www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introduction to CUDA: The Basics

Antonio J. Peña

Based on material from NVIDIA's GPU Teaching Kit

Barcelona, April 8 2019

Agenda

- Day 1
  - 9:00 The Basics
  - 10:45 Coffee Break
  - 11:15 Parallelism Model
  - 13:00 Lunch Break
  - 14:00 Memory and Data Locality
  - 15:45 Coffee Break
  - 16:15 Hands-on
  - 18:00 Adjourn

Agenda

- Day 2
  - 9:00 Efficiency and Performance Considerations
  - 10:45 Coffee Break
  - 11:15 Atomics and Histogramming, Reductions
  - 13:00 Lunch Break
  - 14:00 Architectural Considerations
  - 14:45 Efficient Host-Device Data Transfers
  - 15:45 Coffee Break
  - 16:15 Hands-on
  - 18:00 Adjourn

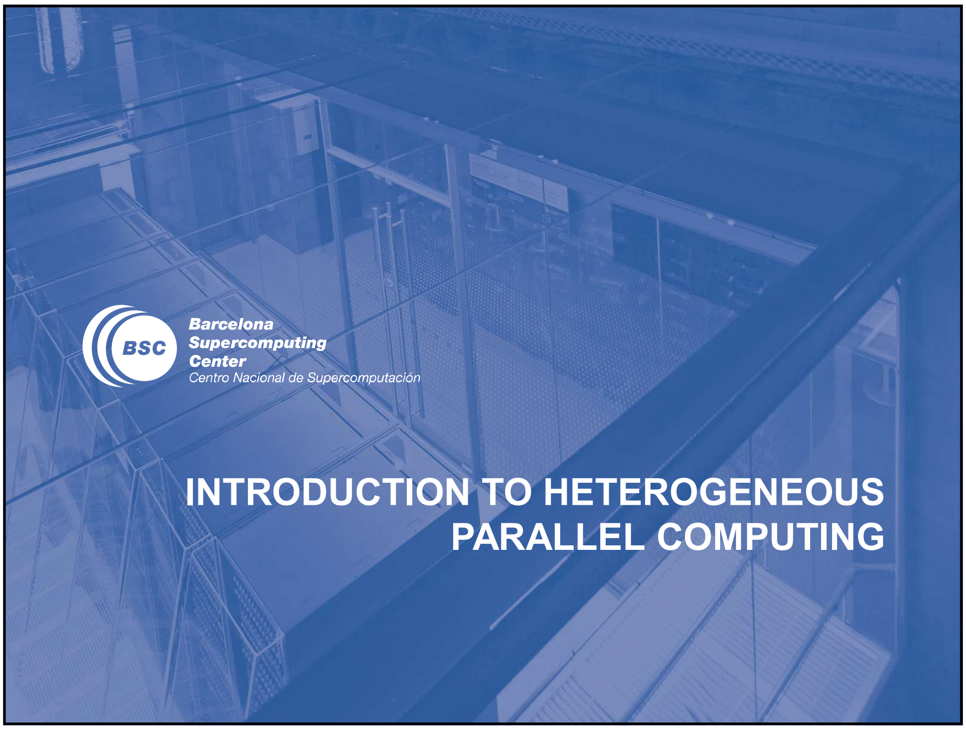
Agenda

- Day 3
  - 9:00 OpenACC and Other Approaches to GPU Computing
  - 10:45 Coffee Break
  - 11:15 Volta & CUDA 10
  - 13:00 Lunch Break
  - 14:00 Hands-on
  - 18:00 Adjourn

