



An Introduction to Graphics Processing Unit Architecture and Programming Models

Argonne Training Program on Exascale Computing

Tim Warburton

John K. Costain Faculty Chair in the College of Science
Professor Of Mathematics and Affiliate Faculty in CMDA
Virginia Tech



```
git clone https://github.com/tcew/ATPESC18
```

“To be ready for supercomputing ...
... you are going to need to know ...
OpenCL or CUDA for GPUs.”

Tim Mattson, Intel.

... and what Tim's NVIDIA rant unintentionally reveals about coding @CPUs ...

Overview

Part 0: GPU Myths.

Part 1: NVIDIA Graphical Processing Unit

Part 2: Compute Unified Device Architecture (CUDA)

- NVIDIA's threaded offload programming model.
- Hands on: area of the Mandelbrot

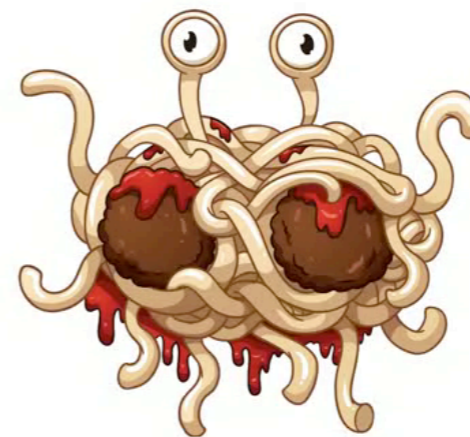
Part 3: Interlude on CUDA optimization.

Part 4: Portable programming models:

- Open Computing Language (OpenCL)
- Open Concurrent Computing Abstraction (OCCA)

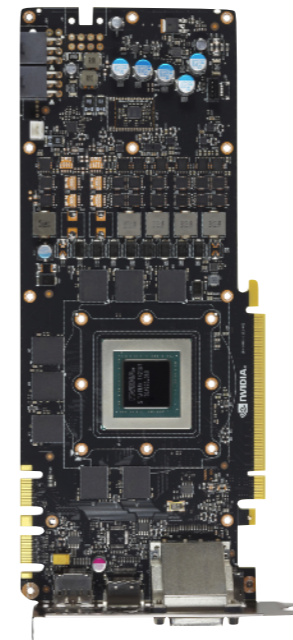
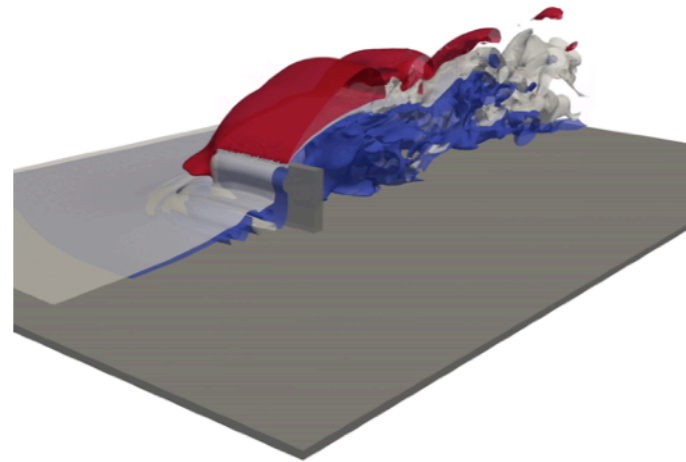
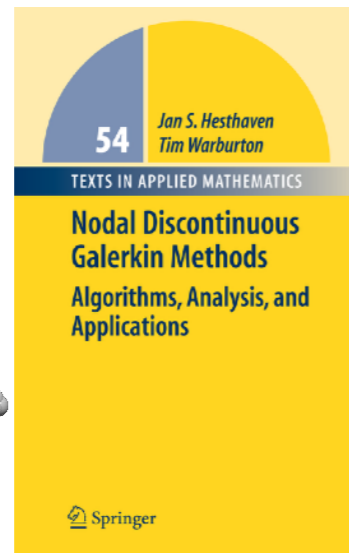
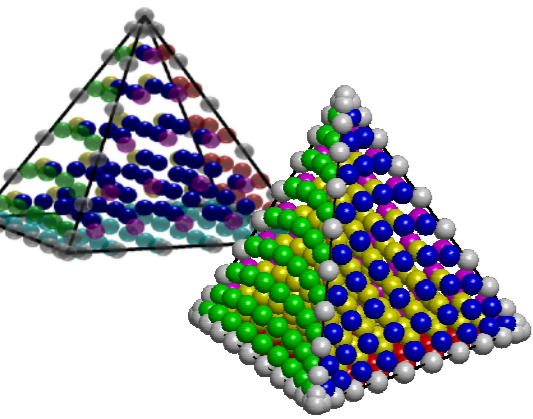
Part 5: Hands on flow simulation:

- Prep: find a png image with white background.
- Run GPU flow simulation using your image.
- Visualize your results as a movie.
- Enter competition.



My Research...

Goal: fast, scalable, flexible & accurate numerical PDE solvers adapted for modern many-core architectures.



Approximation Theory

Numerical Analysis

Numerical methods & Physical PDE Modeling

Accelerated Computing

High Performance Scalability

Basic science

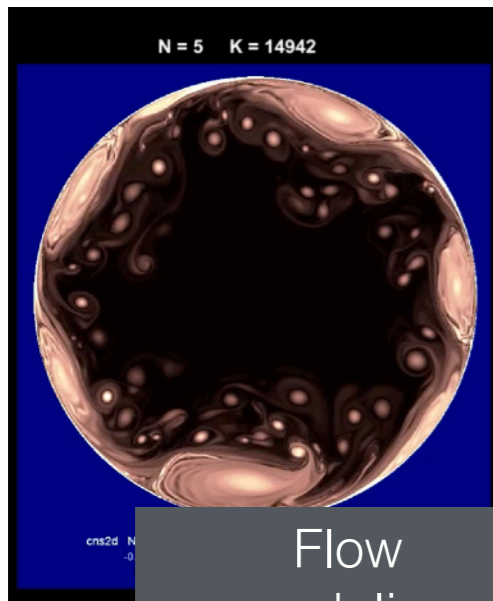
Application

Industrial Scale

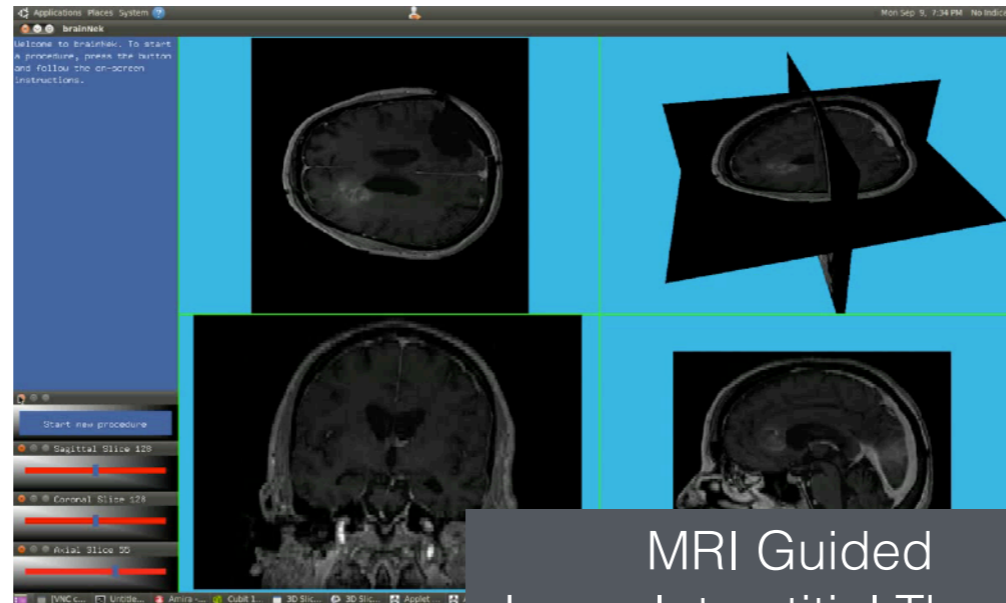
High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.

Some GPU Accelerated Apps...

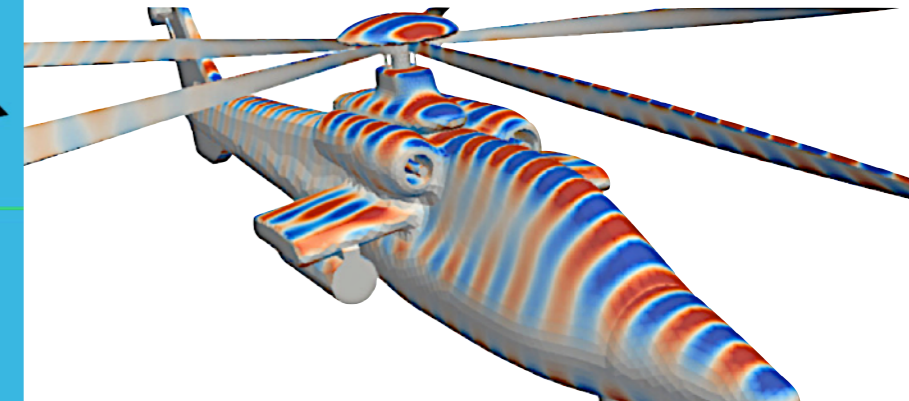
We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy...



Flow modeling



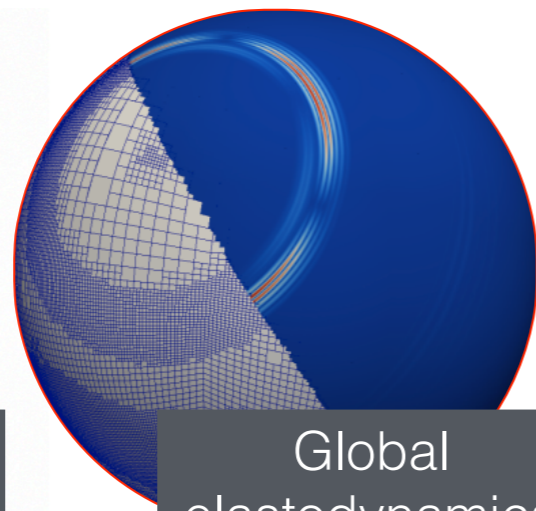
MRI Guided Laser Interstitial Therapy



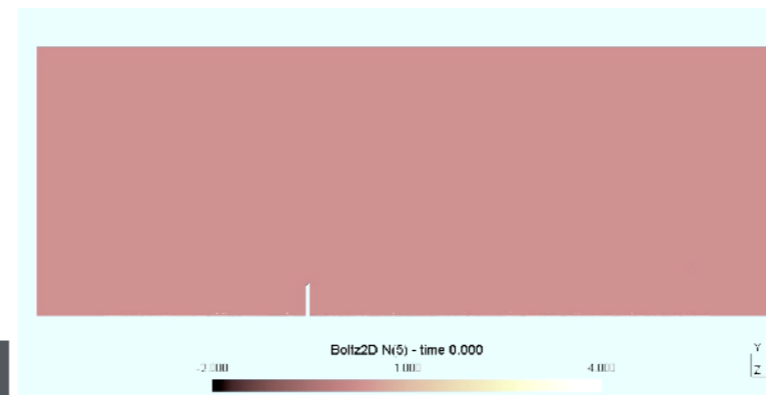
Electromagnetic scattering



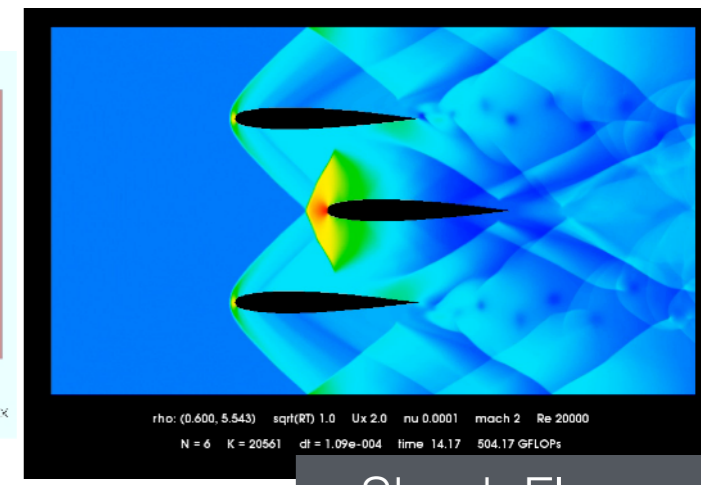
Tsunami propagation



Global elastodynamics



Kinetic Models



Shock Flows

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.



