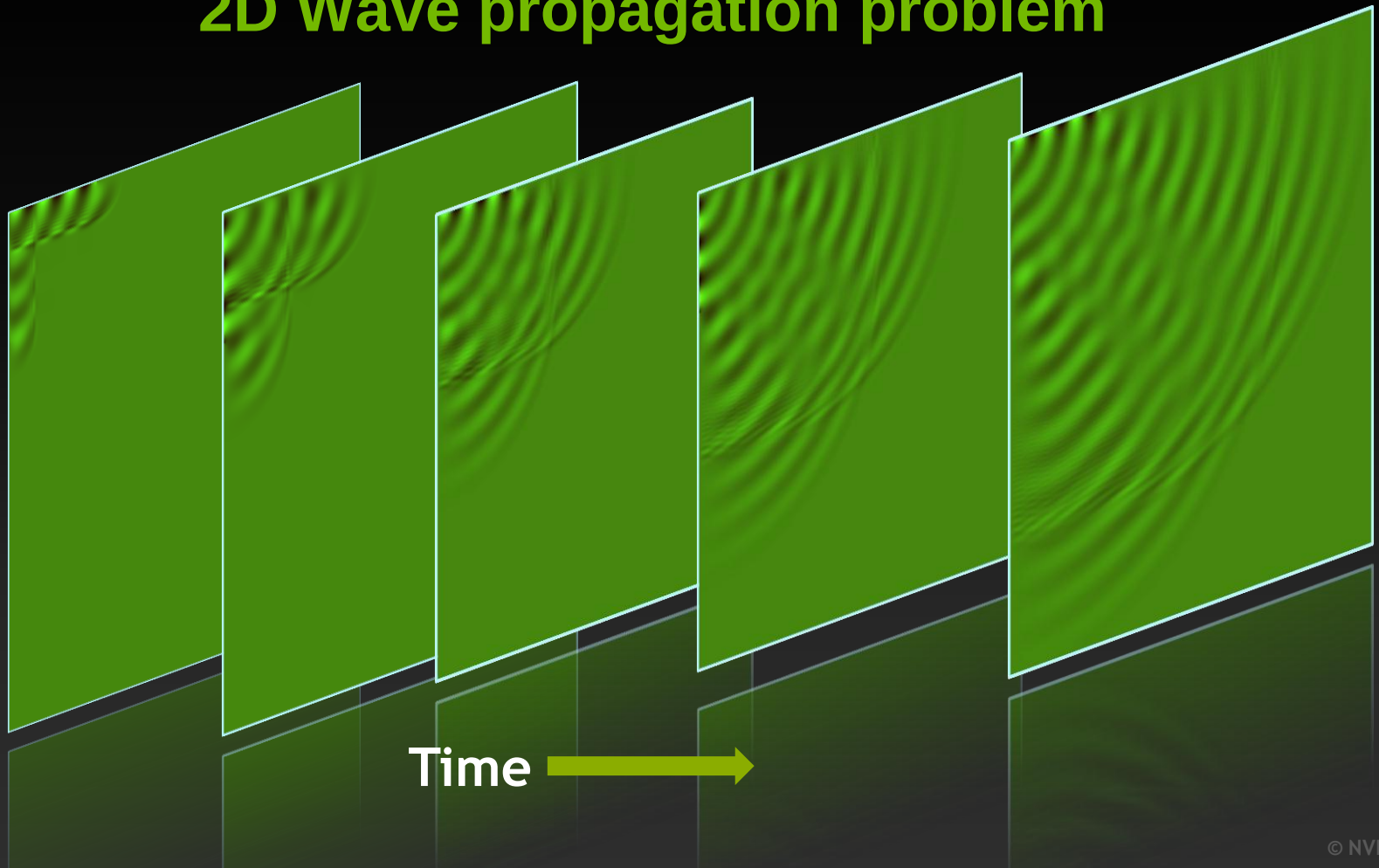
Hands-On Overview

Jiri Kraus, NVIDIA

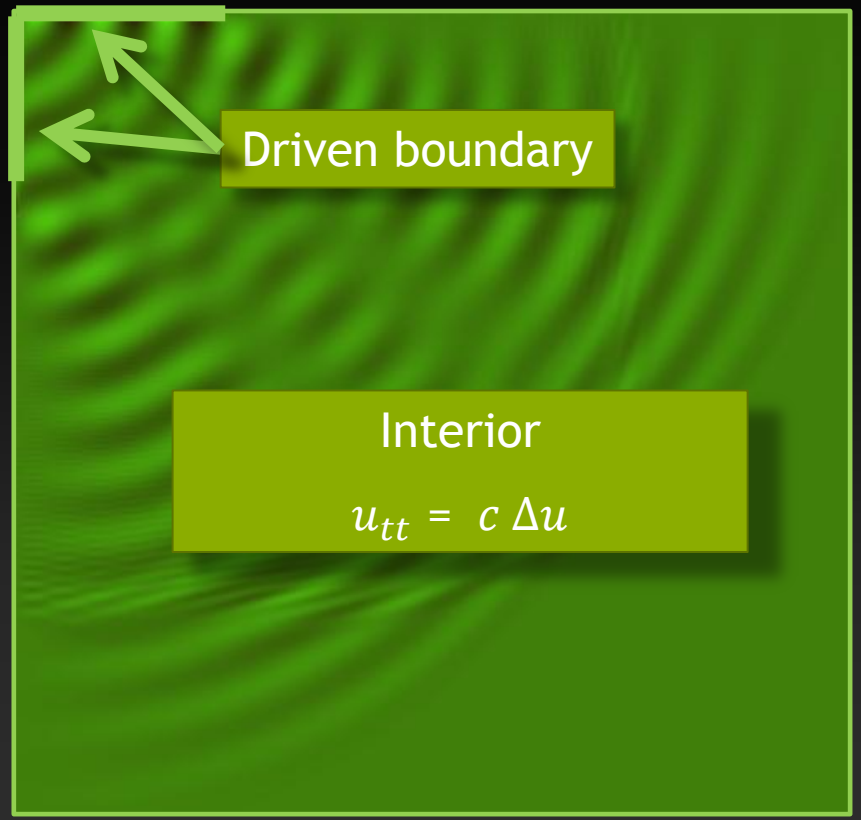
Hands-On

- We will use a basic 2D scalar wave solver as an example
  - Introduced next
- The tutorial examples are provided for
  - CUDA-Fortran in `~/euler/CUDA-Fortran`
  - CUDA C in `~/euler/CUDA-C`
- You can use
  - CUDA C or CUDA-Fortran
  - make + the editor of your choice and NVIDIA Visual Profiler
  - Nsight eclipse edition

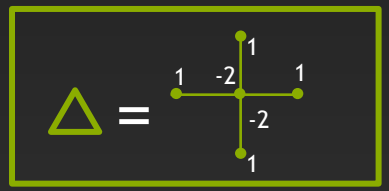
2D Wave propagation problem

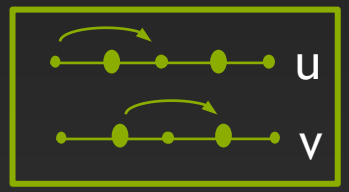


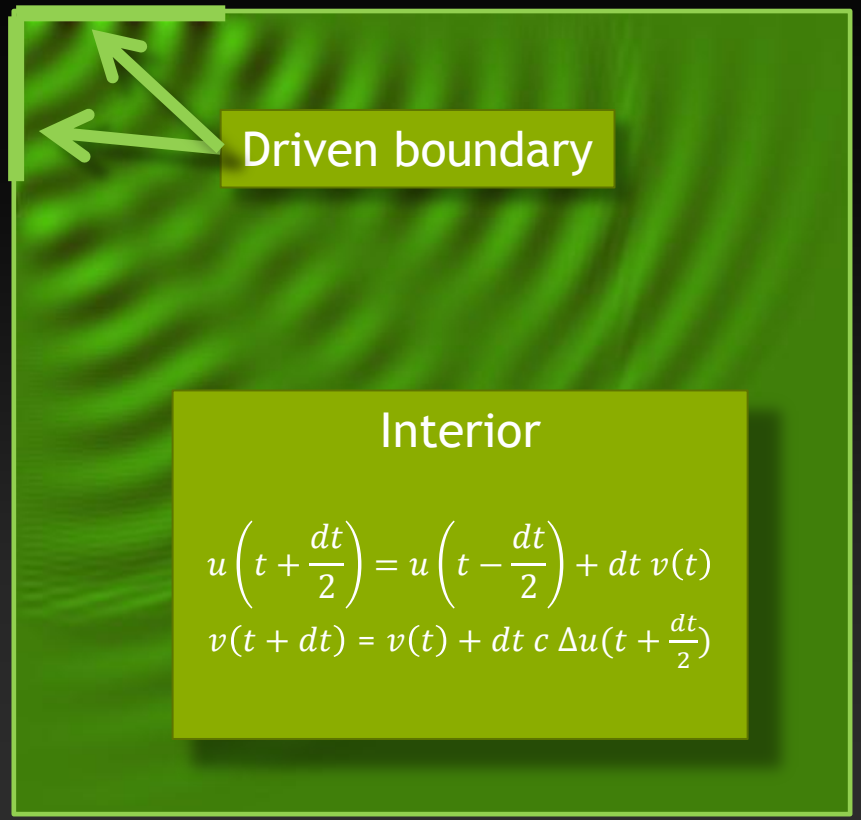
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$



A basic 2D scalar wave solver: $u_{tt} = c\Delta u$







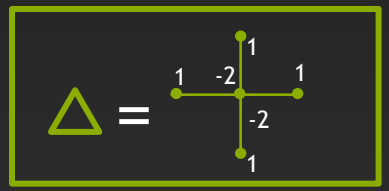
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$

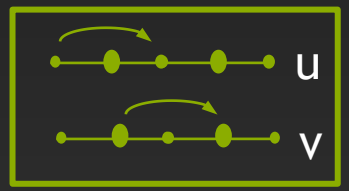
Initialize fields u, v

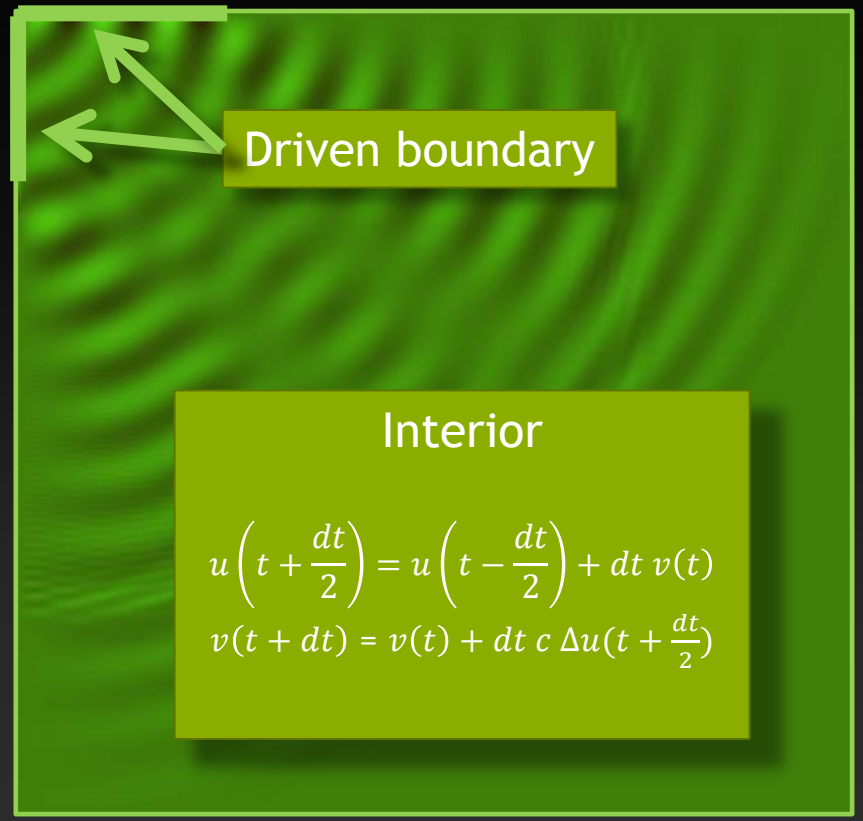
u = u + dt * v

v = v + dt * c * Δ u

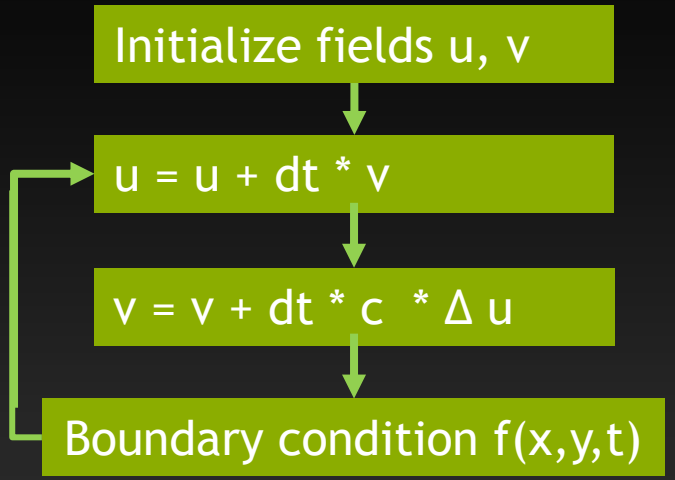
Boundary condition f(x,y,t)