Agenda

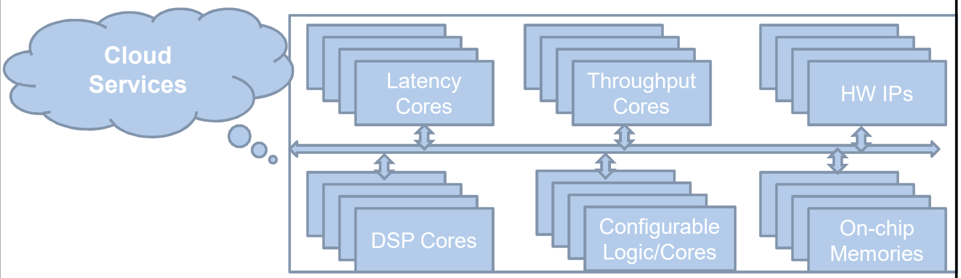
- Day 4
  - 9:00 Hands-on
  - 10:45 Coffee Break
  - 11:15 Hands-on
  - 13:00 Lunch Break
  - 14:00 Open Labs
  - 18:00 Adjourn

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

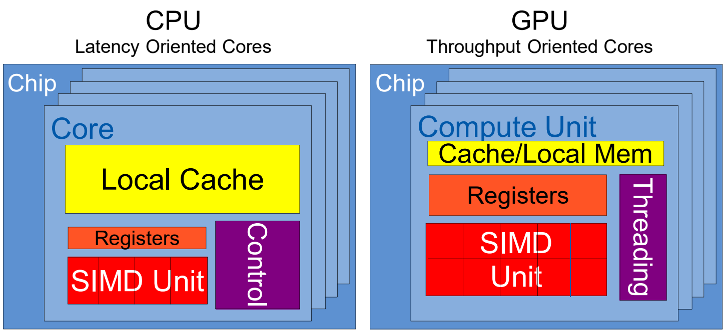


Heterogeneous Parallel Computing

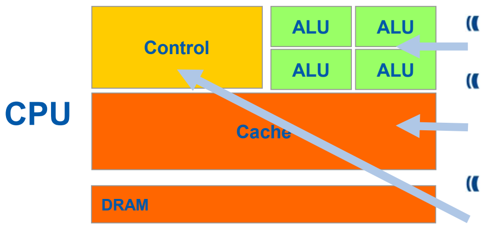
– Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently



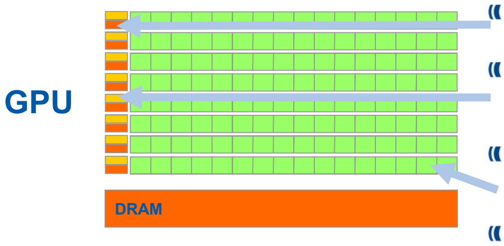
CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

GPUs: Throughput Oriented Design



- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

Heterogeneous Parallel Computing in Many Disciplines

- Financial Analysis
- Scientific Simulation
- Engineering Simulation
- Data Intensive Analytics
- Medical Imaging
- Digital Audio Processing
- Digital Video Processing
- Computer Vision
- Biomedical Informatics
- Electronic Design Automation
- Statistical Modeling
- Numerical Methods
- Ray Tracing Rendering
- Interactive Physics

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

PROGRAMMING ALTERNATIVES TO USE THE GPU

3 Ways to Accelerate Applications

Applications		
Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code	Most Performance Most Flexibility

Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

GPU Accelerated Libraries

Linear Algebra FFT, BLAS, SPARSE, Matrix	NVIDIA cuFFT, cuBLAS, cuSPARSE · CULA tools · MAGMA · CUSP
Numerical & Math RAND, Statistics	NVIDIA Math Lib · ArrayFire · NVIDIA cuRAND
Data Struct. & AI Sort, Scan, Zero Sum	Thrust · GPU AI - Board Games · GPU AI Path Finding
Visual Processing Image & Video	NVIDIA NPP · Sundog Software