Exascale Co-Design

The **Center for Efficient Exascale Discretizations (CEED)** is a co-design center within the U.S. Department of Energy (DOE) [Exascale Computing Project \(ECP\)](#) with the following goals:

- Help applications leverage future architectures by providing them with [state-of-the-art discretization algorithms](#) that better exploit the hardware and deliver a significant performance gain over conventional low-order methods.
- Collaborate with hardware vendors and software technologies projects to utilize and impact the upcoming exascale hardware and its software stack through CEED-developed [proxies and miniapps](#).
- Provide an efficient and user-friendly unstructured PDE discretization component for the upcoming [exascale software ecosystem](#).

CEED is a research partnership involving [30+ computational scientists](#) from two DOE labs and five universities, including members of the [Nek5000](#), [MFEM](#), [MAGMA](#), [OCCA](#) and [PETSc](#) projects. You can reach us by emailing ceed-users@llnl.gov or by leaving a comment in the [CEED user forum](#).

The center's co-design efforts are organized in [four interconnected R&D thrusts](#), focused on the following computational motifs and their performance on [exascale hardware](#). See also our [publications](#).

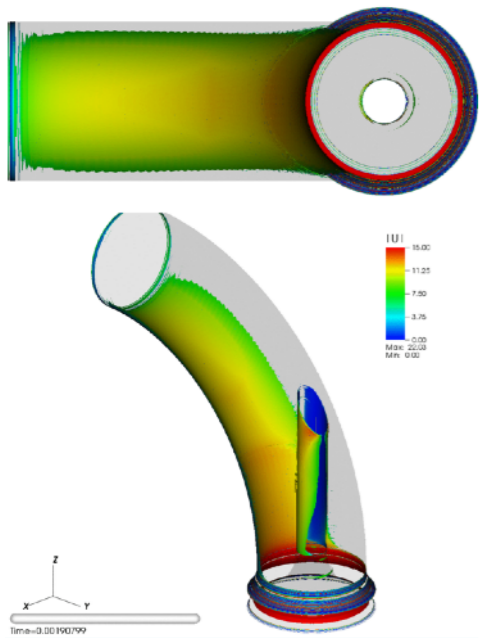
<http://ceed.exascaleproject.org>

You will hear more about this from Tzanio Kolev (LLNL) & Mark Shepard (RPI) on 08/06

libParanumal: GPU enabled solvers

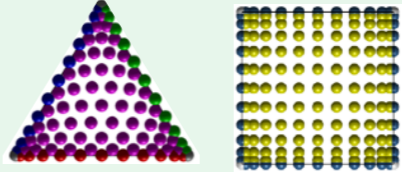
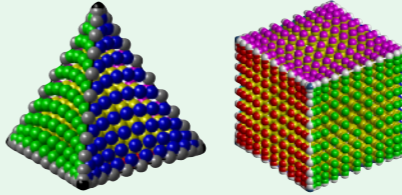
DOE ECP Flow App

Nek5K



<https://ceed.exascaleproject.org/nek/>

libParanumal

Flow Solvers	Discretization	Core
Incompressible Navier-Stokes	Discontinuous Galerkin methods	Heterogeneous hybrid multigrid
Linear Elliptic	Galerkin finite element methods	
Compressible Navier-Stokes	Implicit time steppers	 High-order Elements
Galerkin-Boltzmann	Implicit-explicit time steppers	
Benchmarks	Explicit time steppers	Parallel mesh wrangling

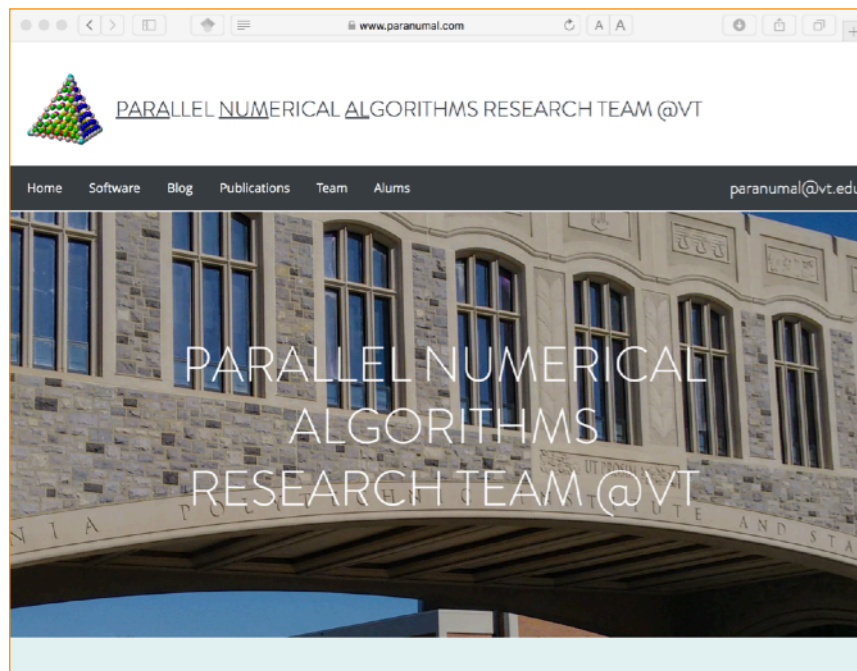
OCCA

Backends
NVIDIA CUDA
AMD HIP
OpenMP
OpenCL

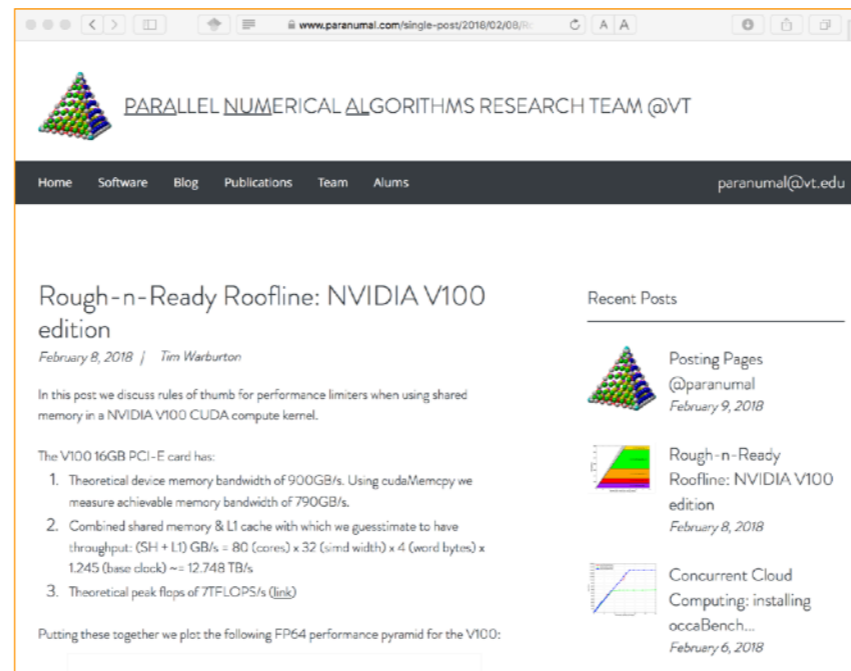
<https://github.com/paranumal/libparanumal>

Goal: Drop in replacement for core elliptic & flow solver functionality of the Nek5000 simulation code.
Animation from Giannakopoulos et al and the SEAL lab <http://fischerp.cs.illinois.edu/seal/>

Resources: libParanumal release on 8.1.18



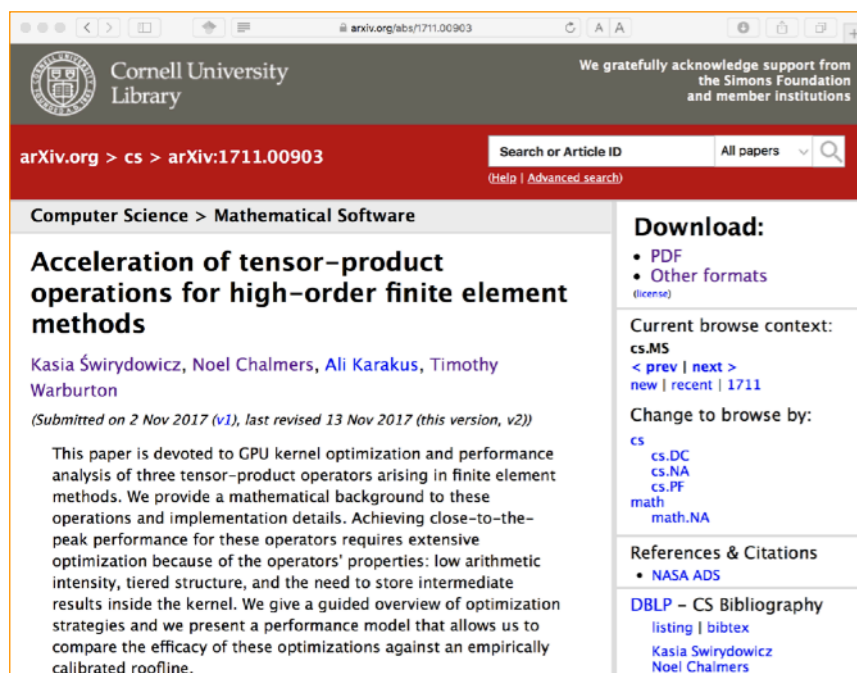
Web page: paranumal.com



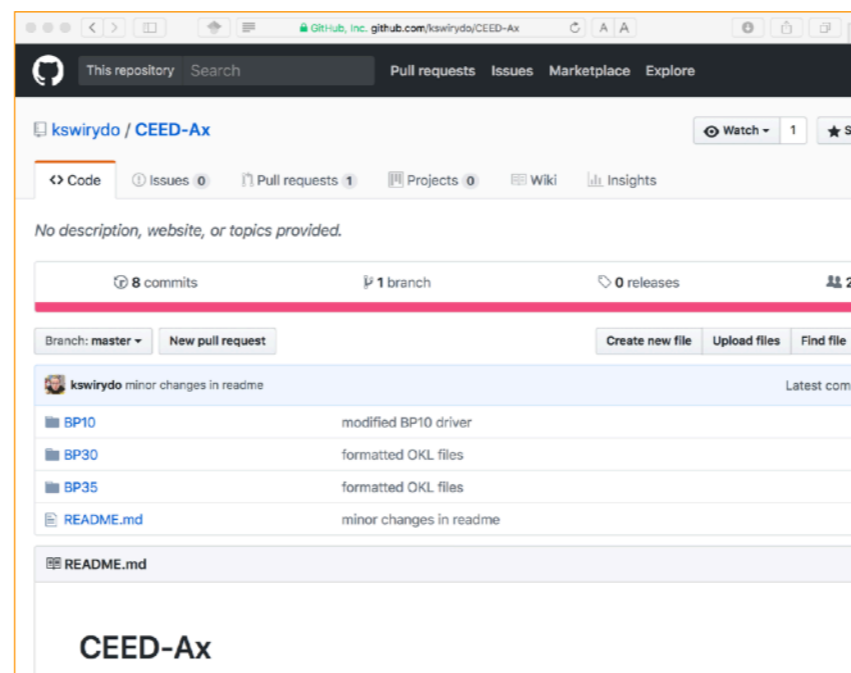
Blog: paranumal.com/blog



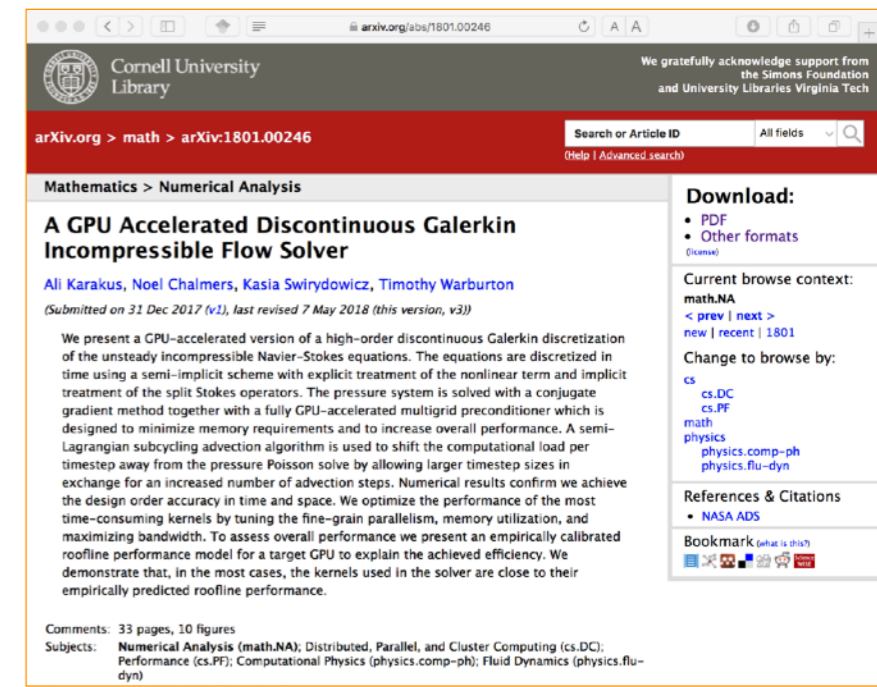
Twitter: twitter.com/paranumal



VT CEED BK paper: P100



BKs: github.com/kswirydo/CEED-Ax



VT INS2D+OCCA+Sub-cycling+AMG

You can also send queries to: paranumal@vt.edu
libParanumal source code: <https://github.com/paranumal/libparanumal>