Sample: Basic Euler Solver




```
do t = 1, 1000
  u(:,:) = u(:,:) + dt  * v(:,:)

  do y=2, ny-1
        do x=2,nx-1
                    v(x,y) = v(x,y) + dt * c * (  &
                              (u(x,y+1) - 2*u(x,y) + u(x,y-1)) / dy2 +  &
                              (u(x+1,y) - 2*u(x,y) + u(x-1,y)) / dx2 )
        end do
   end do

   call BoundaryConditon(u)
end do
```

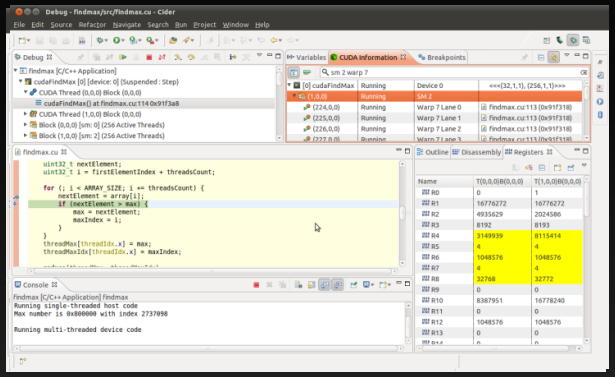
NVIDIA CUDA C Compiler NVCC

- Use NVIDIAs CUDA C Compiler to compile CUDA C programs
  - Replace compiler and linker with nvcc (e.g. nvcc instead of gcc)
  - rename files to *.cu
  - Add –arch flag matching the architecture of your GPU
    - We are using Fermi: -arch=sm_20
- nvcc splits source file in host part and device part
  - Host part is passed through to host compiler (e.g. gcc)
    - Use –Xcompiler to pass command line options to host compiler

PGI CUDA Fortran Compiler pgf90

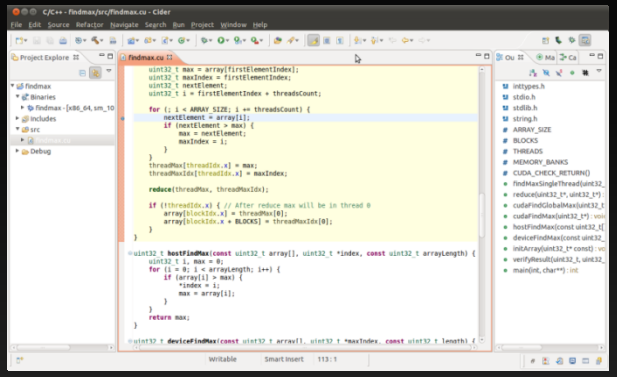
- PGIs Fortran Compiler comes with CUDA Fortran
  - add –Mcuda=cc20 to your command line
  - add –Mcuda to your linker line
  - Rename files to **.CUF

NVIDIA® Nsight,™ Eclipse Edition



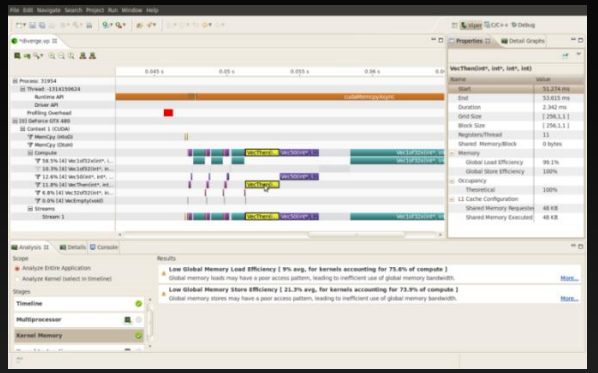
CUDA-Aware Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs



Nsight Debugger

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



Nsight Profiler

- Quickly identifies performance issues
- Integrated expert system
- Automated analysis
- Source line correlation

Available for Linux and Mac OS

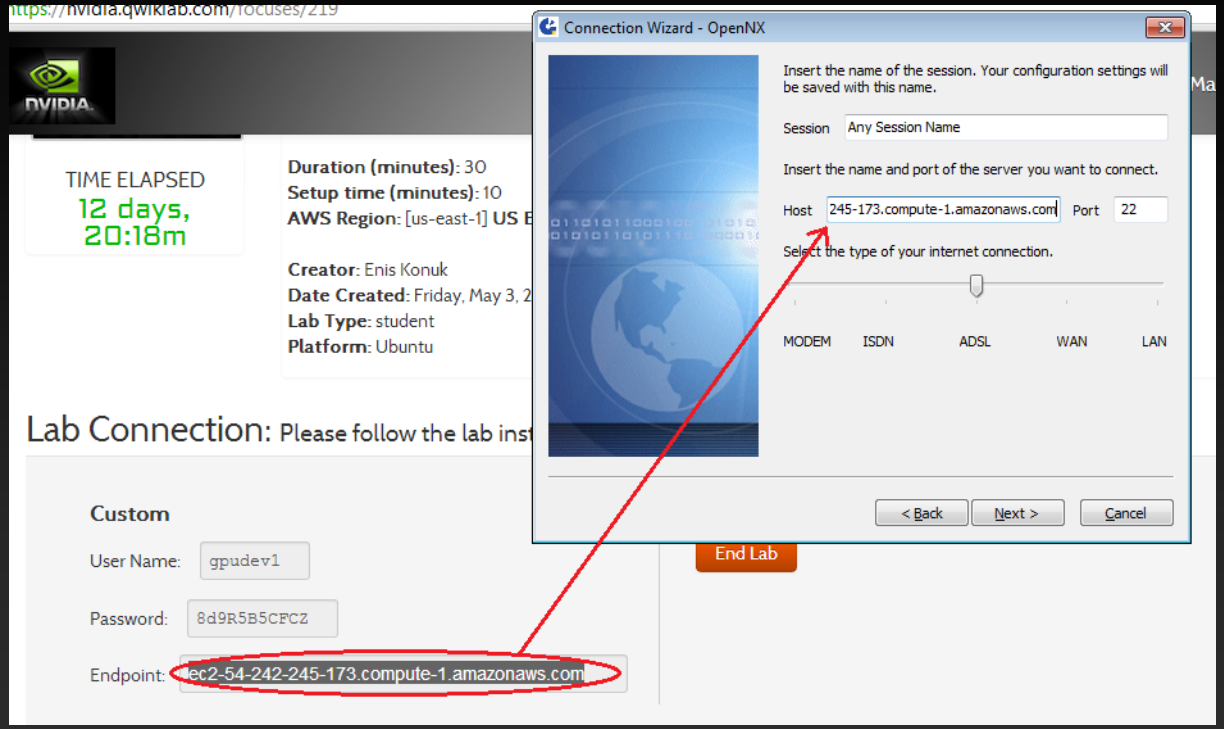
Task 1: Setup Compilation tool chain

- Update Makefile and rename files
  - CUDA C:
    - use nvcc instead of gcc
    - add –arch=sm_20
    - rename *.c -> *.cu
  - CUDA Fortran:
    - add –Mcuda=cc20 flag
    - rename *.F90 -> *.CUF
  - Follow TODOs in Makefile
- Compile and run
- Optional: Use Nsight Eclipse Edition (explained next)

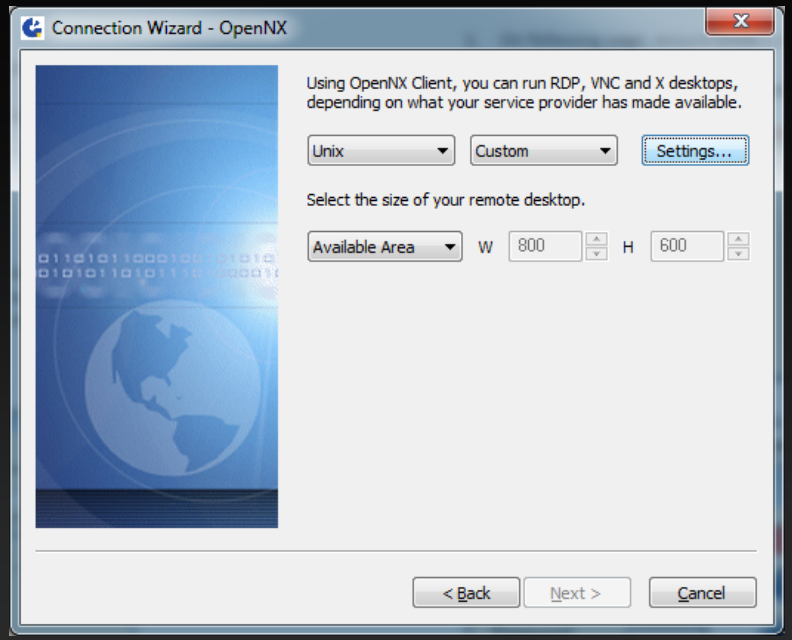
How to connect to Amazon EC2?

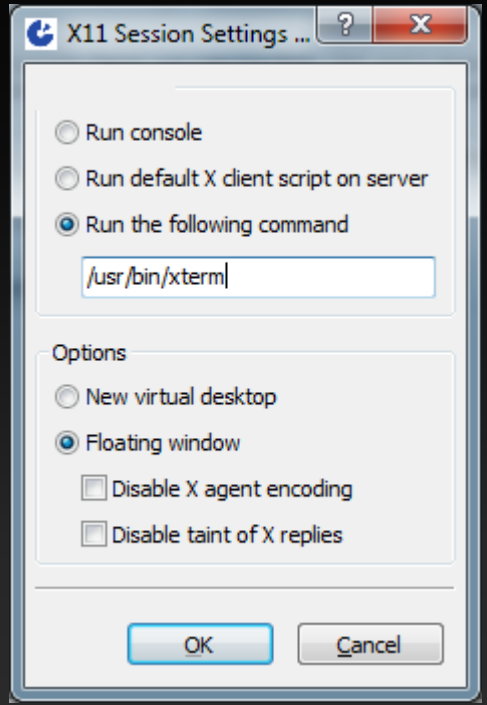
1. If you have not done already:
   - Download and install OpenNX
   - Instructions: https://developer.nvidia.com/cuda-cloud
2. Surf to https://nvidia.qwiklab.com/
   1. Sign in and select appropriate class
   2. Find the correct lab‘s listing, and once enabled press Start Lab
   3. Enter the Token number provided when prompted
3. Setting up OpenNX Client Connection

Setting up OpenNX Client Connection I (demo)

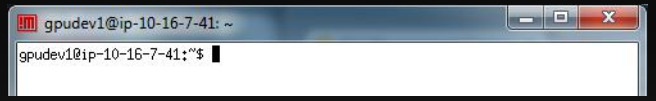


Setting up OpenNX Client Connection II (demo)

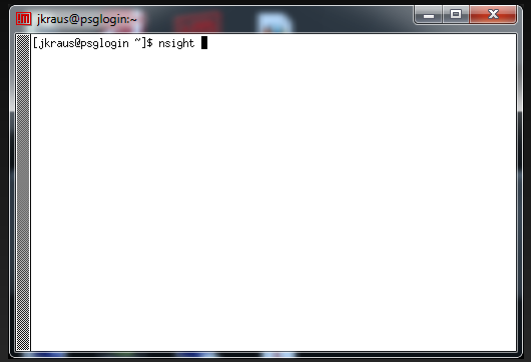


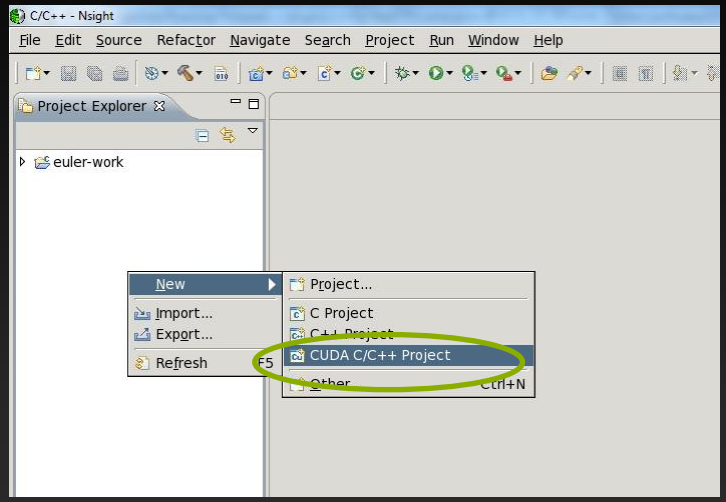


Setting up OpenNX Client Connection III (demo)

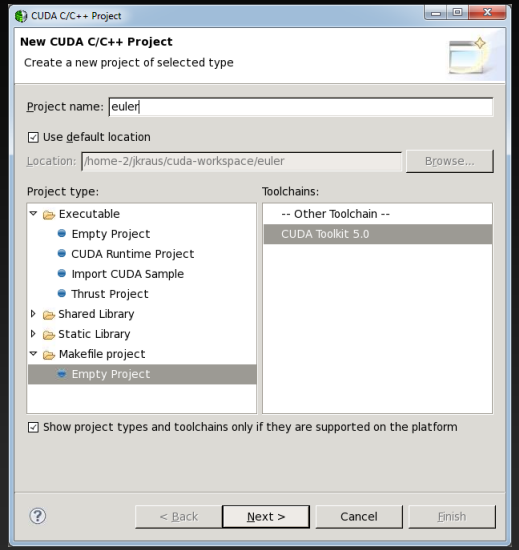


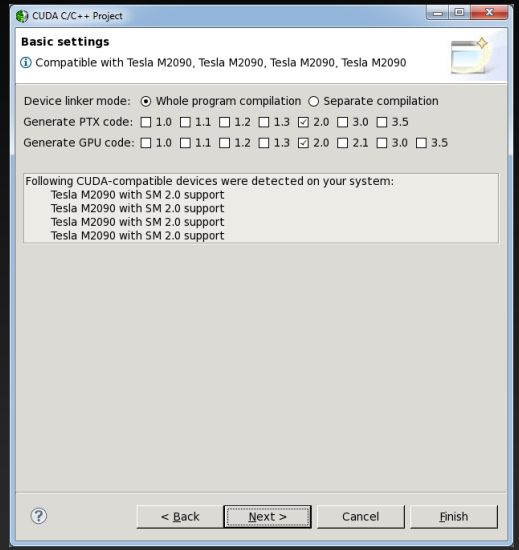
Use Nsight Eclipse Edition – Import existing Makefile project (demo)