Vector Addition in Thrust

```
thrust::device_vector<float> deviceInput1(inputLength);
thrust::device_vector<float> deviceInput2(inputLength);
thrust::device_vector<float> deviceOutput(inputLength);

thrust::copy(hostInput1, hostInput1 + inputLength,
    deviceInput1.begin());
thrust::copy(hostInput2, hostInput2 + inputLength,
    deviceInput2.begin());

thrust::transform(deviceInput1.begin(), deviceInput1.end(),
     deviceInput2.begin(), deviceOutput.begin(),
     thrust::plus<float>());
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenACC

– Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
   copyout(output[0:inputLength])
  for(i = 0; i < inputLength; ++i) {
     output[i] = input1[i] + input2[i];
  }
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose:** The programmer often needs to express more details

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran
C	CUDA C
C++	CUDA C++
Python	PyCUDA, Copperhead, Numba
F#	Alea.cuBase

CUDA - C

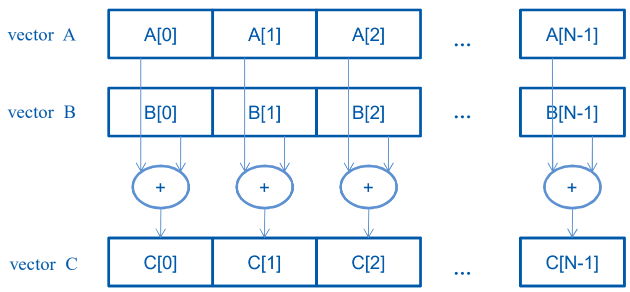
Applications

Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code	**Most Performance** **Most Flexibility**

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MEMORY ALLOCATION AND DATA MOVEMENT API FUNCTIONS

Data Parallelism - Vector Addition Example



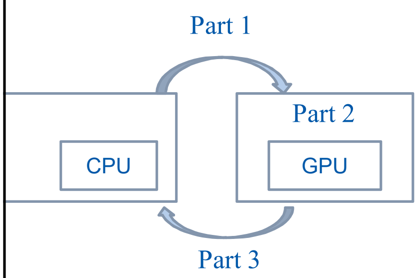
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Vector Addition – Traditional C Code

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```


Heterogeneous Computing vecAdd CUDA Host Code

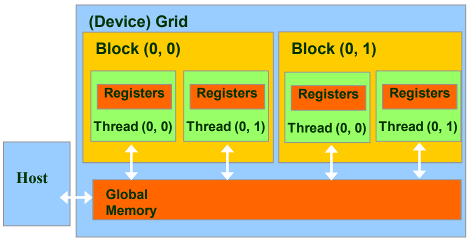



```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size = n* sizeof(float);
  float *d_A, *d_B, *d_C;
  // Part 1
  // Allocate device memory for A, B, and C
  // copy A and B to device memory

  // Part 2
  // Kernel launch code – the device performs the actual vector addition

  // Part 3
  // copy C from the device memory
  // Free device vectors
}
```

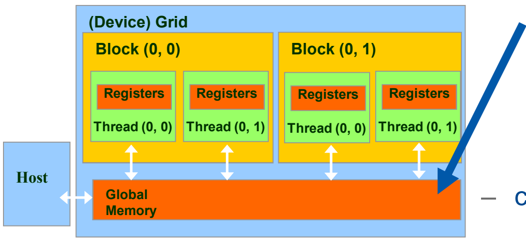
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

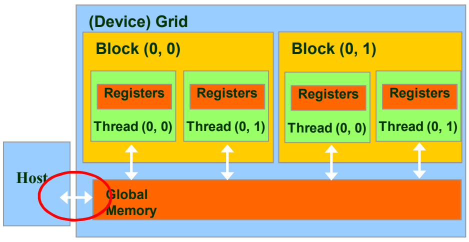
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float);
   float *d_A, *d_B, *d_C;

   cudaMalloc((void **) &d_A, size);
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &d_B, size);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &d_C, size);

   // Kernel invocation code – to be shown later

   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
   cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

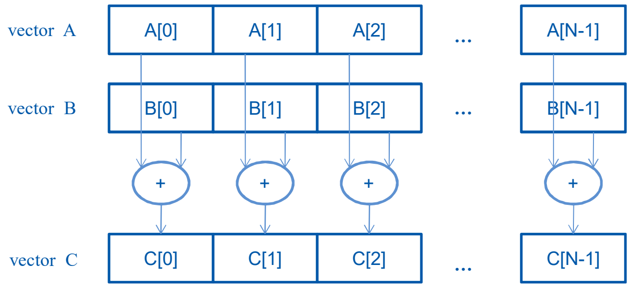
In Practice, Check for API Errors in Host Code