Today: slides & repos

Slides:

www.math.vt.edu/people/tcew/ATPESC18

Examples:

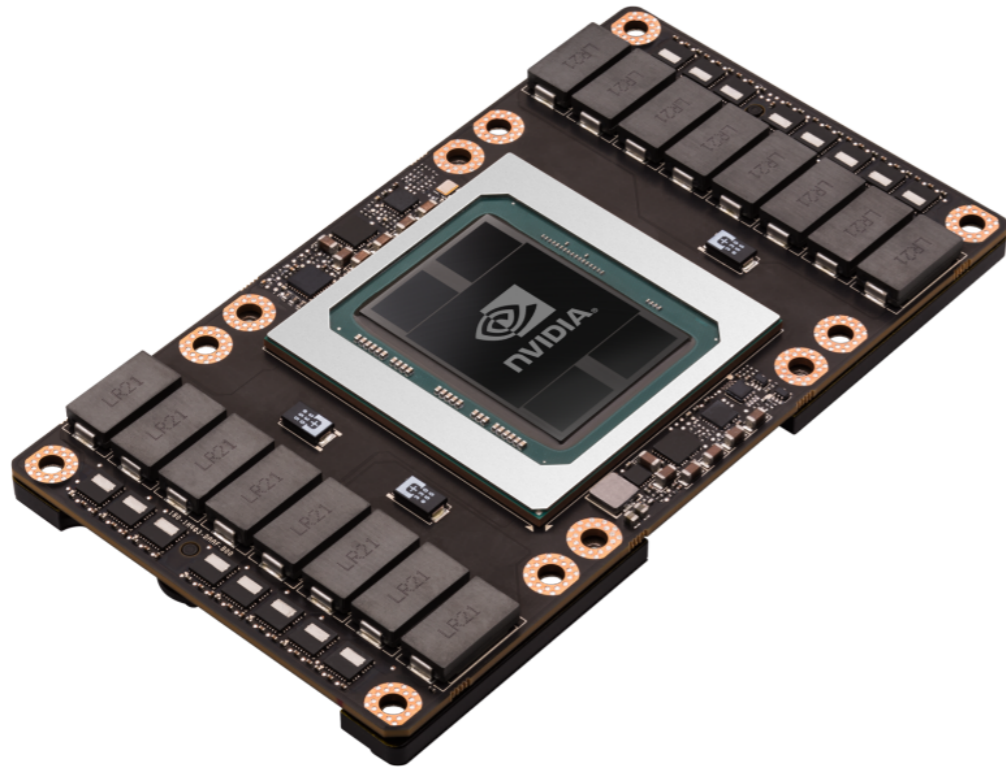
git clone <https://github.com/tcew/ATPESC18>

OCCA repo (0.2 branch version for this tutorial):

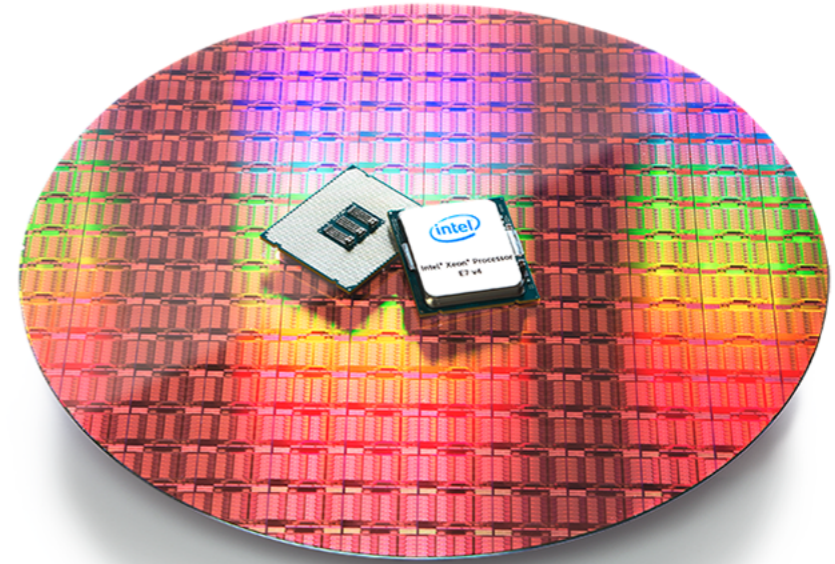
git clone <https://github.com/libocca/occa> -b 0.2

Part 0: GPU Reality Check

Myth #1: GPU 100x faster than CPU



NVIDIA P100: High Bandwidth Memory
up to 732 GB/s



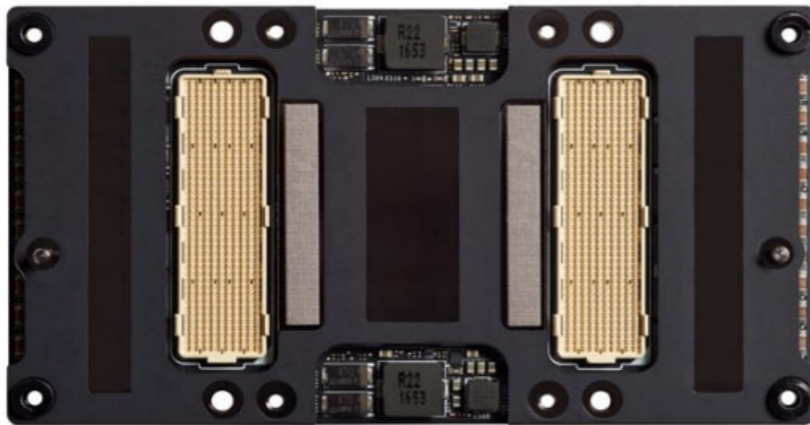
Intel E7-8894 v4: 4 memory channels
up to 85 GB/s per socket

<http://www.nvidia.com/object/tesla-p100.html>

https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz

Majority of HPC codes are memory and network bound.

Myth #1: GPU 100x faster than CPU



NVIDIA V100 GPU:
up to 900 GB/s (HBM2)



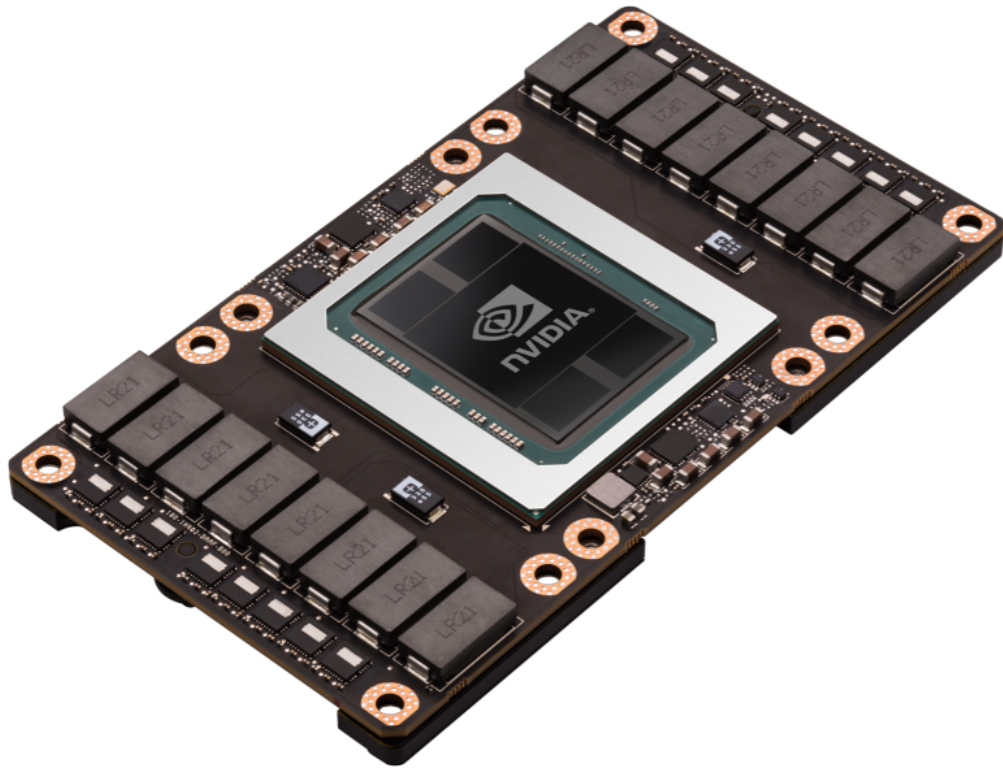
Intel Intel® Xeon® Platinum 8180 CPU:
up to 119 GB/s bandwidth per socket

For well optimized bandwidth limited codes with more data than cache
=> one GPU is about 3.5x faster than a dual socket CPU.

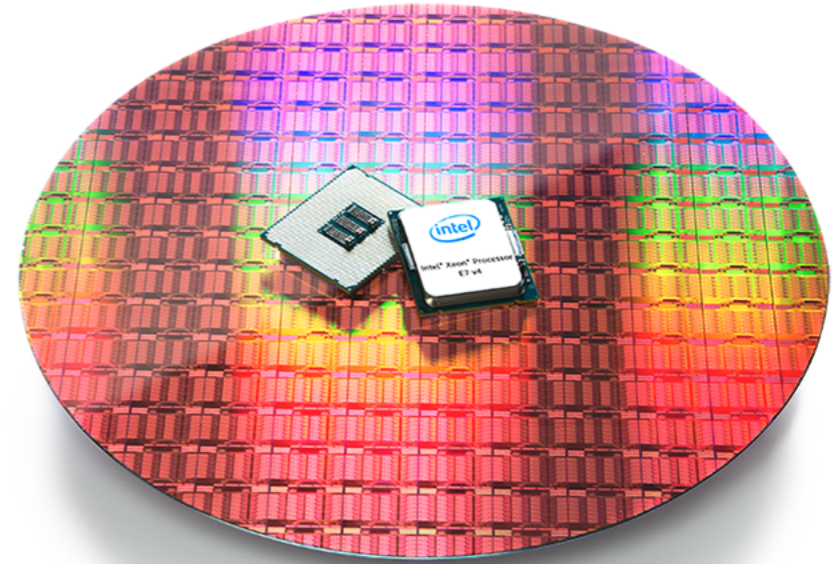
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz
<https://www.gamepc.com/shop/products?sku=900-2G503-0000-000-16GB>

Majority of HPC codes are memory and network bound.