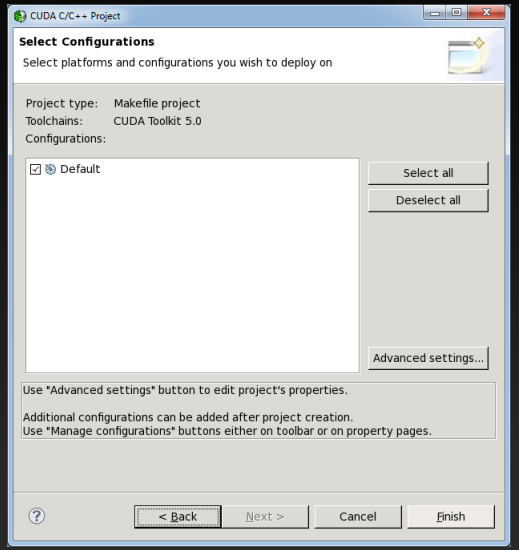




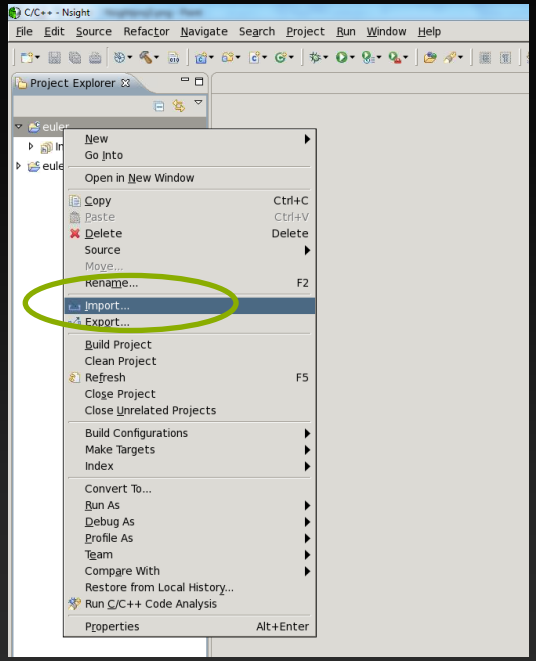
Use Nsight Eclipse Edition – Import existing Makefile project (demo)

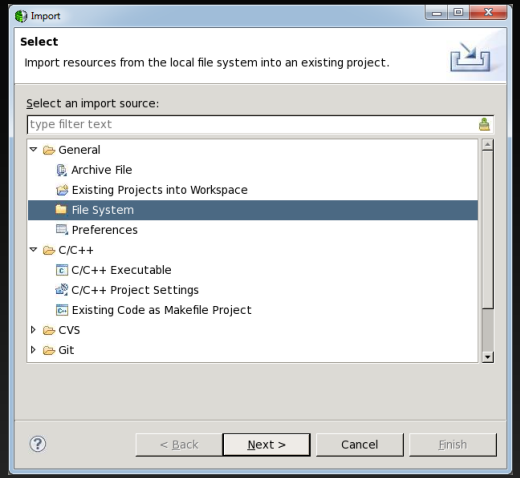


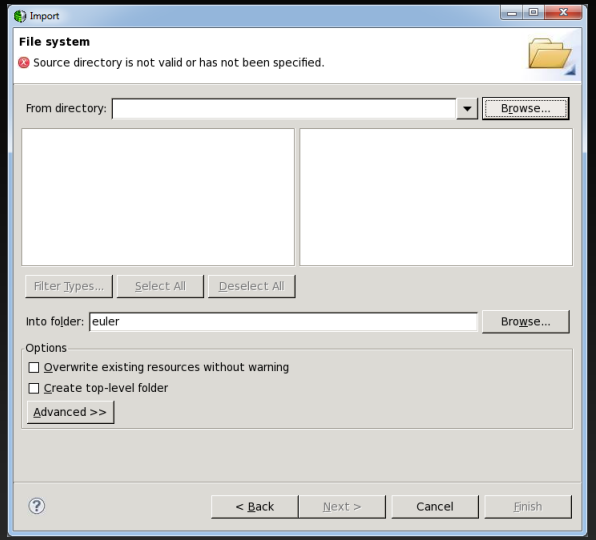




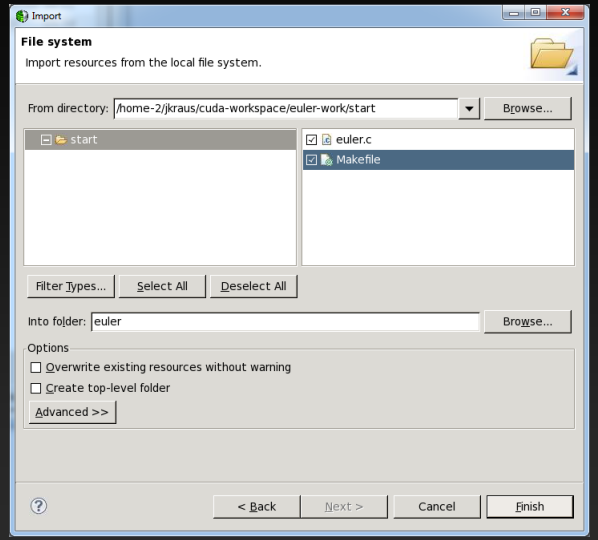
Use Nsight Eclipse Edition – Import existing Makefile project (demo)

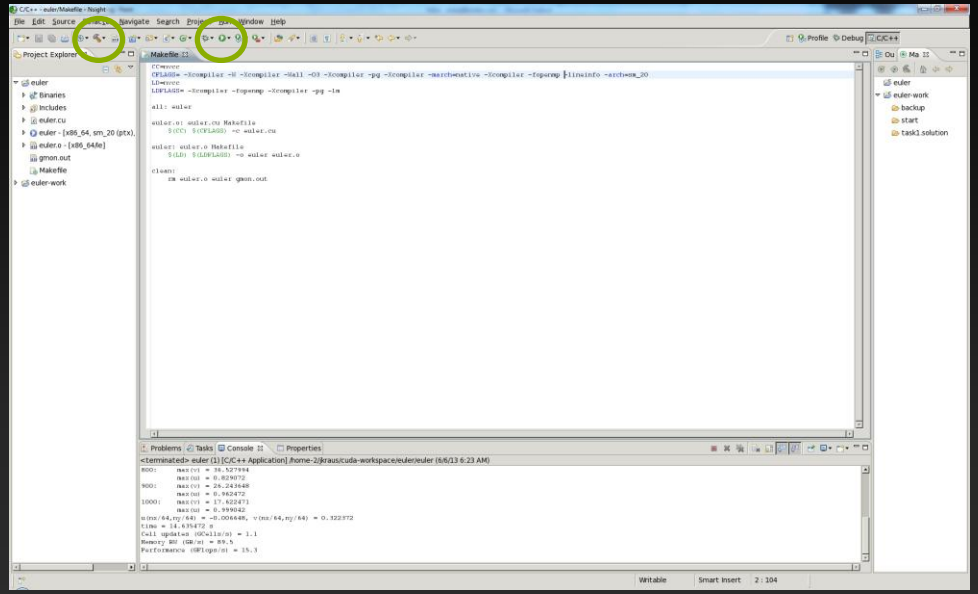






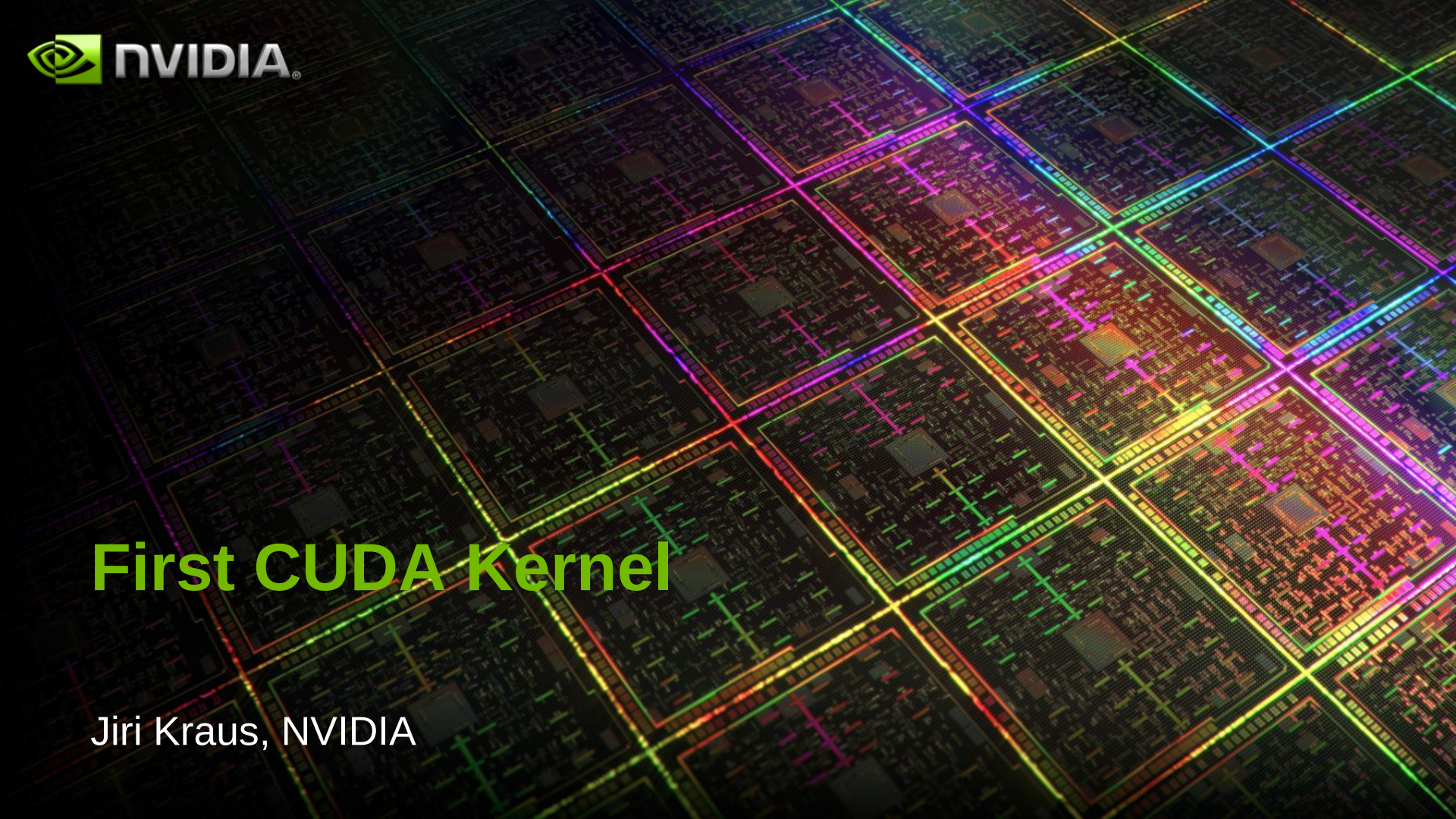
Use Nsight Eclipse Edition – Import existing Makefile project (demo)





Prepare Hand-On

- OpenNX Instructions:
  https://developer.nvidia.com/cuda-cloud
- Examples are in:
  `~/euler/CUDA-C`
  `~/euler/CUDA-Fortran`
- Task 1:
  - Follow TODOs in
    `~/euler/CUDA-Fortran/task1/Makefile`
    `~/euler/CUDA-C/task1/Makefile`

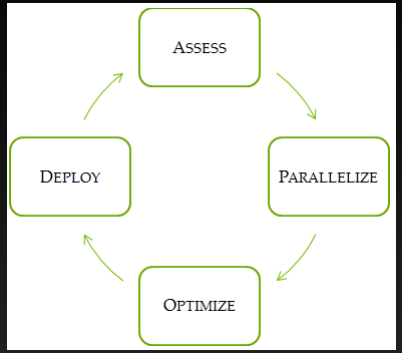


First CUDA Kernel

Jiri Kraus, NVIDIA

Assess, Parallelize, Optimize, Deploy

- This Tutorial follows the Access, Parallelize, Optimize, Deploy cycle



See also:

- CUDA C Best Practices Guide

  http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

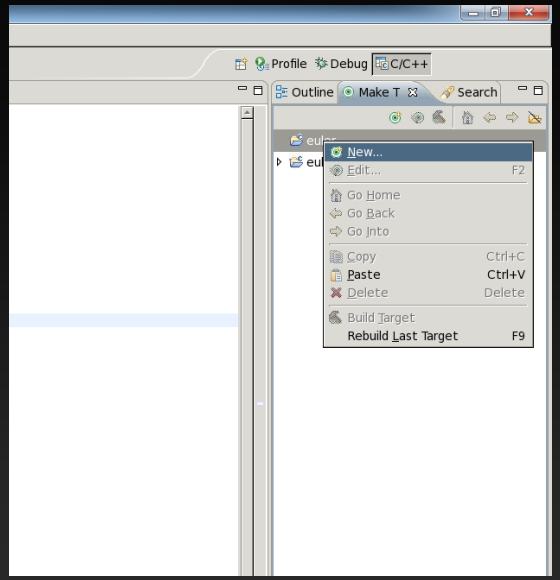
- Parallel Forall: Assess, Parallelize, Optimize, Deploy