```
cudaError_t err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf(“%s in %s at line %d\n”,   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```


THREADS AND KERNEL FUNCTIONS

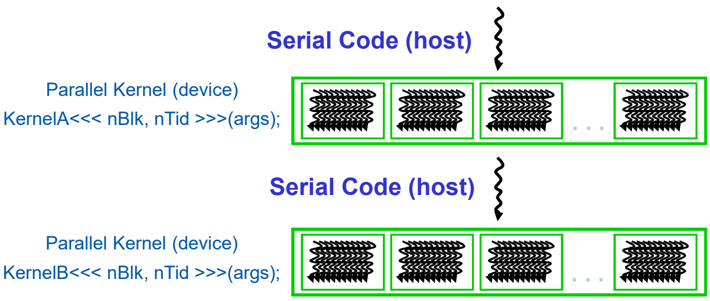
Data Parallelism - Vector Addition Example



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



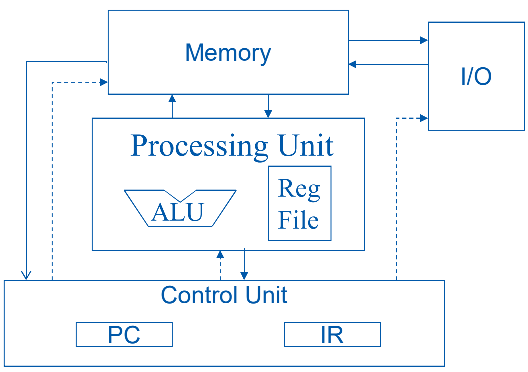
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
  - Both CPUs and GPUs are designed based on (different) instruction sets
- Program instructions operate on data stored in memory and/or registers.

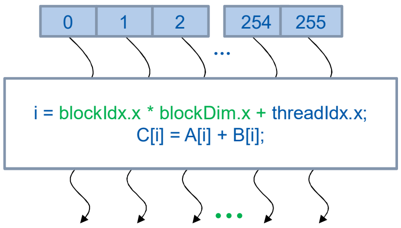
A Thread as a Von-Neumann Processor

A thread is a "virtualized" or "abstracted" Von-Neumann Processor

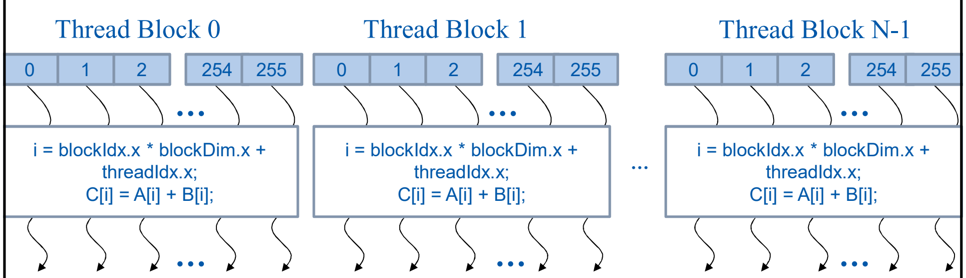


Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



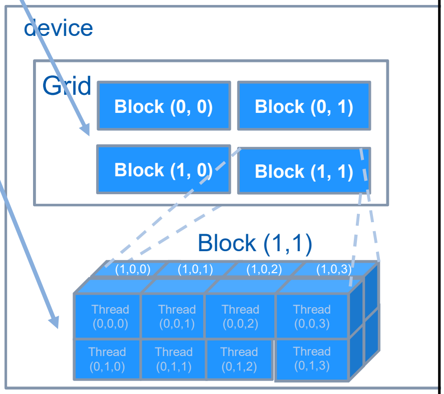
Thread Blocks: Scalable Cooperation



- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

NVCC Compiler

- NVIDIA provides a CUDA-C compiler
  - nvcc
- NVCC compiles device code then forwards code on to the host compiler (e.g. g++)
- Can be used to compile & link host only applications