Myth #2: GPUs are expensive



NVIDIA P100: ~\$5K

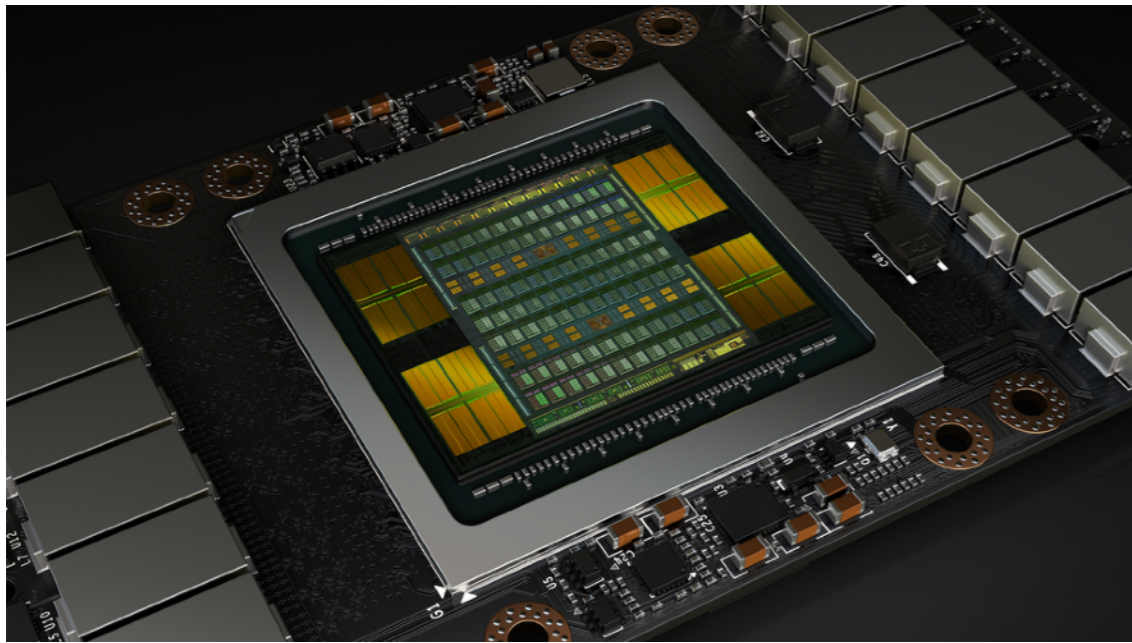


Intel E7-8894 v4: ~\$9K

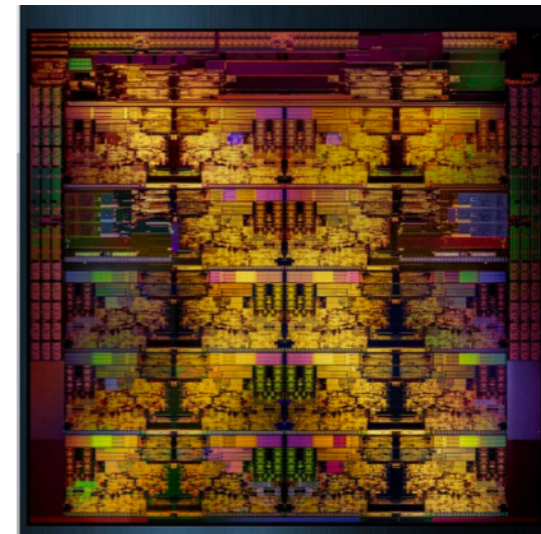
Price estimates from <http://www.thinkmate.com/>
<http://www.anandtech.com/show/11121/intel-xeon-e7-8894-v4-cpu-24c-48t-9000-usd>

*Prices for non-consumer GPUs are carefully calibrated to be similar to CPU.
[these are both premium high-end processors]*

Myth #2: GPUs are expensive



NVIDIA V100 GPU:
\$7-12K each



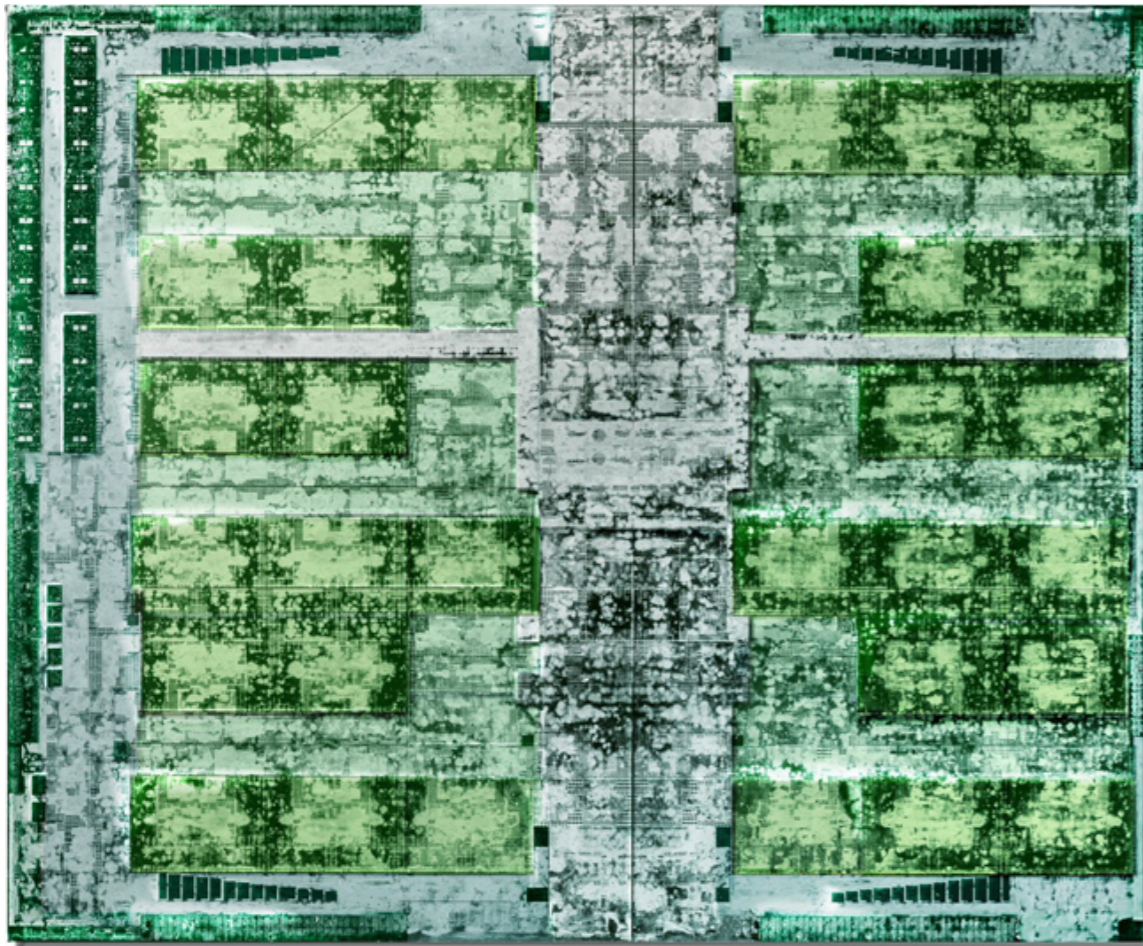
Intel Intel® Xeon® Platinum 8180 CPU:
~\$10K per socket

↑
Super ridiculous top end CPU

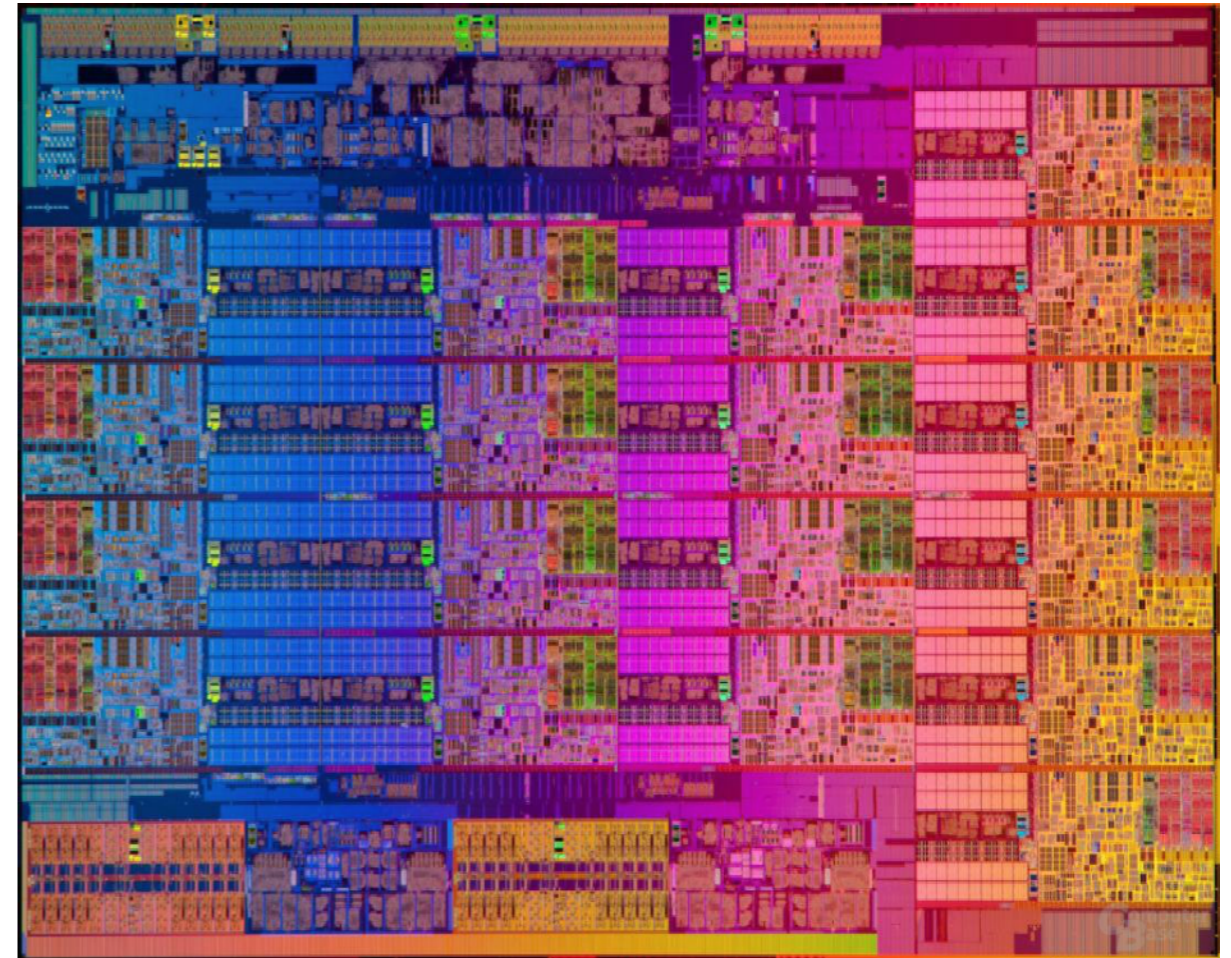
Price estimates from <http://www.thinkmate.com/>
<http://www.anandtech.com/show/11121/intel-xeon-e7-8894-v4-cpu-24c-48t-9000-usd>

*Prices for non-consumer GPUs are carefully calibrated to be similar to CPU.
[these are both premium high-end processors]*

Myth 3: GPU & CPU are very different



NVIDIA P100: 56 “cores” with
4 32-way SIMT units



Intel E7-8894 v4: 24 hyper-
threading cores with 256 bit AVX2 instructions

<http://www.nvidia.com/object/tesla-p100.html>

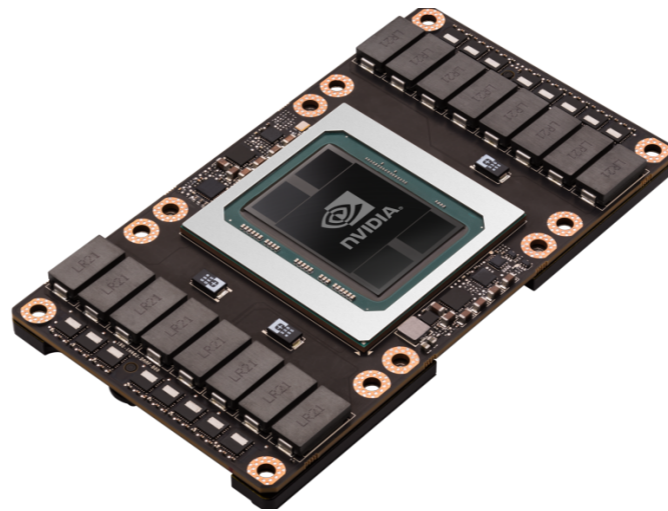
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz

GPUs and CPUs both consist of multiple cores each equipped with SIMD vector units.

Myth #4.0: OpenACC is magic

OpenACC
Directives for Accelerators

+



≠



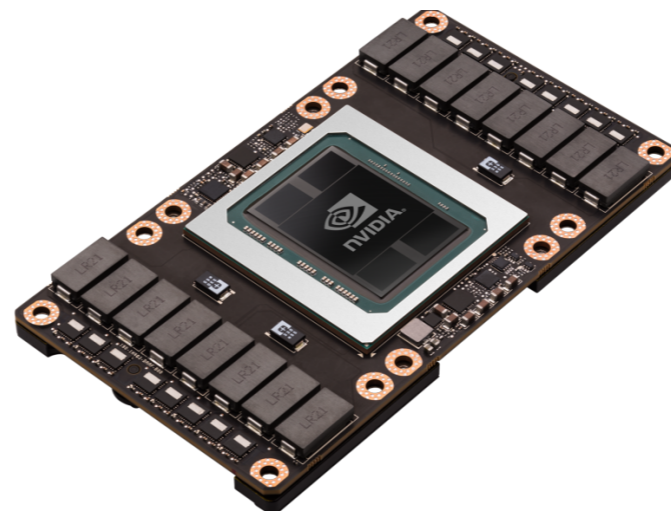
<http://icons.iconarchive.com/icons/hadezign/hobbies/256/Magic-icon.png>

“good” OpenACC codes are quite often derived from good CUDA implementations.

Myth #4.1: CUDA is magic



+



≠



<http://icons.iconarchive.com/icons/hadezign/hobbies/256/Magic-icon.png>

“good” CUDA codes quite often emerge from a prolonged gestation.

Reality Check

It takes more than 3 hours to master GPUs...