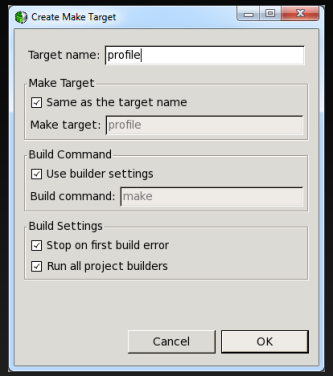
  https://developer.nvidia.com/content/assess-parallelize-optimize-deploy

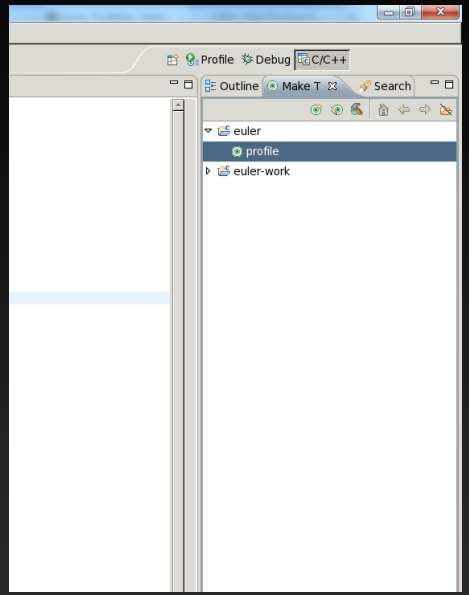
Asses CPU Version (demo)

- Use gprof to identify most time consuming CPU method

Asses CPU Version (demo)







Asses CPU Version (demo)

```
gprof ./euler
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 73.93     45.02    45.02     1000    45.02    45.02  update_v
 25.25     60.40    15.38     1000    15.38    15.38  update_u
  0.40     60.64     0.24       20    12.03    12.03  max_value
  0.05     60.67     0.03                             main
  0.02     60.68     0.01     1000     0.01     0.01  calc_bc
```

CUDA Version of Euler

Basic necessary steps (besides compiler and linker)

- **Initialize CUDA:**

  CUDA-C: `cudaSetDevice( device-ordinal )`

  CUDA-Fortran: `use cudafor`

- **Shutdown CUDA:**

  CUDA-C: `cudaDeviceReset()`

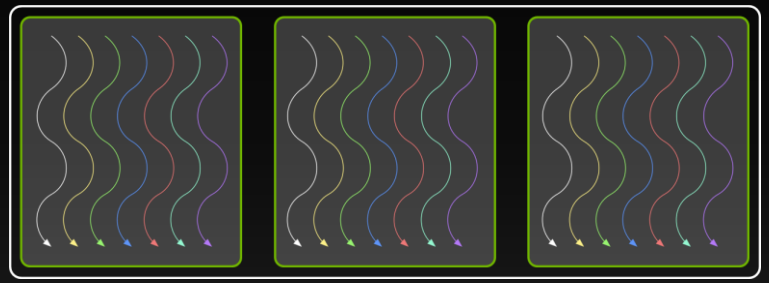
update_v

```
void update_v(double* v, double* u,
    int nx, int ny, double dt,
    double dx2, double dy2)
{
    for (int y=1; y<(ny-1); ++y) {
        for (int x=1; x<(nx-1); ++x) {
            v[y*nx+x] = v[y*nx+x] + dt * 100.0 * (
                      (u[(y+1)*nx+ x   ]
                - 2.0*u[y*nx+x]
                    + u[(y-1)*nx+ x   ])/dy2 +
                     (u[ y   *nx+(x+1)]
                - 2.0*u[y*nx+x]
                    + u[ y   *nx+(x-1)])/dx2 );
        }
    }
}
```

- Write of `v[y*nx+x]` depends on
  - Itself
  - elements of `u`
- `u` does not change in `update_v` (read only)
- Conclusion: all updates of `v` are independent

Threads/Blocks/Grids

- Threads are grouped into blocks
  - Up to 1536 per block for Fermi
  - Identified with `threadIdx.x`
- Blocks are grouped into a grid
  - Identified with `blockIdx.x`
  - Blocksize is given by `blockDim.x`

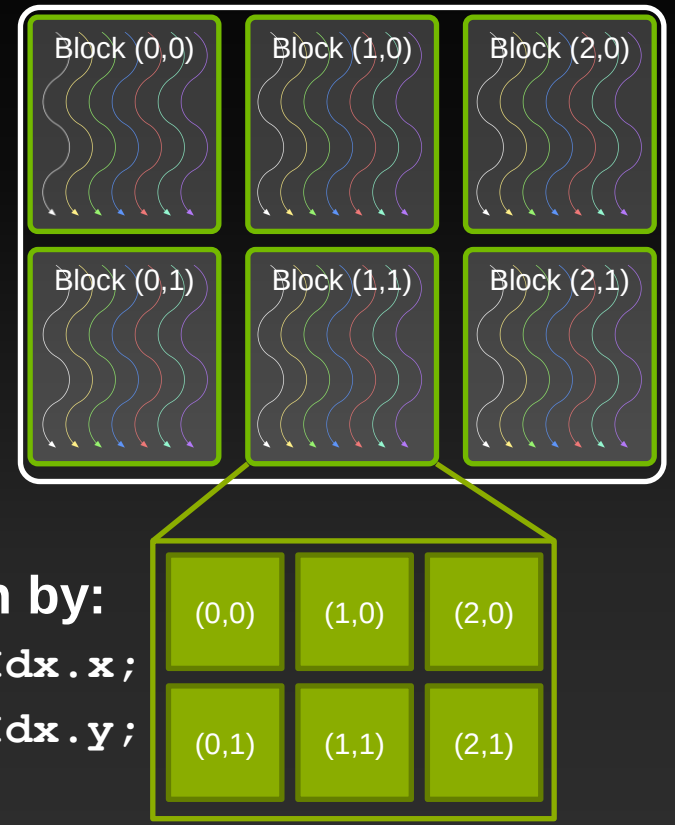


A unique index for each thread is given by:

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

Threads/Blocks/Grids

- Blocks/Grids can be 1D/2D/3D
- 2D blocks
  - `threadIdx.x`
  - `threadIdx.y`
- 2D grid
  - `blockIdx.x` and `blockDim.x`
  - `blockIdx.y` and `blockDim.y`



A unique coordinate for each thread is given by:

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

How to parallelize update_v

- We need lots of parallelism to fill the GPU and hide latencies
- Start one thread per cell in the domain

update_v kernel – CUDA C

```
void update_v(double* v, double* u,
    int nx, int ny, double dt,
    double dx2, double dy2)
{
    for (int y=1; y<(ny-1); ++y) {
        for (int x=1; x<(nx-1); ++x) {
            v[y*nx+x] = v[y*nx+x] + dt * 100.0 * (
                      (u[(y+1)*nx+ x   ]
                - 2.0*u[y*nx+x]
                    + u[(y-1)*nx+ x   ])/dy2 +
                     (u[ y   *nx+(x+1)]
                - 2.0*u[y*nx+x]
                    + u[ y   *nx+(x-1)])/dx2 );
        }
    }
}
```

```
__global__ void update_v_kernel(
    double* v, double* u, int nx, int ny,
    double dt, double dx2, double dy2){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if( x > 0 && x<(nx-1) && y > 0 && y<(ny-1) ){
            v[y*nx+x] = v[y*nx+x] + dt * 100.0 * (
                      (u[(y+1)*nx+ x   ]
                - 2.0*u[y*nx+x]
                    + u[(y-1)*nx+ x   ])/dy2 +
                     (u[ y   *nx+(x+1)]
                - 2.0*u[y*nx+x]
                    + u[ y   *nx+(x-1)])/dx2 );
        }
    }
}
```

update_v kernel – CUDA Fortran

```
subroutine update_v(v,u,nx,ny,dt,dx2,dy2)
  implicit none
  real*8, dimension(nx*ny) :: v
  real*8, dimension(nx*ny) :: u
  integer :: nx, ny, x, y
  real*8 :: dt, dx2, dy2
  do y=2, ny-1
    do x=2,nx-1
      v((y-1)*nx + x) = v((y-1)*nx + x)+&
        dt * 100.0 * (&
        (u((y+1-1)*nx+x)-2.0*u((y-1)*nx+x)&
        +u((y-1-1)*nx+x))/dy2 + &
        (u((y-1)*nx+(x+1))-2.0*u((y-1)*nx+x)&
        + u((y-1)*nx+(x-1)))/dx2 )
    end do
  end do
end subroutine update_v
```

```
attributes(global) &
subroutine update_v(u,v,nx,ny,dt,dx2,dy2)
  implicit none
  real*8, dimension(nx*ny) :: v
  real*8, dimension(nx*ny) :: u
  integer :: nx, ny, x, y
  real*8 :: dt, dx2, dy2
  x = (blockIdx%x-1)*blockDim%x + threadIdx%x
  y = (blockIdx%y-1)*blockDim%y + threadIdx%y
  if( (x .ge. 2) .and. (x .le. nx-1) .and. &
      (y .ge. 2) .and. (y .le. ny-1)) then
    v((y-1)*nx + x) = v((y-1)*nx + x)+&
        dt * 100.0 * (&
        (u((y+1-1)*nx+x)-2.0*u((y-1)*nx+x)&
        +u((y-1-1)*nx+x))/dy2 + &
        (u((y-1)*nx+(x+1))-2.0*u((y-1)*nx+x)&
        + u((y-1)*nx+(x-1)))/dx2 )
  endif
end subroutine update_v
```

update_v kernel – How to launch?

- **Kernels are launched with**

  `<<<,>>>`

- **Launch configuration is passed with**

  `<<<dimGrid,dimBlock>>>`

- **Use**

```
struct dim3 {
    unsigned int x,y,z;
}
```

  to define launch configuration.

```
void update_v(double* v, double* u,
    int nx, int ny, double dt,
    double dx2, double dy2) {
    double* ud, vd;
    dim3 dimBlock(16,16);
    dim3 dimGrid((nx/dimBlock.x)+1,
                 (ny/dimBlock.y)+1);
    update_v_kernel<<<dimGrid,dimBlock>>>
        (v,u,nx,ny,dt,dx2,dy2);
}
```

CUDA Fortran syntax

```
subroutine update_v(u, v, nx, ny, dt, dx2, dy2)
  implicit none
  real*8, dimension(nx*ny), device :: v
  real*8, dimension(nx*ny), device :: u
  integer :: nx, ny, x, y, istat
  real*8 :: dt, dx2, dy2
  type(dim3) :: dimGrid, dimBlock
  dimBlock = dim3(16, 16, 1)
  dimGrid  = dim3((nx/dimBlock%x)+1,(ny/dimBlock%y)+1, 1)
  call update_v_kernel<<<dimGrid, dimBlock>>>(u,v,nx,ny,dt,dx2,dy2)
end subroutine update_v
```