Example: Hello World

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

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

Output:

```
$ nvcc main.cu
$ ./a.out
Hello World!
```

- Notes
  - `mykernel` does nothing
  - nvcc only parses .cu files for CUDA

Developer Tools - Debuggers

NSIGHT **CUDA-GDB** **CUDA MEMCHECK**

NVIDIA Provided

allinea DDT TotalView®

3rd Party

https://developer.nvidia.com/debugging-solutions

Compiler Flags

- There are two compilers being used
  - NVCC: Device code
  - Host Compiler: C/C++ code
- NVCC supports some host compiler flags
  - If flag is unsupported, use –Xcompiler to forward to host
    - e.g. –Xcompiler –fopenmp
- Debugging Flags
  - -g: Include host debugging symbols
  - -G: Include device debugging symbols
  - -lineinfo: Include line information with symbols

CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary
    %> cuda-memcheck ./exe
- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:
  - -Xcompiler -rdynamic -lineinfo

http://docs.nvidia.com/cuda/cuda-memcheck

CUDA-GDB

- cuda-gdb is an extension of GDB
  - Provides seamless debugging of CUDA and CPU code
- Works on Linux and Macintosh
  - For a Windows debugger use NSIGHT Visual Studio Edition

http://docs.nvidia.com/cuda/cuda-gdb

Example: cuda-gdb

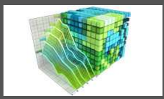
%> cuda-gdb --args ./a.out

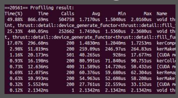
```
(cuda-gdb) b main        //set break point at main
(cuda-gdb) r                            //run application
(cuda-gdb) l                            //print line context
(cuda-gdb) b foo                    //break at kernel foo
(cuda-gdb) c                            //continue
(cuda-gdb) cuda thread              //print current thread
(cuda-gdb) cuda thread 10           //switch to thread 10
(cuda-gdb) cuda block               //print current block
(cuda-gdb) cuda block 1             //switch to block 1
(cuda-gdb) d                  //delete all break points
(cuda-gdb) set cuda memcheck on    //turn on cuda memcheck
(cuda-gdb) r                  //run from the beginning
```

http://docs.nvidia.com/cuda/cuda-gdb

Developer Tools - Profilers

NSIGHT | **NVVP** | **NVPROF**





NVIDIA Provided

TAU | **VampirTrace**

3rd Party

https://developer.nvidia.com/performance-analysis-tools

NVPROF

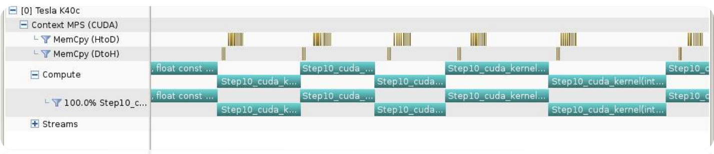
Command Line Profiler
- Compute time in each kernel
- Compute memory transfer time
- Collect metrics and events
- Support complex process hierarchy's
- Collect profiles for NVIDIA Visual Profiler
- No need to recompile

Example: nvprof

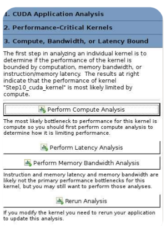
1. Collect profile information
   %> nvprof ./a.out
2. View available metrics
   %> nvprof --query-metrics
3. View global load/store efficiency
   %> nvprof --metrics gld_efficiency,gst_efficiency ./a.out
4. Store a timeline to load in NVVP
   %> nvprof –o profile.timeline ./a.out
5. Store analysis metrics to load in NVVP
   %> nvprof –o profile.metrics --analysis-metrics ./a.out

NVIDIA's Visual Profiler (NVVP)