... but we can discuss some of the basics ...

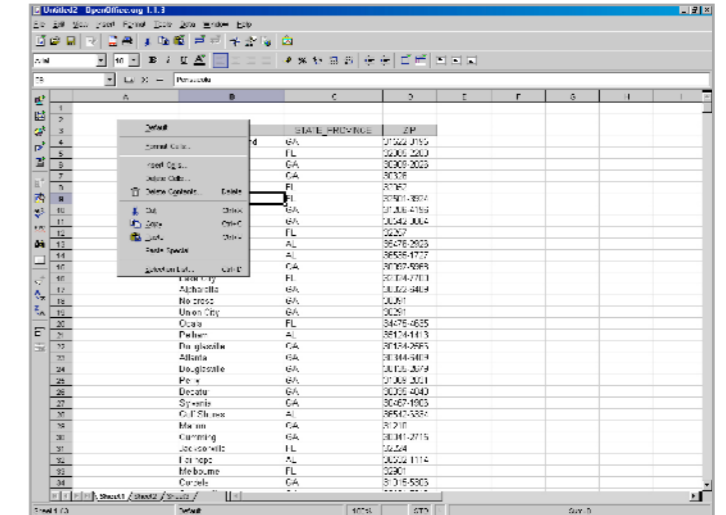
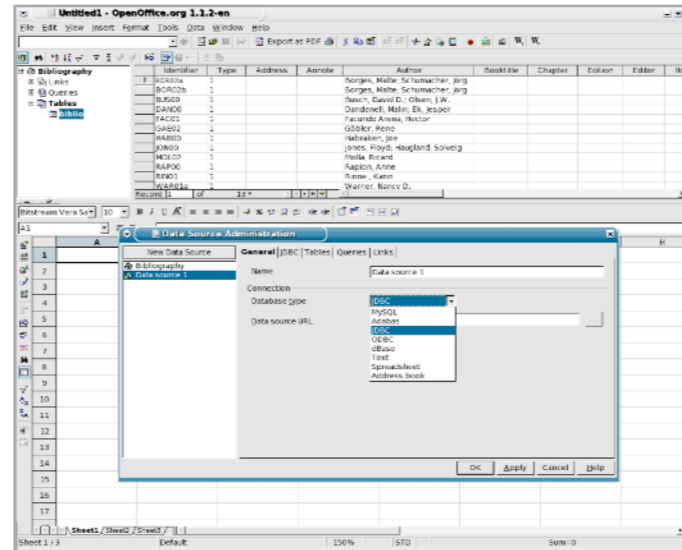
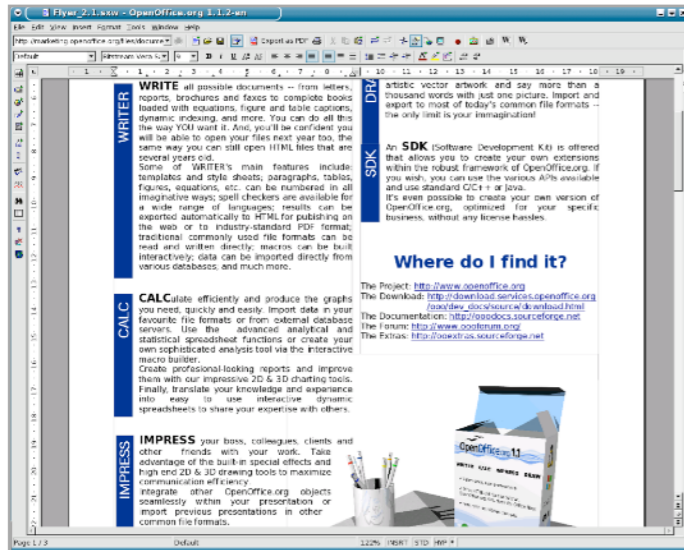
... background to the Kokkos & Raja talks on **easier** GPU computing ...

... there are many web resources ...

... and nothing beats hands on.

Part 1: From CPU to GPU

CPU: architecture follows purpose

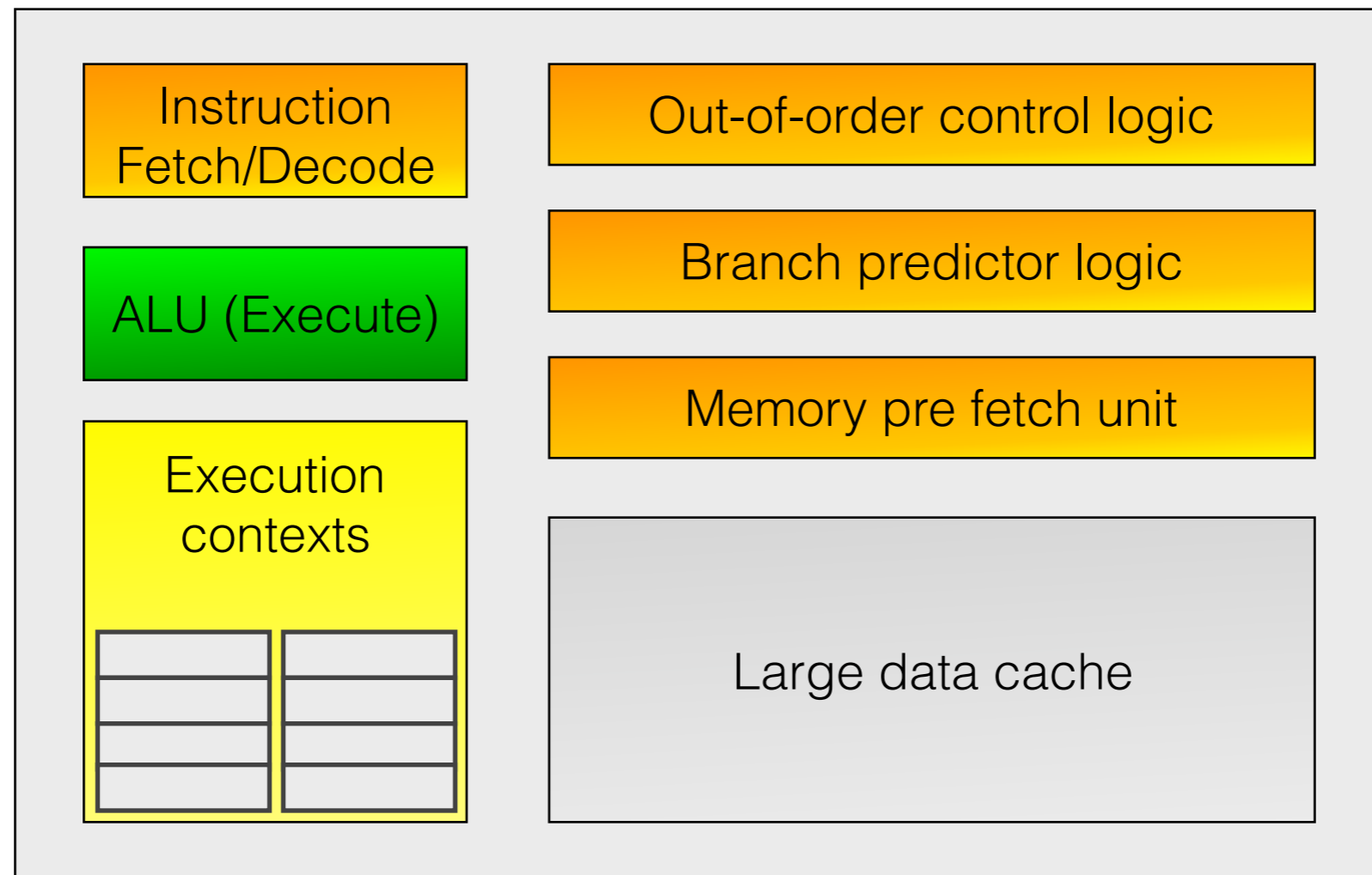


Original design goals for CPUs:

- Make single threads very fast.
- Reduce latency through large caches.
- Predict, speculate.

CPU: abstract modern architecture

Modern “CPU-Style” core design emphasizes individual thread performance.



Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

Execution context: memory and hardware associated to a specific stream of instructions, e.g. registers.

GPU: massively parallel processing



Fallout 4 Screenshot

<http://www.gamespot.com/articles/check-out-fallout-4-1080p-screenshots-from-the-deb/1100-6427822/>

<http://developer.nvidia.com/object/gpu-gems-3.html>

GPU: massively parallel compute



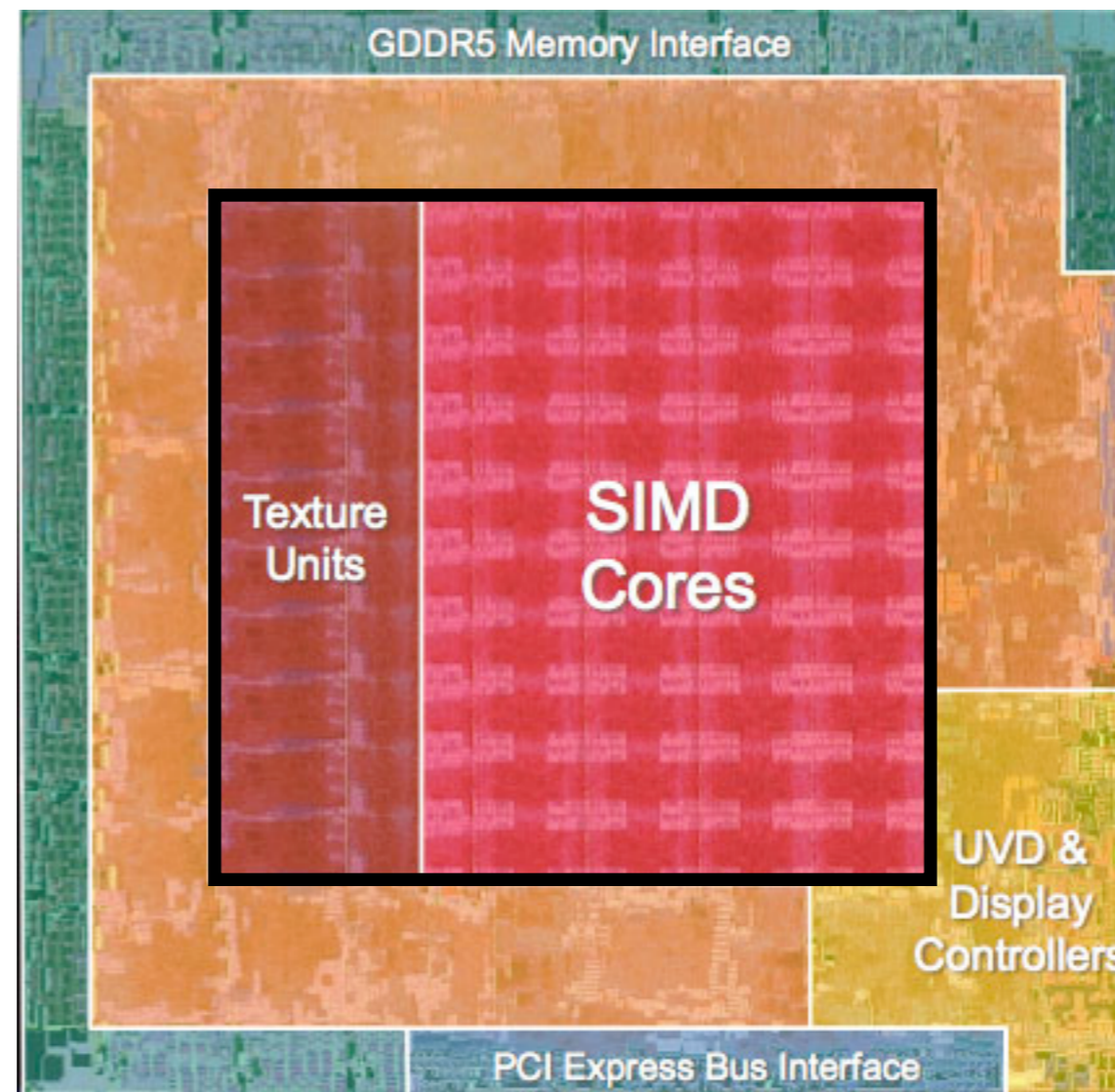
Design goals for GPUs:

- Throughput matters and single threads do not.
- Hide memory latency through parallelism.
- Let programmer deal with “raw” storage hierarchy.
- Avoid high frequency clock speed:
 - Desirable for portable devices, consoles, laptops...