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()`

update_v kernel – Memory Management

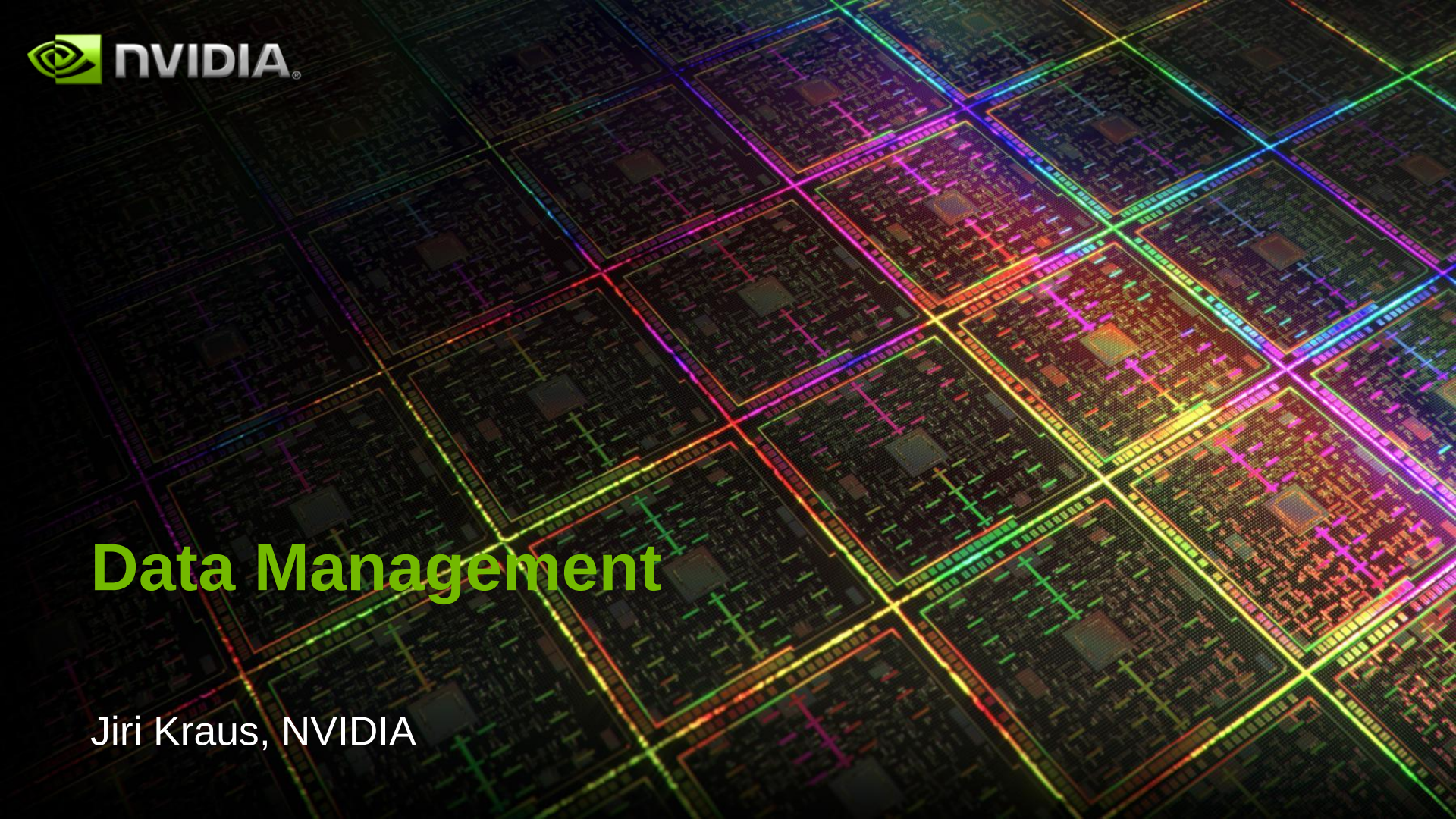
```
void update_v(double* v, double* u,
    int nx, int ny, double dt,
    double dx2, double dy2) {
    double* ud, vd;
    cudaMalloc((void**)&ud,nx*ny*sizeof(double));
    cudaMalloc((void**)&vd,nx*ny*sizeof(double));
    cudaMemcpy(ud,u,nx*ny*sizeof(double),
               cudaMemcpyHostToDevice);
    cudaMemcpy(vd,v,nx*ny*sizeof(double),
               cudaMemcpyHostToDevice);
    dim3 dimBlock(16,16);
    dim3 dimGrid((nx/dimBlock.x)+1,
                 (ny/dimBlock.y)+1);
    update_v_kernel<<<dimGrid,dimBlock>>>
        (vd,ud,nx,ny,dt,dx2,dy2);
    cudaMemcpy(v,vd,nx*ny*sizeof(double),
               cudaMemcpyDeviceToHost);
    cudaFree( ud );cudaFree( vd );
}
```

```
subroutine update_v( v, u, nx, ny, dt, dx2, dy2)
  use cudafor
  implicit none
  real*8, dimension(nx*ny) :: v
  real*8, dimension(nx*ny) :: u
  integer :: nx, ny, x, y, istat
  real*8 :: dt, dx2, dy2
  real*8, dimension(nx*ny), device :: d_u
  real*8, dimension(nx*ny), device :: d_v
  type(dim3) :: dimGrid, dimBlock
  d_u = u
  d_v = v
  dimBlock = dim3(16, 16, 1);
  dimGrid  = dim3((nx/16)+1,(ny/16)+1, 1);
  call update_v_kernel<<<dimGrid, dimBlock>>> &
   (d_u,d_v,nx,ny,dt,dx2,dy2)
  u = d_u
  v = d_v
end subroutine update_v
```

Parallelize -Task 2: update_v

Follow TODOs in `task2/euler.cu - task2/euler.CUF`

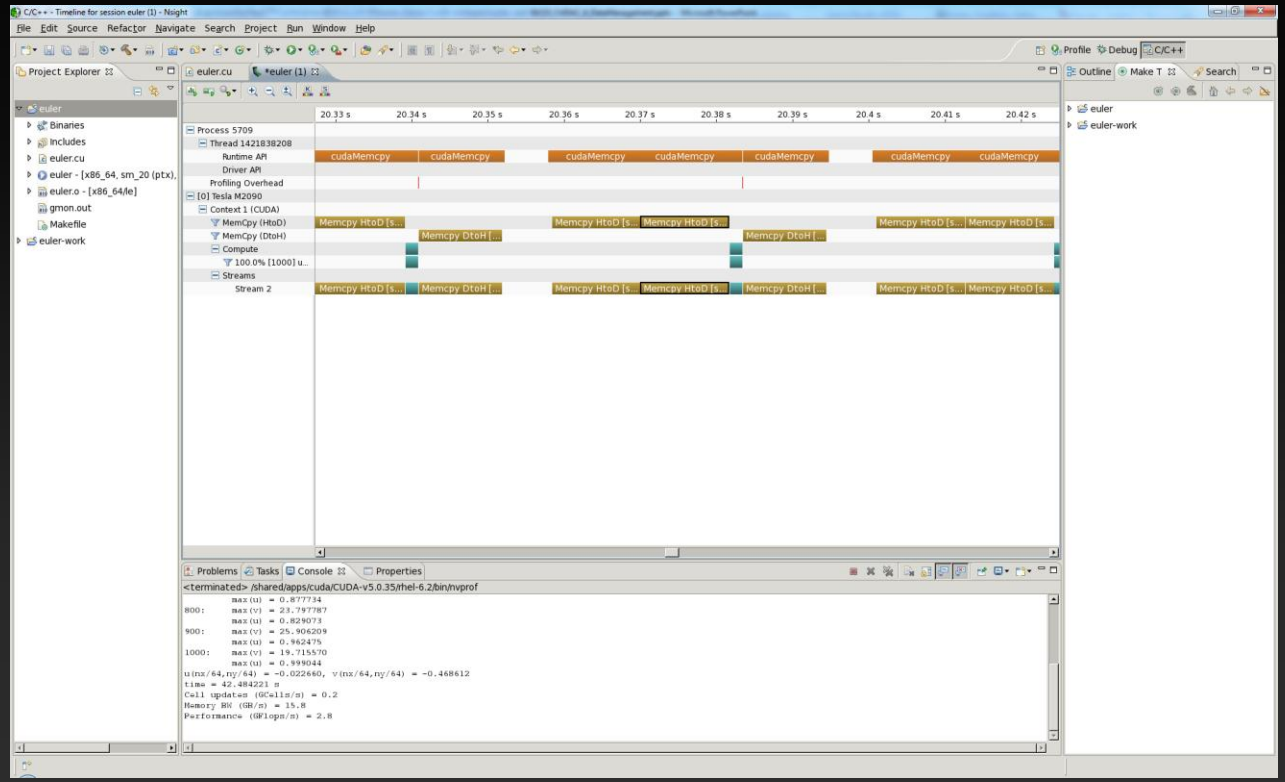
- **Initialize/Shutdown CUDA:**
  - `cudaSetDevice( device-ordinal )` / **use cudafor**
  - `cudaDeviceReset()`
- **Memory management:**
  - `cudaMalloc()` /, **device** ,`cudaFree()`,`cudaMemcpy()` / **=**
- **Kernel Launch**
  - `__global__` / **attributes(global)**
  - `dim3 dimBlock, dimGrid;` / **type(dim3)**
  - `<<<dimGrid,dimBlock>>>`



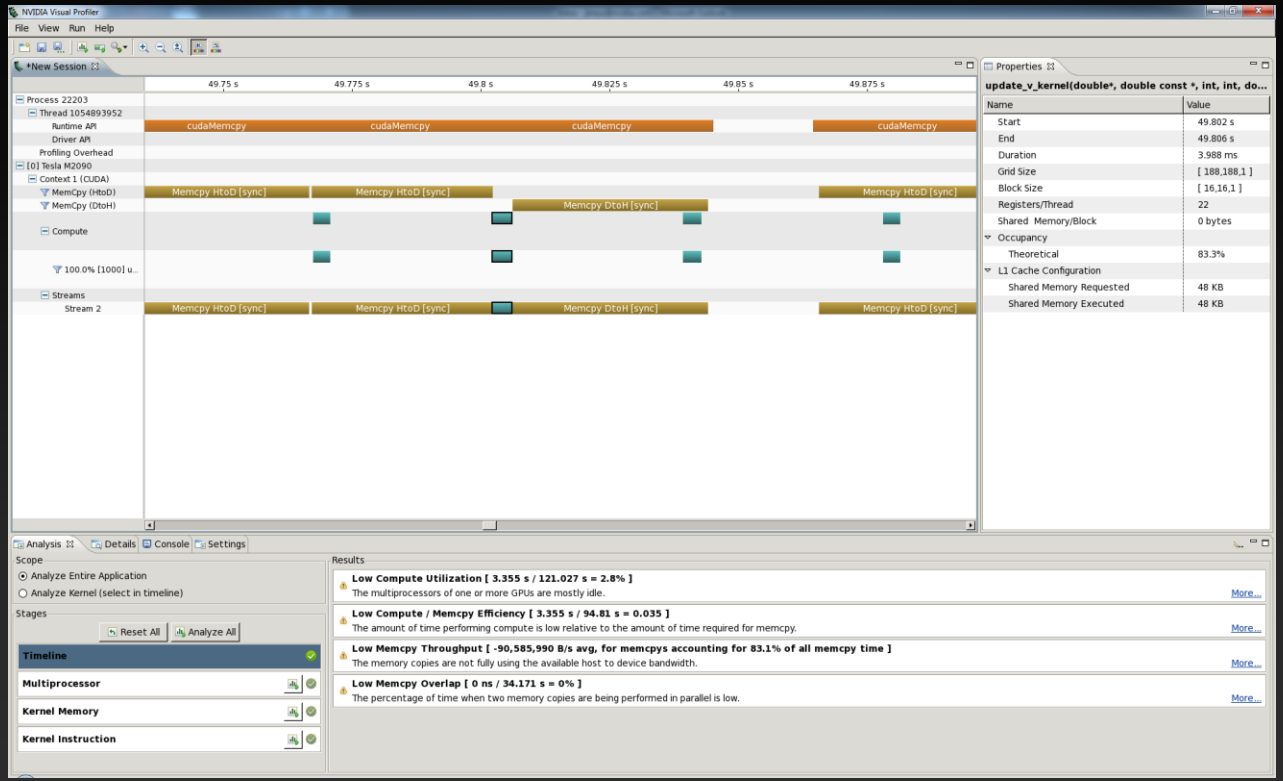
Data Management

Jiri Kraus, NVIDIA

Assess with Nsight Eclipse edition (demo)



Assess with NVIDIA Visual Profiler (demo)



Assess

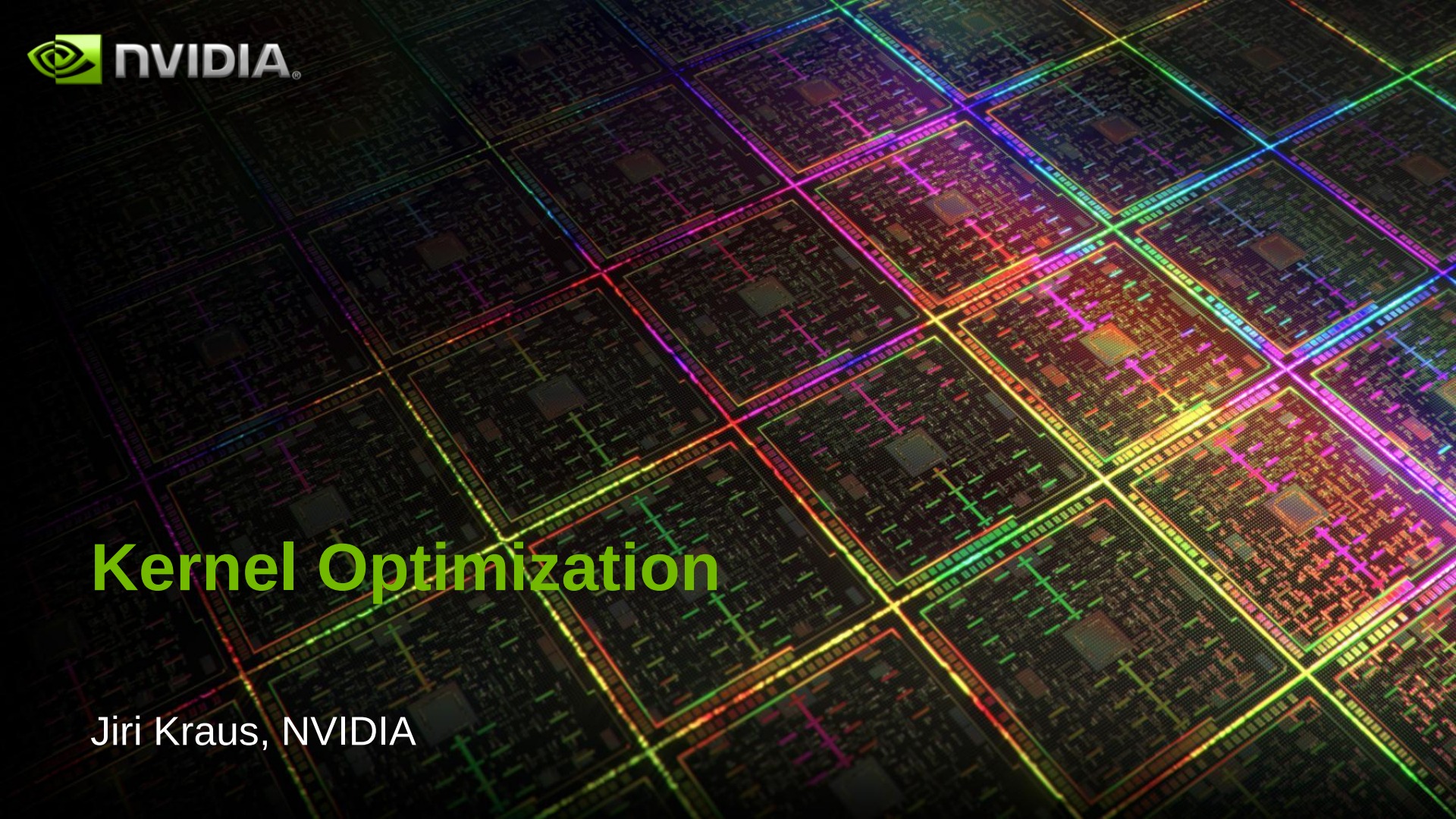
- Why is CUDA version with explicit data management slower than expected?
  - Most of the time is spend in memory copies
- Remedy:
  - Copy `u` and `v` only once before and after the time loop
  - Need to also use CUDA for `update_u` and `calc_bc`
  - As an intermediate step leave `max_value` on the CPU and copy `u` and `v` every 100 iterations to the CPU.