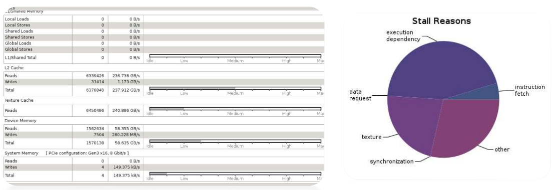
Timeline



Guided System



Analysis

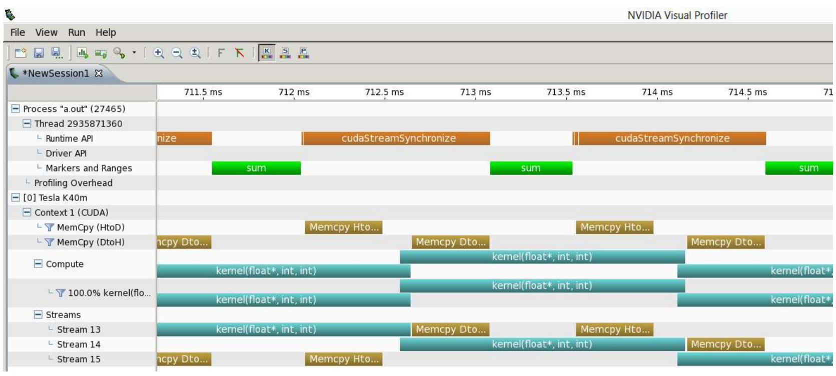


NVTX

- Our current tools only profile API calls on the host
  - What if we want to understand better what the host is doing?
- The NVTX library allows us to annotate profiles with ranges
  - Add: #include <nvToolsExt.h>
  - Link with: -lnvToolsExt
- Mark the start of a range
  - nvtxRangePushA("description");
- Mark the end of a range
  - nvtxRangePop();
- Ranges are allowed to overlap

http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/

NVTX Profile



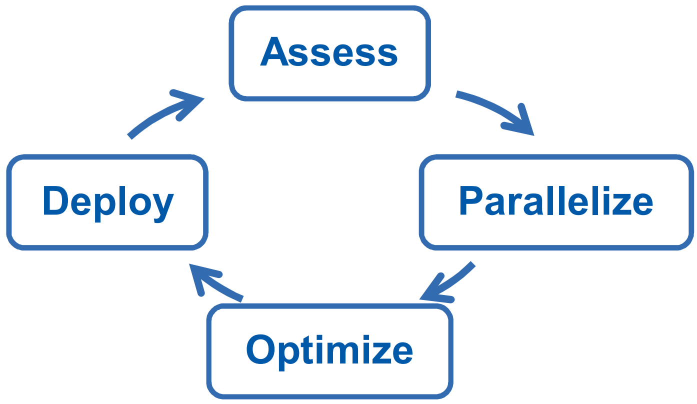
NSIGHT

- CUDA enabled Integrated Development Environment
  - Source code editor: syntax highlighting, code refactoring, etc
  - Build Manger
  - Visual Debugger
  - Visual Profiler
- Linux/Macintosh
  - Editor = Eclipse
  - Debugger = cuda-gdb with a visual wrapper
  - Profiler = NVVP
- Windows
  - Integrates directly into Visual Studio
  - Profiler is NSIGHT VSE

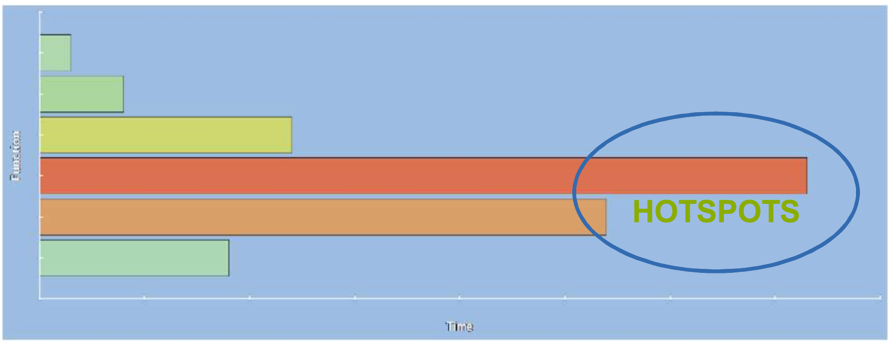
Profiler Summary

- Many profile tools are available
- NVIDIA Provided
  - NVPROF: Command Line
  - NVVP: Visual profiler
  - NSIGHT: IDE (Visual Studio and Eclipse)
- 3rd Party
  - TAU
  - VAMPIR