<http://developer.nvidia.com/object/gpu-gems-3.html>

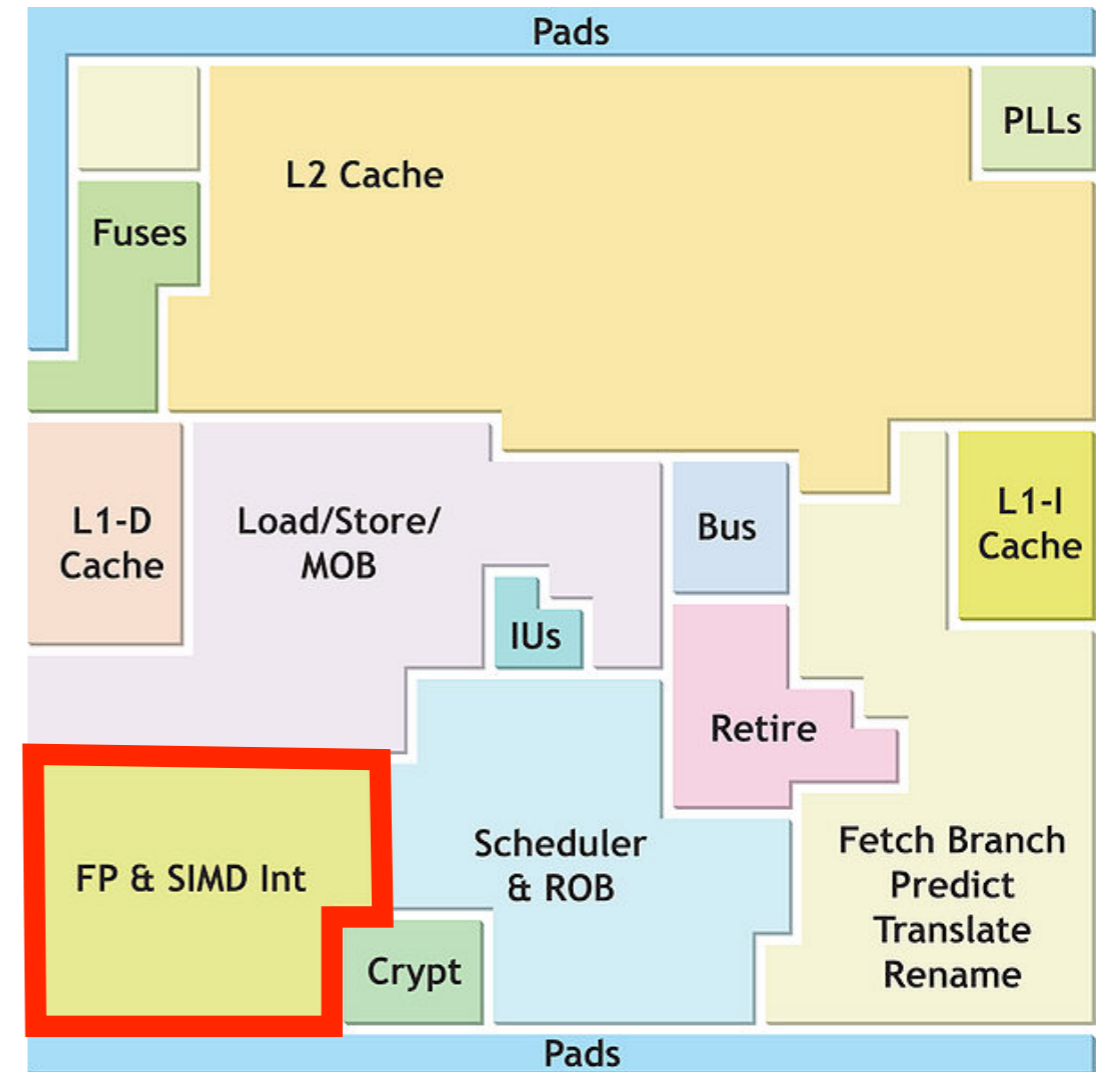
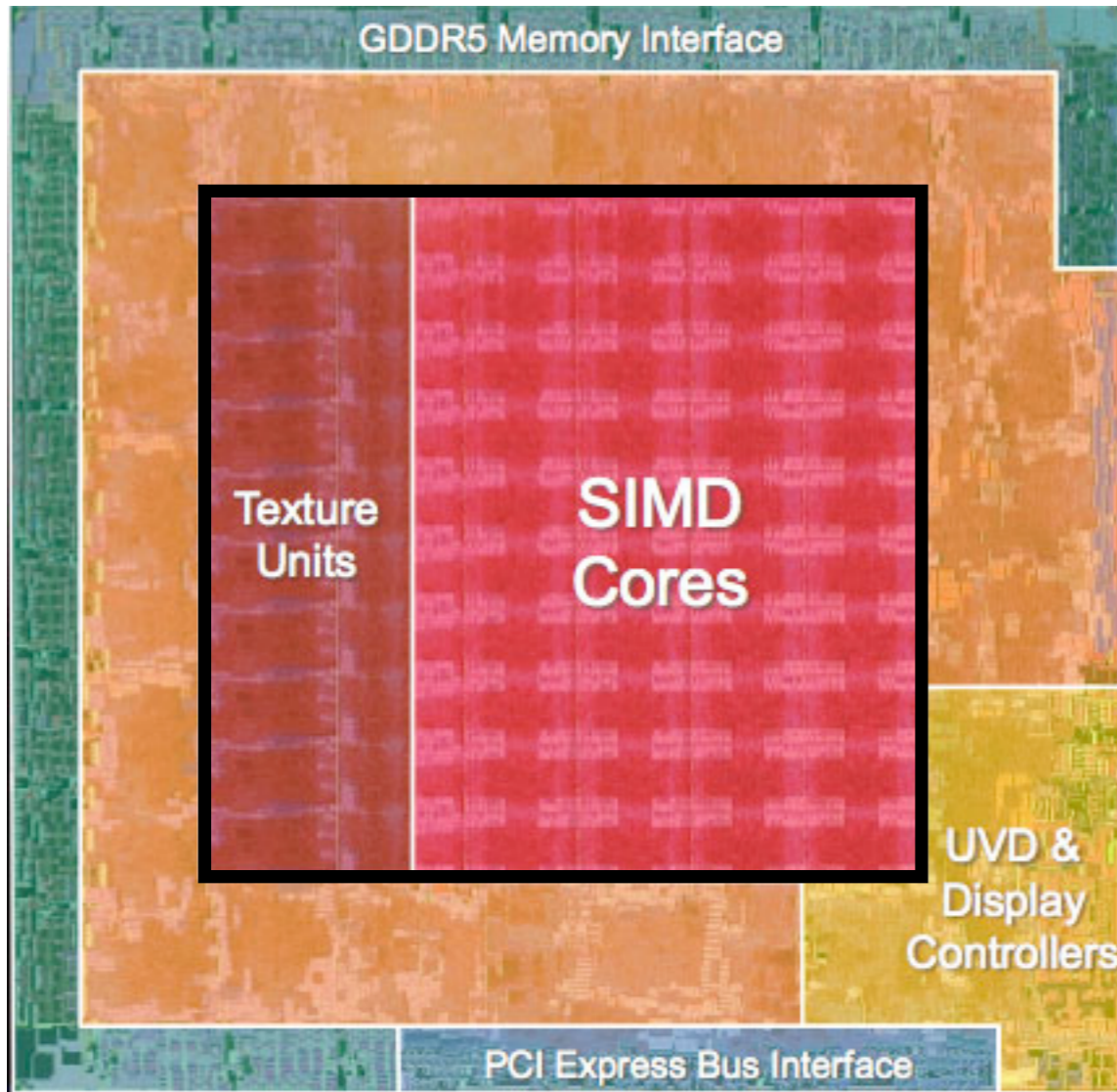
GPU: early example

Die floorplan: AMD RV770 (2008) 55 nm, 800 SP simultaneous ops
The majority of the silicon is devoted to computation