Parallelize - Task 3: update_u, calc_bc

Follow TODOs in `task3/euler.cu task3/euler.CUF`:

- **Copy `u` and `v` to GPU before loop over time steps**
- **Copy `u` and `v` to host every 100 iterations for max calculation**
- **Copy `u` and `v` to host after loop over time steps**
- **Parallelize `update_u` with CUDA (`calc_bc` is already done in `task3/euler.cu task3/euler.CUF`)**

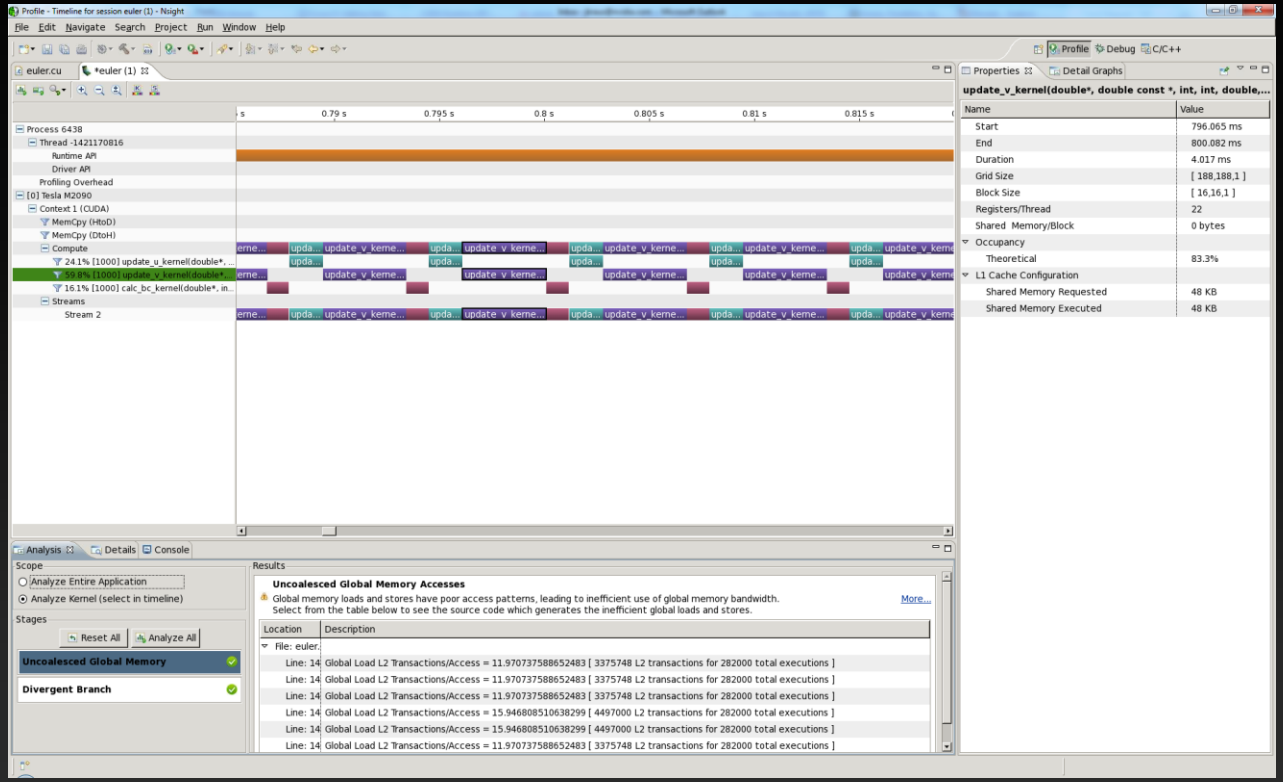
- **Memory management:**
  - `cudaMalloc()` / `device,allocatable`, `cudaFree()`, `cudaMemcpy()`
- **Kernel Launch**
  - `dim3 dimBlock, dimGrid; / type(dim3)`
  - `<<<dimGrid,dimBlock>>>`



Kernel Optimization

Jiri Kraus, NVIDIA

Assess (demo)

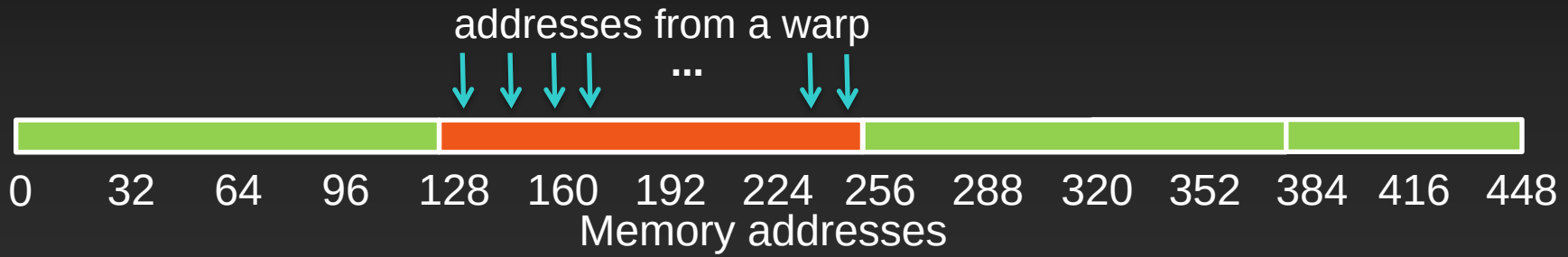


Assess

- Analyze longest running Kernel (`update_v`)
- Why is does `update_v` has so many uncoalesced memory accesses?
  1. Access to left (x-1) and right (x-1) neighbors does not always align with cache line boundaries.
     - Not much we can do about his in this tutorial
  2. The accesses to the elements of all rows except the first row are not aligned to cache line boundaries.
- Remedy: Pad rows to align all rows of the domain to fix 2

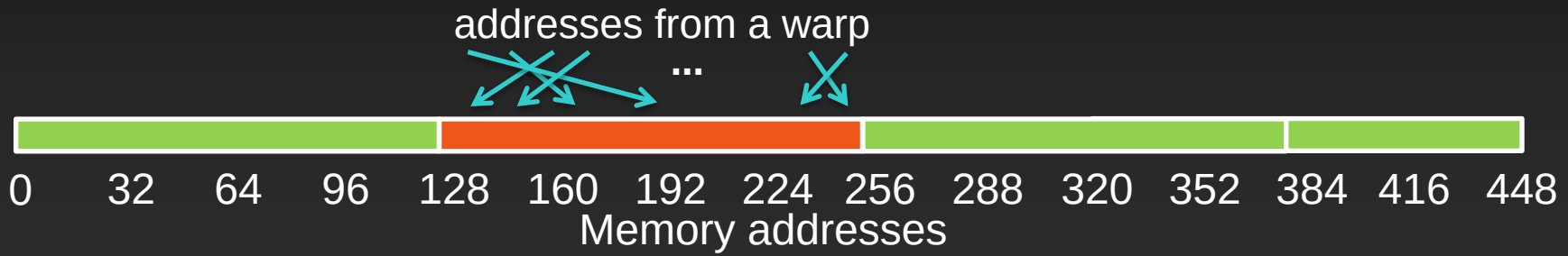
Memory coalescing

- **Scenario:**
  - **Warp requests 32 aligned, consecutive 4-byte words**
- **Addresses fall within 1 cache-line**
  - **Warp needs 128 bytes**
  - **128 bytes move across the bus**
  - **Bus utilization: 100%**



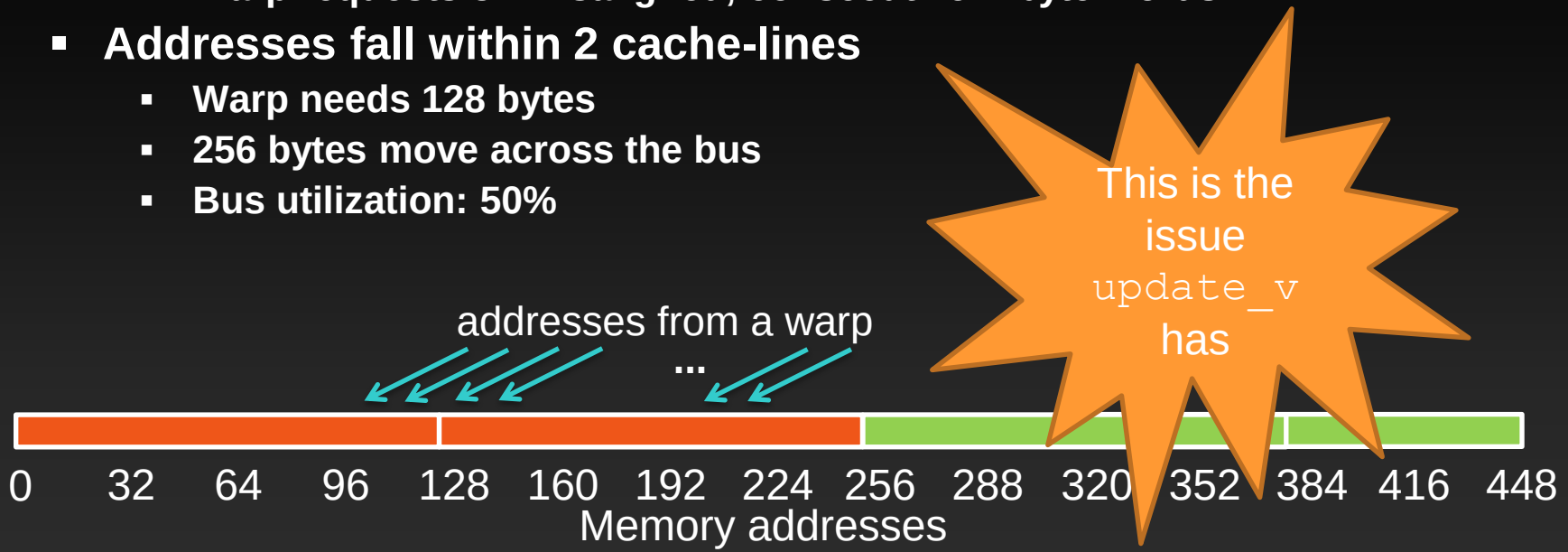
Memory coalescing

- **Scenario:**
  - **Warp requests 32 aligned, permuted 4-byte words**
- **Addresses fall within 1 cache-line**
  - **Warp needs 128 bytes**
  - **128 bytes move across the bus**
  - **Bus utilization: 100%**



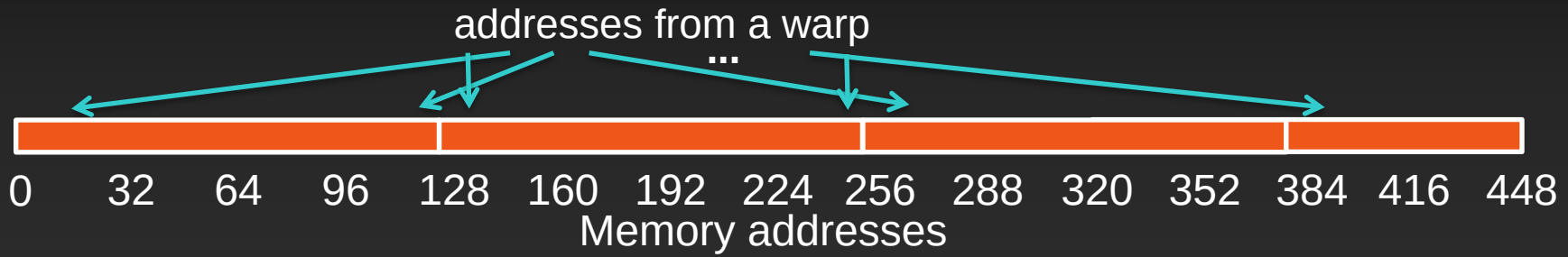
Memory coalescing

- **Scenario:**
  - **Warp requests 32 misaligned, consecutive 4-byte words**
- **Addresses fall within 2 cache-lines**
  - **Warp needs 128 bytes**
  - **256 bytes move across the bus**
  - **Bus utilization: 50%**



Memory coalescing

- **Scenario:**
  - **Warp requests 32 scattered 4-byte words**
- **Addresses fall within *N* cache-lines**
  - **Warp needs 128 bytes**
  - ***N**128 bytes move across the bus on a miss**
  - **Bus utilization: 128 / (*N**128)**