Optimization



Assess



– Profile the code, find the hotspot(s)
– Focus your attention where it will give the most benefit

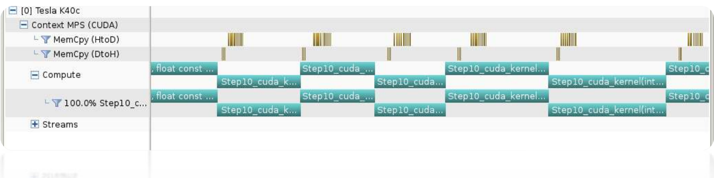
Parallelize

Applications

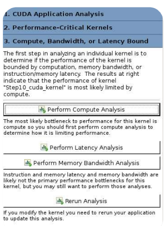
Libraries | Compiler Directives | Programming Languages

Optimize

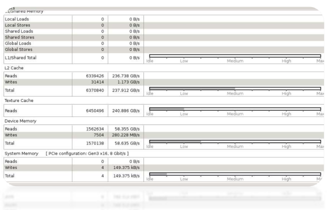
Timeline

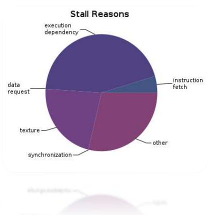


Guided System



Analysis





Question 1

- If we want to allocate an array of *v* integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the cudaMalloc() call?

  a) n
  b) v
  c) n * sizeof(int)
  d) v * sizeof(int)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 1

- If we want to allocate an array of *v* integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the cudaMalloc() call?

  a) n
  b) v
  c) n * sizeof(int)
  d) v * sizeof(int)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 2

- If we want to allocate an array of *n* floating-point elements and have a floating-point pointer variable *d_A* to point to the allocated memory, what would be an appropriate expression for the first argument of the *cudaMalloc()* call?

  a) n
  b) (void *) d_A
  c) *d_A
  d) (void **) &d_A

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 2 - Answer

- If we want to allocate an array of *n* floating-point elements and have a floating-point pointer variable *d_A* to point to the allocated memory, what would be an appropriate expression for the first argument of the *cudaMalloc()* call?

  a) n
  b) (void *) d_A
  c) *d_A
  d) (void **) &d_A

Explanation: *&d_A* is pointer to a pointer of *float*. To convert it to a generic pointer required by *cudaMalloc()* should use *(void **)* to cast it to a generic double-level pointer.

Question 3

- If we want to copy 3,000 bytes of data from host array *h_A* (*h_A* is a pointer to element 0 of the source array) to device array *d_A* (*d_A* is a pointer to element 0 of the destination array), what would be an appropriate API call for this in CUDA?

  a) cudaMemcpy(3000, h_A, d_A, cudaMemcpyHostToDevice);
  b) cudaMemcpy(h_A, d_A, 3000, cudaMemcpyDeviceTHost);
  c) cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);
  d) cudaMemcpy(3000, d_A, h_A, cudaMemcpyHostToDevice);

Question 3 - Answer

- If we want to copy 3000 bytes of data from host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array), what would be an appropriate API call for this in CUDA?

  a) cudaMemcpy(3000, h_A, d_A, cudaMemcpyHostToDevice);
  b) cudaMemcpy(h_A, d_A, 3000, cudaMemcpyDeviceTHost);
  c) cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);
  d) cudaMemcpy(3000, d_A, h_A, cudaMemcpyHostToDevice);

Question 4

- How would one declare a variable err that can appropriately receive returned value of a CUDA API call?

  a) int err;
  b) cudaError err;
  c) cudaError_t err;
  d) cudaSuccess_t err;

Question 4 - Answer

- How would one declare a variable *err* that can appropriately receive returned value of a CUDA API call?

  a) int err;
  b) cudaError err;
  c) cudaError_t err;
  d) cudaSuccess_t err;

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
antonio.pena@bsc.es