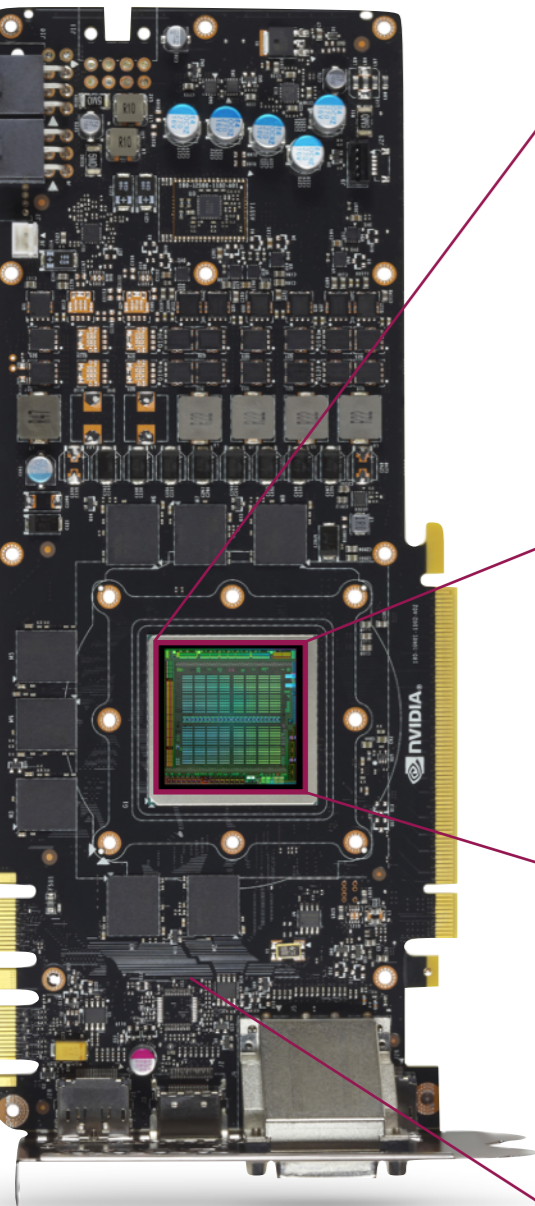
GPU: early example

Comparison of block diagram of vintage GPU and CPU



GPU: Maxwell architecture

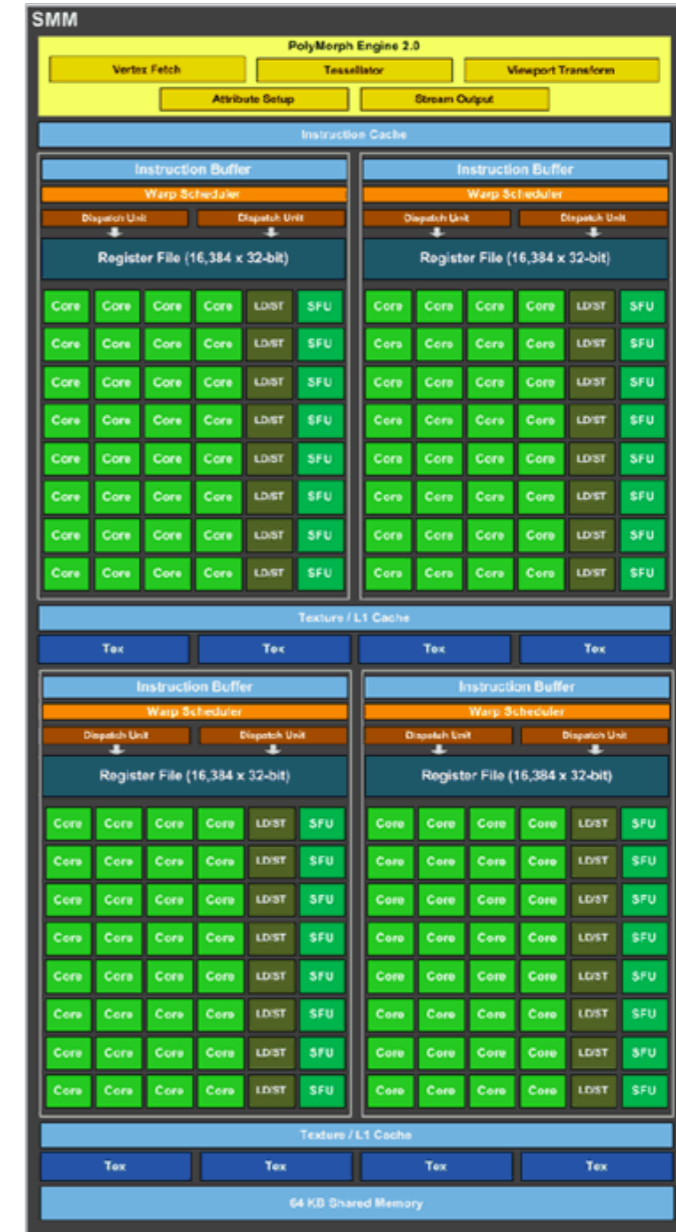
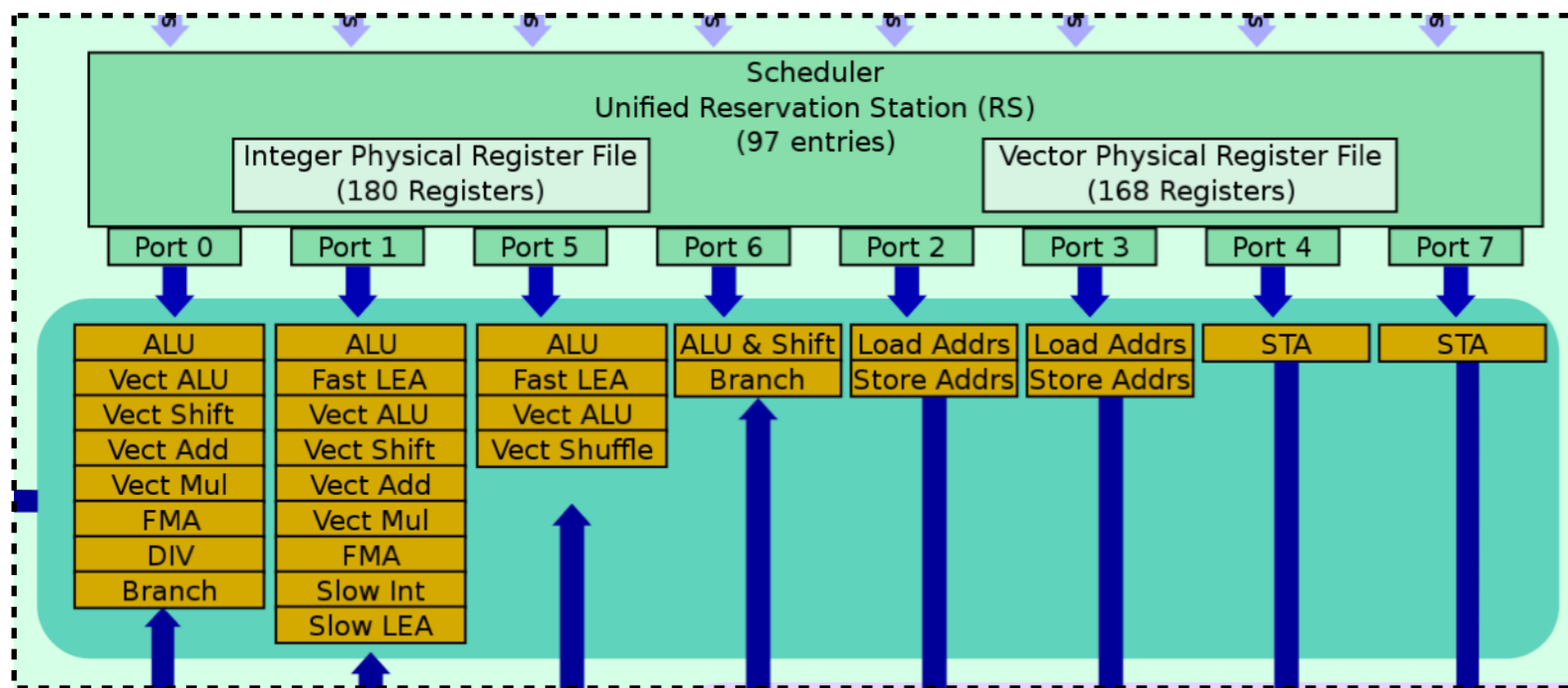
NVIDIA Maxwell GM204 GPU



16 Maxwell cores each have four SIMD clusters with 32 ALUs.
Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)

CPU v GPU: fundamental difference #1

Each CPU core executes scalar or vector operations.
Each GPU core only executes vector instructions.



CPU: Single Instruction Multiple Data (SIMD) parallelism through ILP & vector execution units.

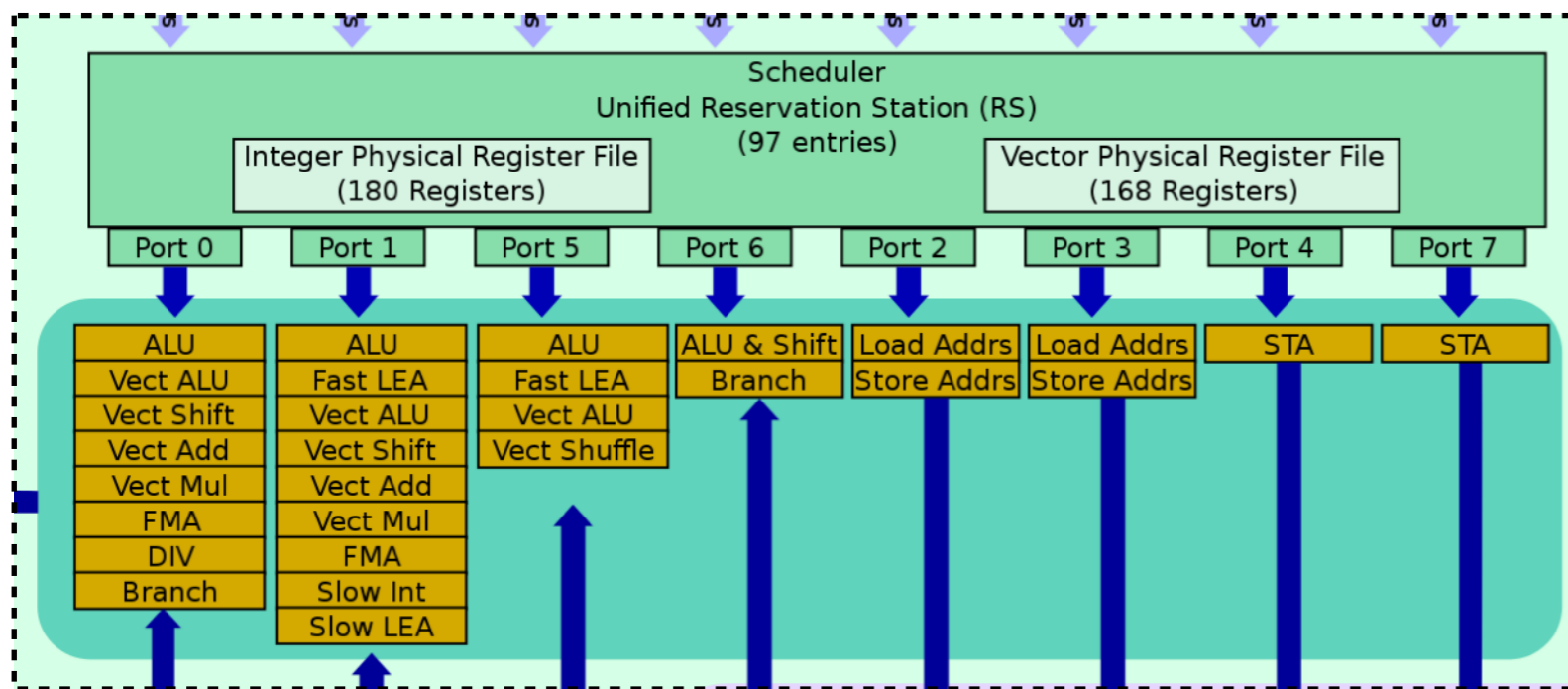
GPU: SIMD parallel execution of **all** operations

<http://en.wikichip.org/wiki/intel/microarchitectures/skylake>

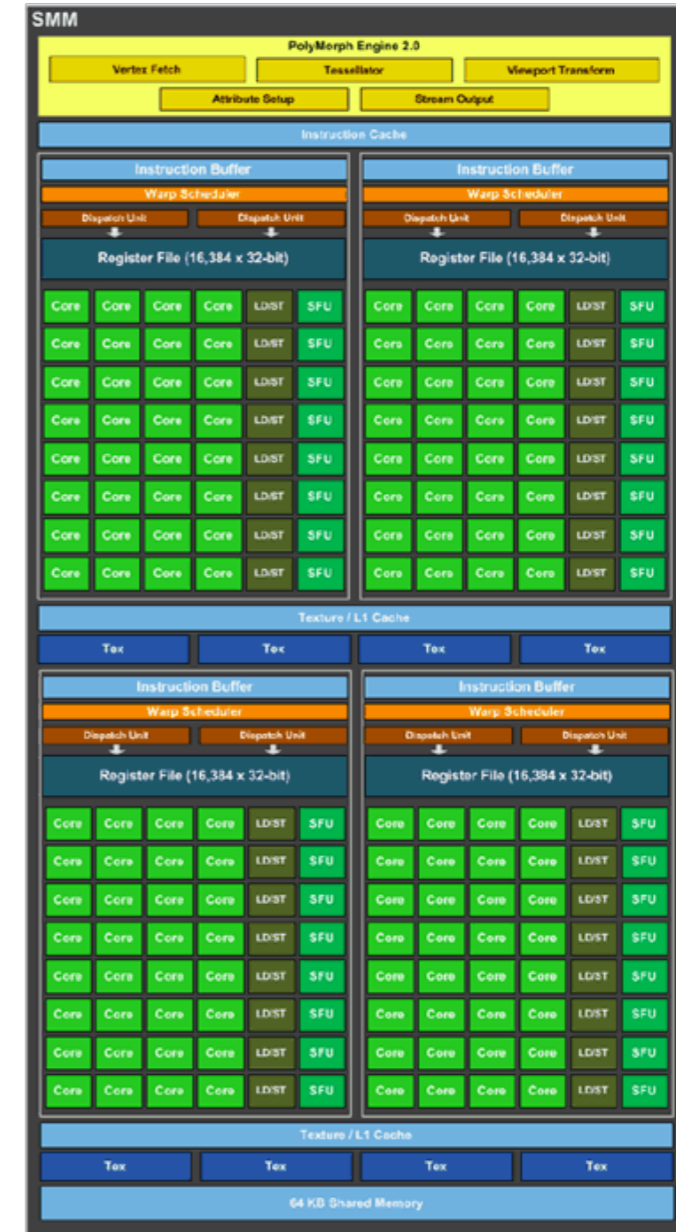
*Compilers may need to be coaxed into generating vector instructions for CPU.
Recall: "Performance, SIMD, Vectorization and Performance Tuning" talk by James Reindeer.*

CPU v GPU: fundamental difference #2

GPU cores are engineered to switch quickly between threads to recover stalls



Skylake core: 180 Integer registers and 168 floating point registers



Maxwell core: 16K registers

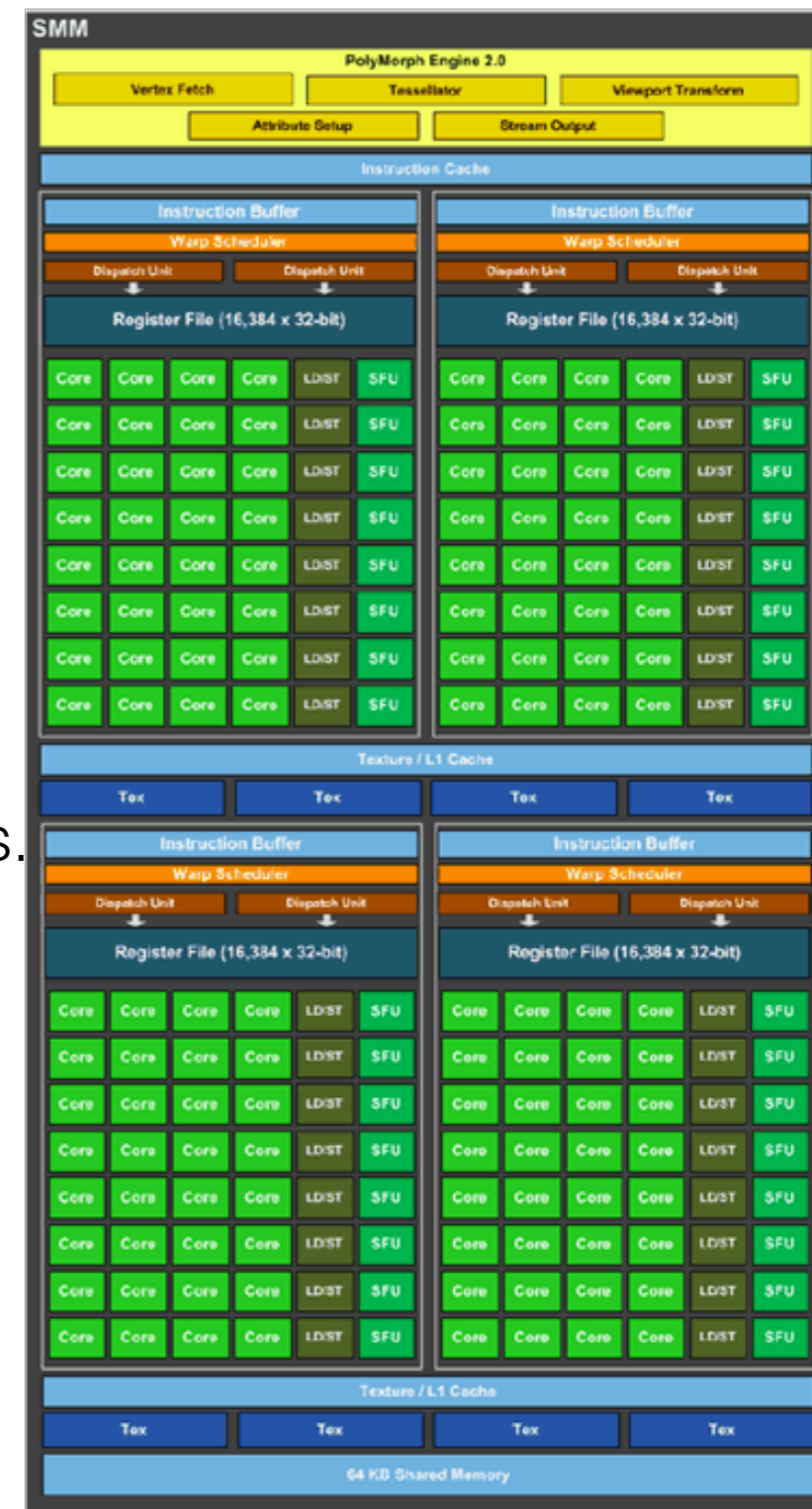
<http://en.wikichip.org/wiki/intel/microarchitectures/skylake>

Compilers may need to be coaxed into generating vector instructions for CPU.