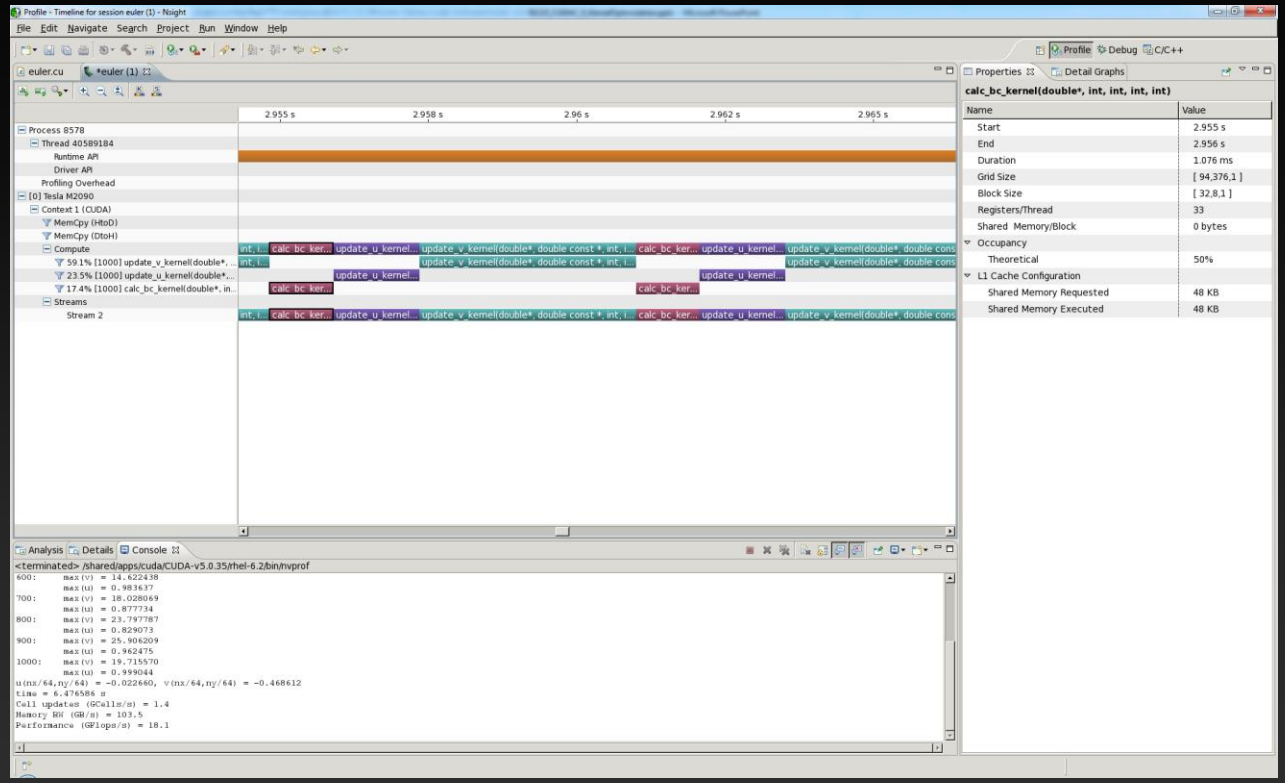


Optimize - Task 4: memory coalescing

Follow TODOs in `task4/euler.cu task4/euler.CUF`:

- Pad rows of `u` and `v` to align all rows of the domain to 128 byte
  - for simplicity just change nx to an appropriate value

Assess (demo)



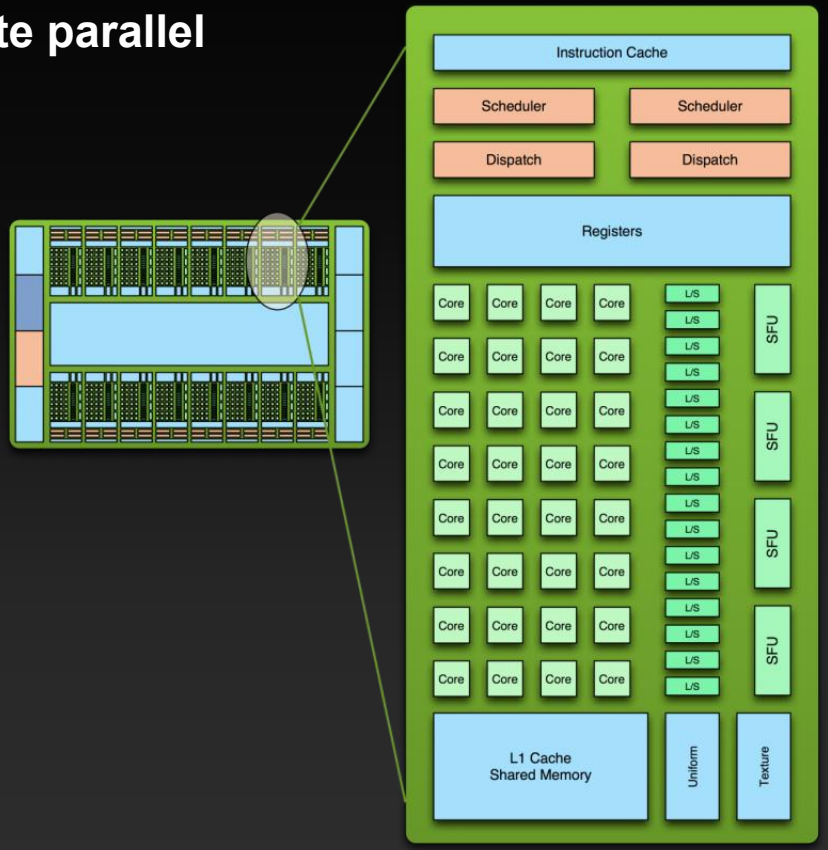
Assess

- Why is the occupancy of `update_v` and `calc_bc` not 100%?
  - Suboptimal launch configuration
- Remedy:
  - Use CUDA Occupancy calculator to derive optimal launch configuration for all 3 Kernels

20-Series Architecture

- 512 ***Scalar Processor (SP) cores*** execute parallel thread instructions

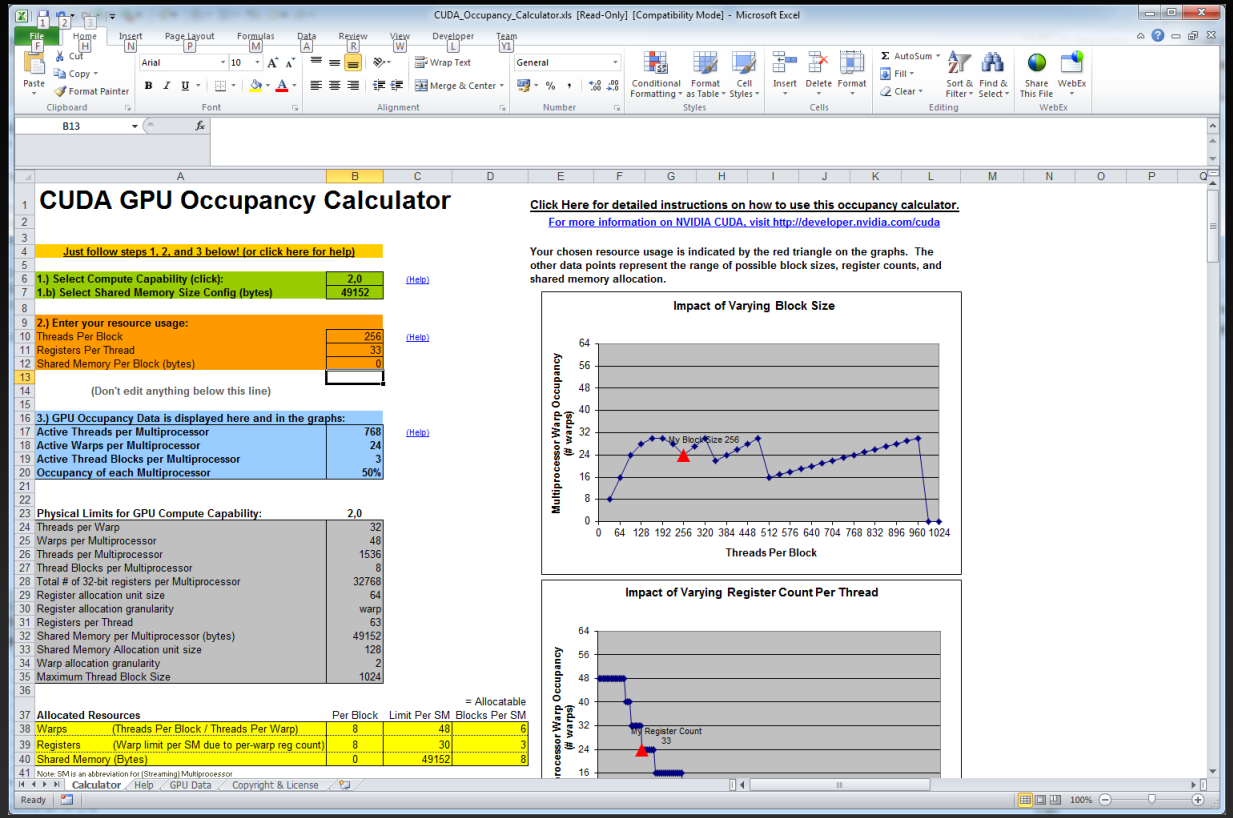
- 16 **Streaming *Multiprocessors (SMs)*** each contains
  - 32 scalar processors
    - 32 fp32 / int32 ops / clock,
    - 16 fp64 ops / clock
  - 4 Special Function Units (SFUs)
  - Shared register file (128KB)
  - ***48 KB / 16 KB Shared memory***
  - 16KB / 48 KB L1 data cache



Occupancy

- We need as much threads a possible on deck to hide latencies
- SM resources limit the maximum number of threads on deck:
  - Maximum number of threads: 1536
    - If we hit this limit we have an optimum of 100% theoretical occupancy
  - Maximum number of thread blocks: 8
    - Thread blocks that are smaller than 1536/8 = 192 can hit this limit
  - Register file size: 32768 32bit registers
    - This is often limiting the occupancy
  - Shared memory: 48kb
    - We do not use shared memory so this limit does not apply for us
- Use CUDA Occupancy calculator to maximize occupancy

CUDA Occupancy calculator (demo)



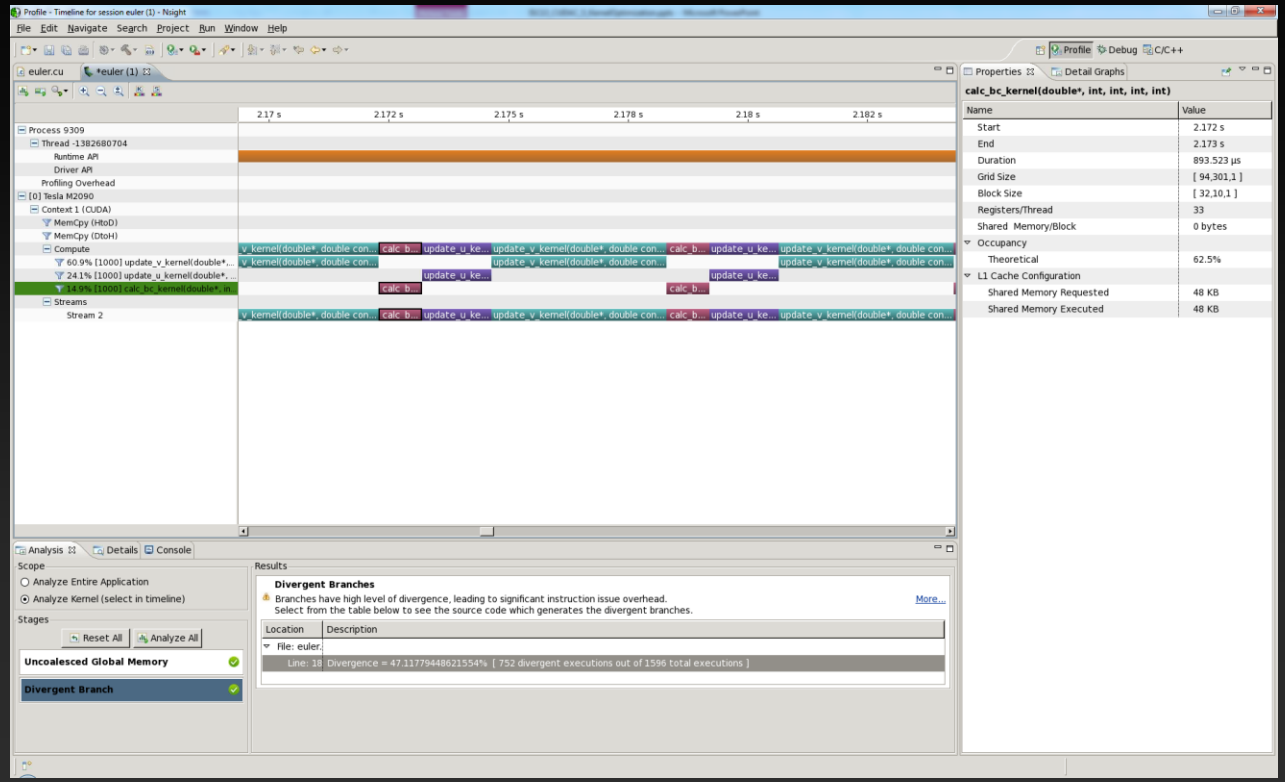
Optimize - Task 5: launch configuration

Follow TODOs in `task5/euler.cu task5/euler.CUF`:

- Optimize thread block sizing of
  - `update_v_kernel`
  - `calc_bc`
- Use NVIDIA Visual Profiler or Nsight Eclipse edition profiler to determine current theoretical occupancy, block size and used 32bit registers
- Use the CUDA Occupancy calculator to calculate the optimal thread block sizing for all kernels

  `/usr/local/cuda/tools/CUDA_Occupancy_Calculator.xls`

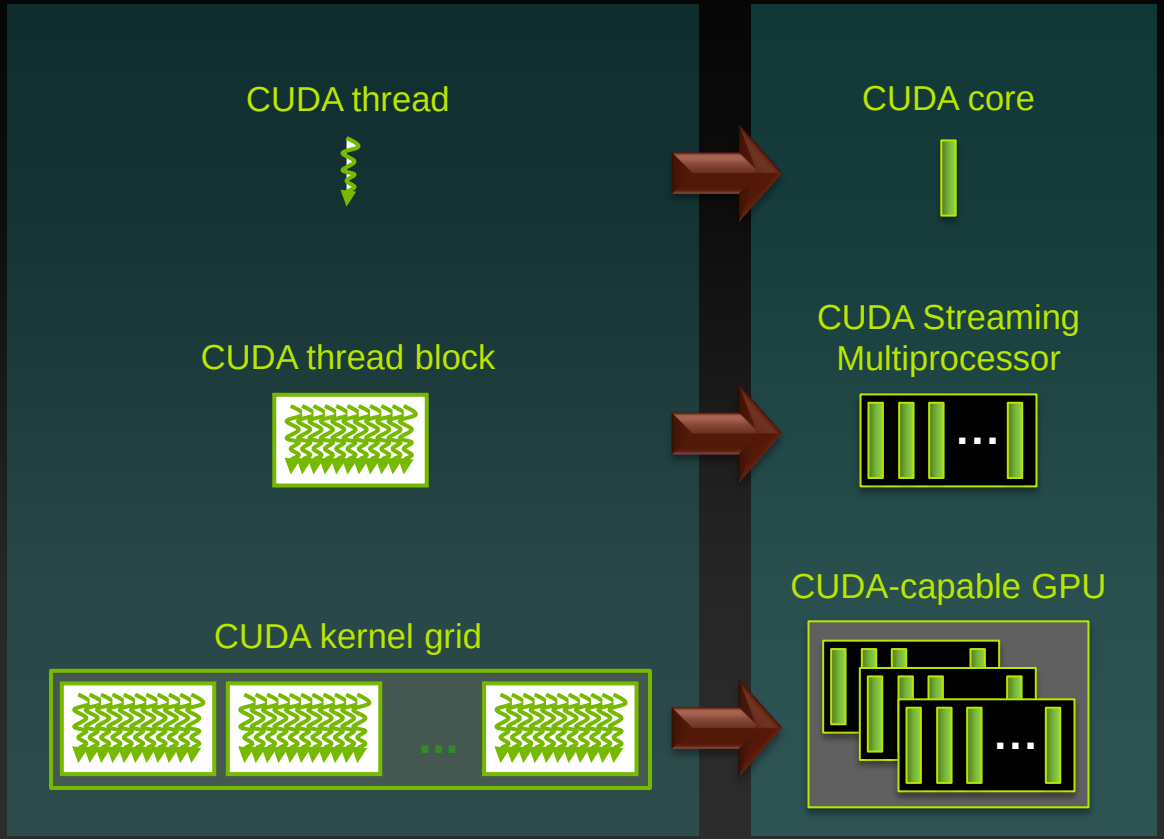
Assess (demo)



Assess

- `update_v` does not have memory or coalescing issue that we can address in this tutorial
- `update_u` does not have memory or coalescing issue
- Why does `calc_c` have so many divergent branches?
  - Suboptimal organization of if clauses
  - Many threads are stared that have nothing to
- Remedy:
  - Process boundary with a 1D thread layout and let each thread handle 4 elements.

Kernel Execution



- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Warps

A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor

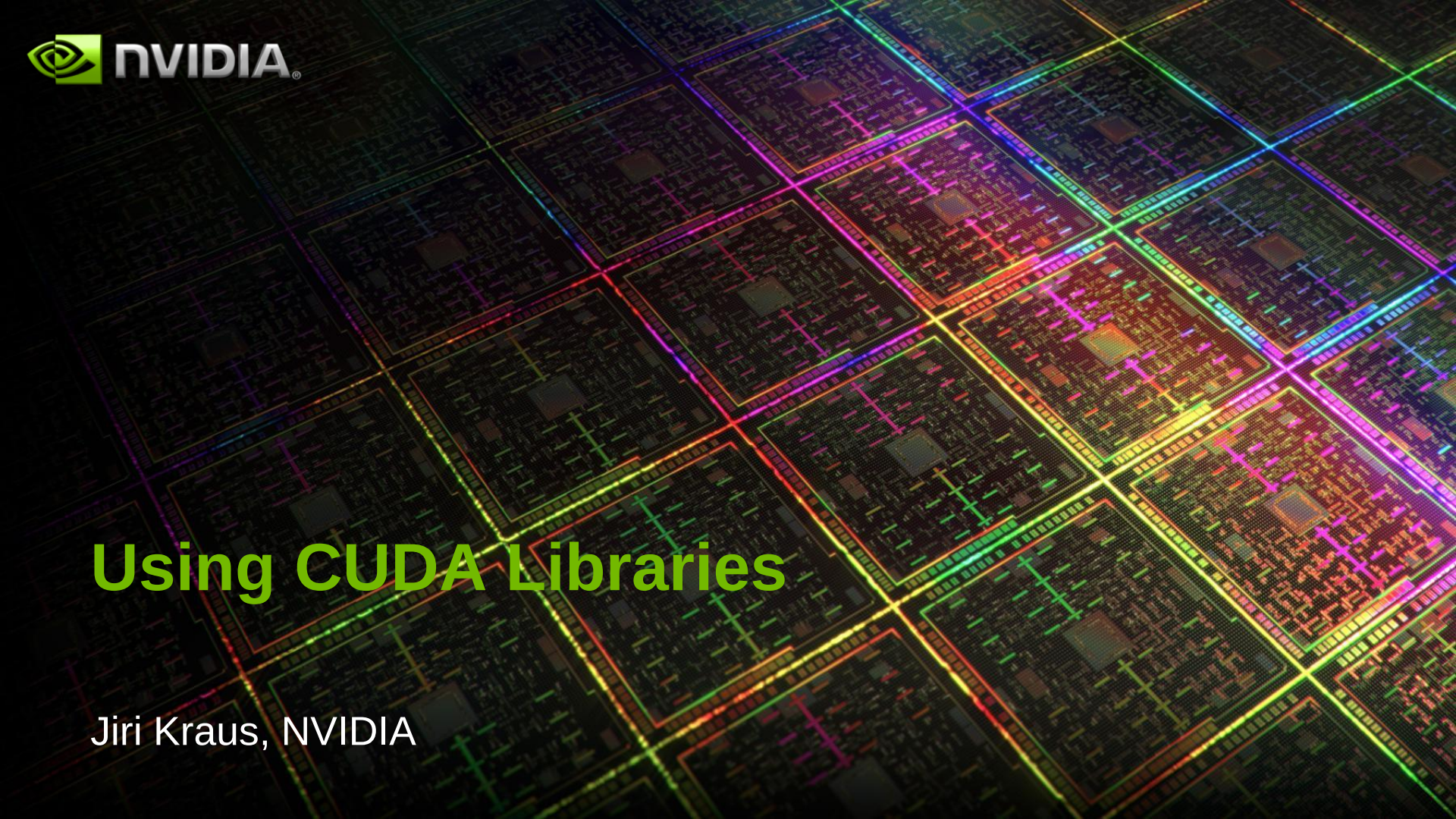
Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance
- Avoid diverging within a warp
  - Example with divergence:
    - `if (threadIdx.x > 2) {...} else {...}`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

Optimize - Task 6: clac_bc

Follow TODOs in `task5/euler.cu`:

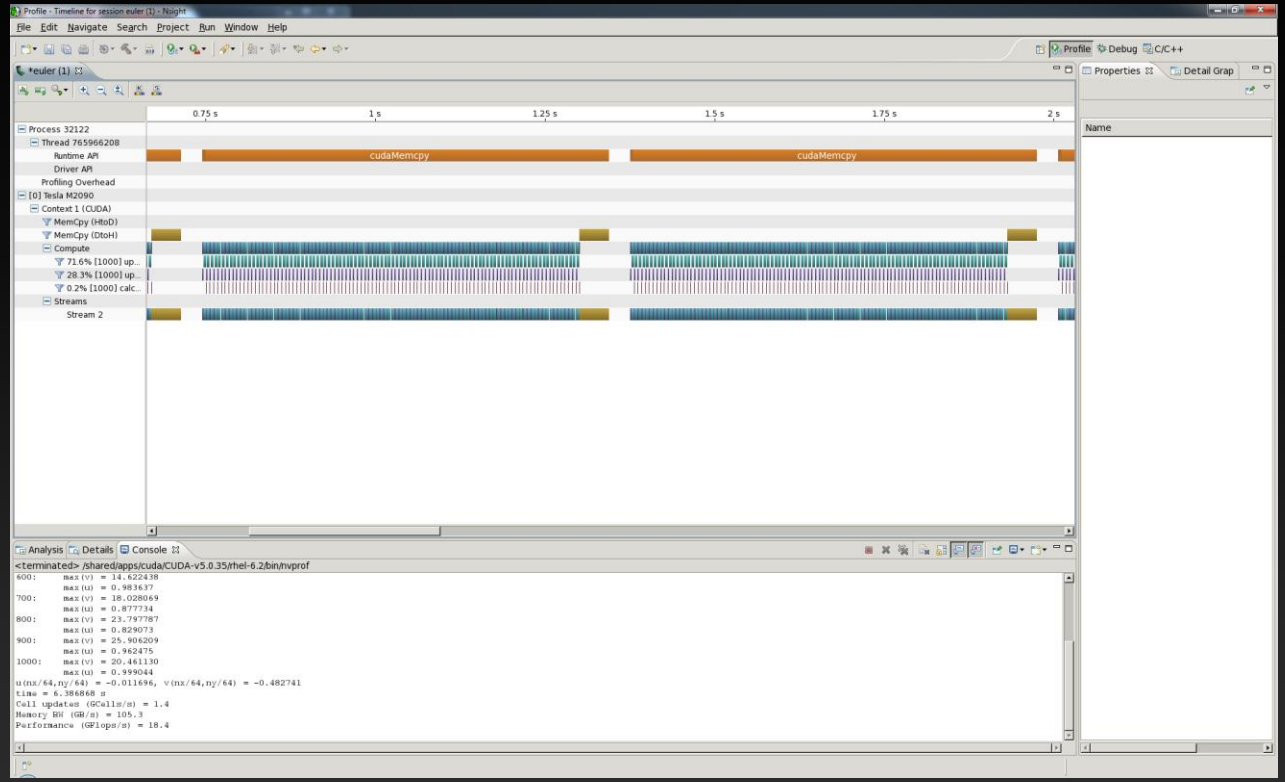
- Use a 1D block and grid instead of the 2D block and grid
- Start only max((nx/4) , (ny/4)) threads in total
  - Process 4 elements per thread



Using CUDA Libraries

Jiri Kraus, NVIDIA

Assess (demo)



Assess

- Analyze timeline
- GPU is not utilize all of the time
  - Idle while CPU calculates `max_value`
  - Idle while `u` and `v` are copied to host
- Remedy: Calculate `max_value` on the GPU with CUBLAS NVIDIAs Basic Linear Algebra Subprogramms (BLAS) implementation.

Using CUDA Libraries

- CUDA Libraries can be used like CPU libraries
  - Depending on library design host or device pointers need to be passed to libraries functions.
- Using CUBLAS as an example
  - Use IDAMAX to calculate `max_value` on the GPU for `u` and `v`

How to use CUBLAS?

```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
```

How to use CUBLAS?

```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

Add “cublas” prefix and use device variables

How to use CUBLAS?

```
int N = 1 << 20;
cublasInit();
```

Initialize CUBLAS

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

```
cublasShutdown();
```

Shut down CUBLAS

How to use CUBLAS?

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);
```

Allocate device vectors

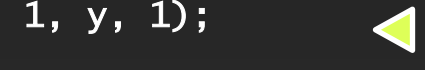
```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

```
cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Deallocate device vectors

How to use CUBLAS?

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```



Calling cuBLAS from CUDA Fortran

```
program cublasTest
  use cublas
  implicit none

  real, allocatable :: a(:,:),b(:,:),c(:,:)
  real, device, allocatable :: a_d(:,:),b_d(:,:),c_d(:,:)
  integer :: k=4, m=4, n=4
  real :: alpha=1.0, beta=2.0

  allocate(a(m,k), b(k,n), c(m,n), a_d(m,k), b_d(k,n), c_d(m,n))

  a = 1; a_d = a
  b = 2; b_d = b
  c = 3; c_d = c

  call cublasSgemm('N', 'N', m , n, k, alpha, a_d, m, b_d, k, beta, c_d, m)

  c=c_d

  deallocate(a, b, c, a_d, b_d, c_d)

end program cublasTest
```

Parallelize - Task 7: max_value

Follow TODOs in `task7/euler.cu/CUF task7/Makefile`:

- **Use CUBLAS IDAMAX instead of `max_value` to calculate the maximum absolute value of `u` and `v` and only download these values**
  - **Link CUBLAS `-lcublas`**
  - **Initialize CUBLAS**
    - Include CUBLAS header `cublas_v2.h` (CUDA C)
    - Call `cublasCreate` (CUDA C)
    - `use cublas` (CUDA Fortran)
  - **Call IDAMAX:** `culasIdamax`
  - **Download elements of `u` and `v` with maximum value**
  - **Finalize CUBLAS** `cublasDestroy` **(CUDA C)**

Conclusion

- Know how to write a CUDA Kernel
- Know how to manage device memory and host to device memory transfers
- Know how to use NVIDIA Tools to identify performance limiters and how to fix them

Thank You!