GPU: summary of architecture

Summary of multi-level GPU parallel architecture

- A GPU has multiple cores and each core:
 - Has one (or more) wide SIMD vector units.
 - Wide SIMD vector units execute one instruction stream.
 - Has a pool of shared memory.
 - Shares a register file shared privately among all the ALUs.
 - Fast switches thread blocks to hide memory latency.
- Branching code (“ifs”) involves partial serialization.
- Nice summary:
<http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html>



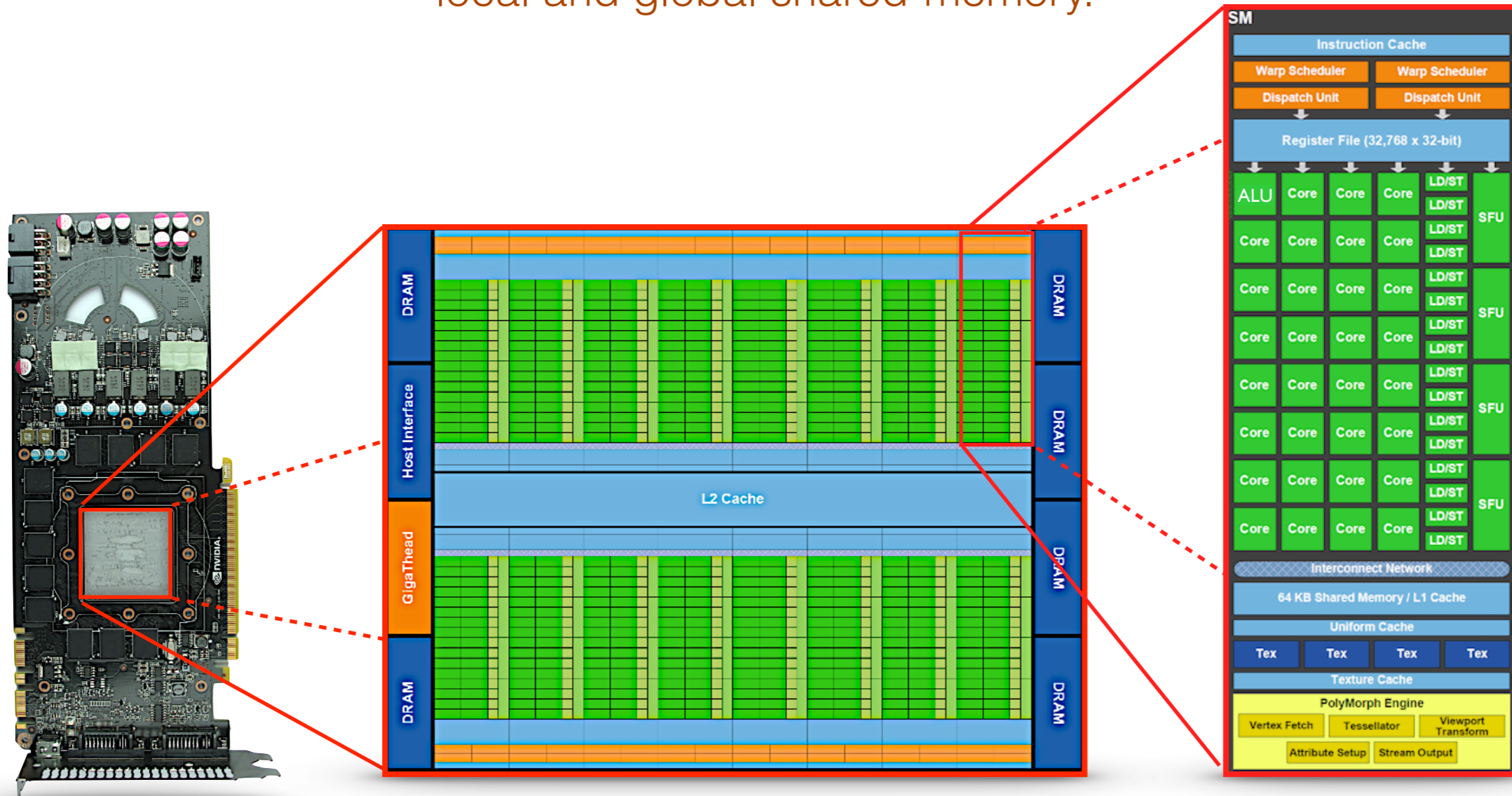
* SIMD width here is the number of ALUs in one of the core's vector unit.
The actual specifics vary but this is a good abstract viewpoint.

Part 2: NVIDIA GPUs Core Evolution

NVIDIA's Compute Unified Device Architecture
GPU programming model

GPU: excess ALUs

Modern GPUs combine: multiple wide vector processing cores with local and global shared-memory.



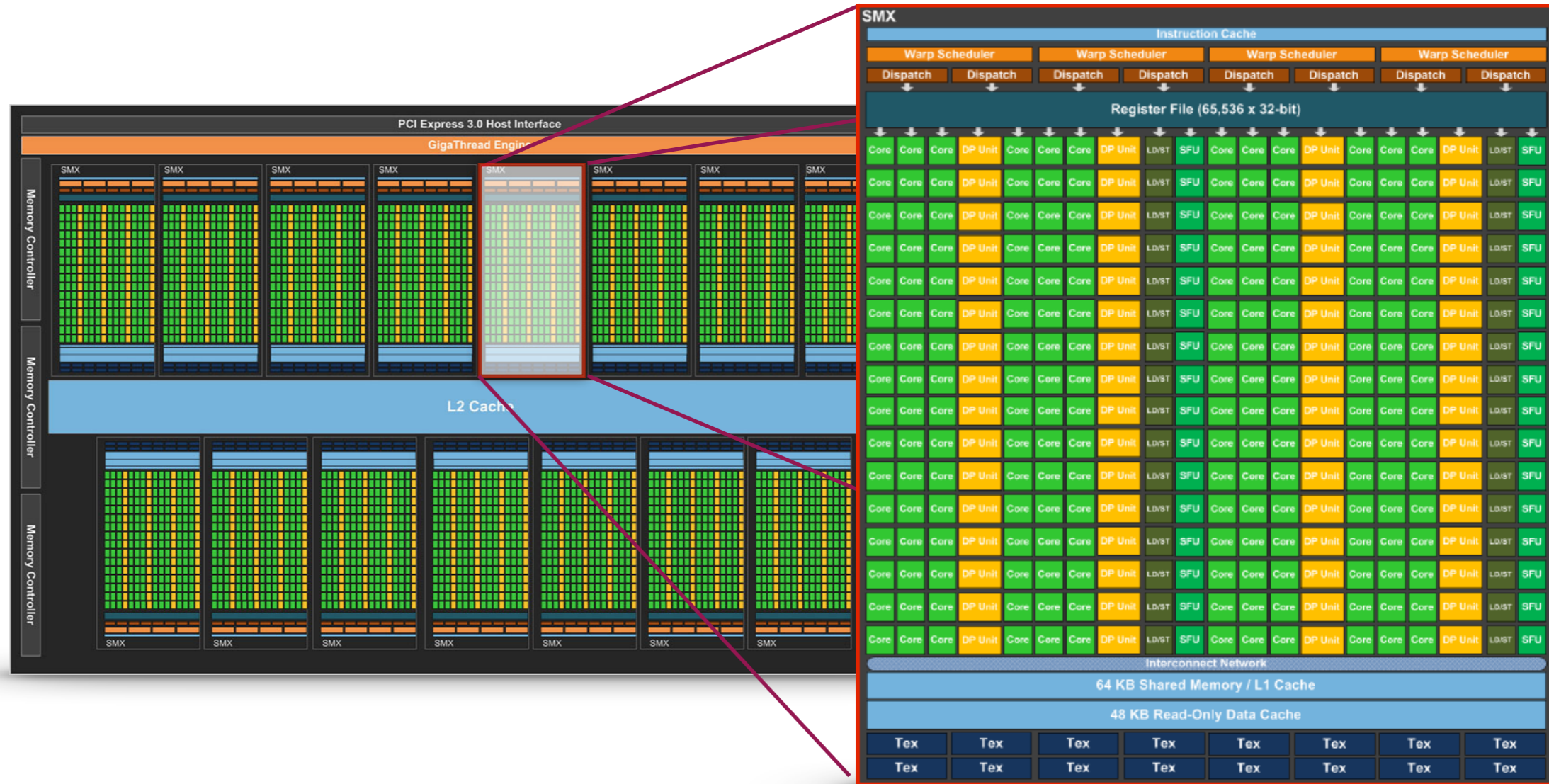
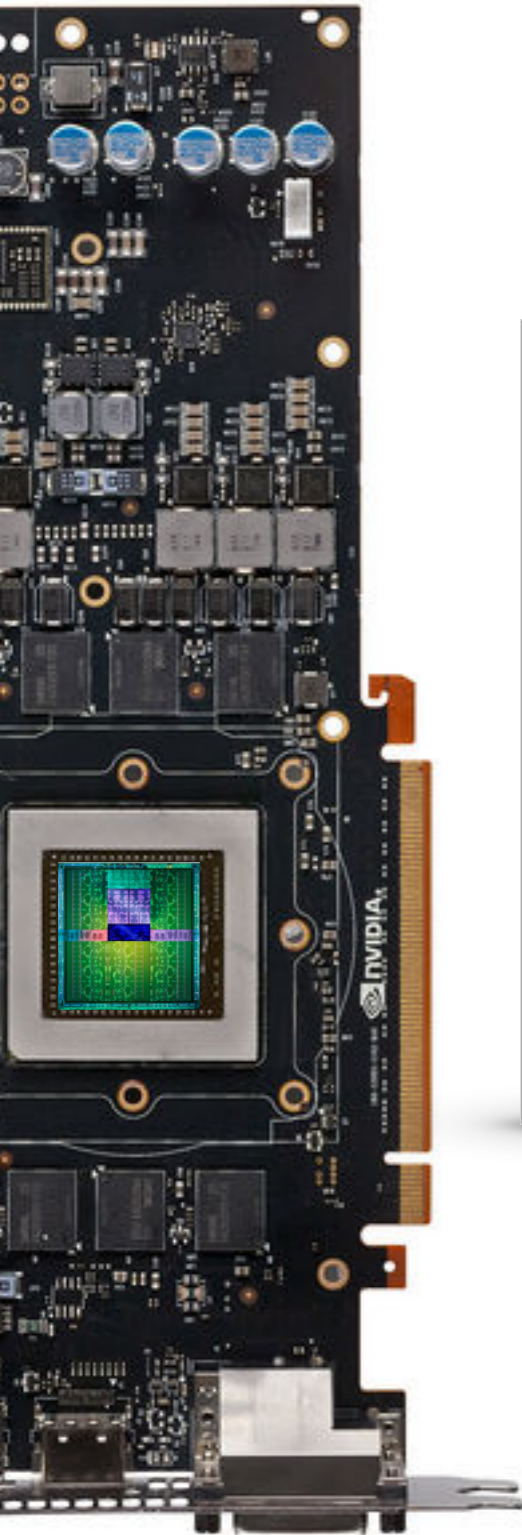
Each Fermi core (SM) has a SIMD clusters of 32 FPUs
Data streams at ~50 GFLOAT/s and computes up to 1.4 TFLOP/s (SP)

Theoretical peak performance requires ~28 FLOP per float moved between device & memory !!!

Note: for the Fermi generation cards they put the L1 and L2 caches back 😊

GPU: Kepler GPU

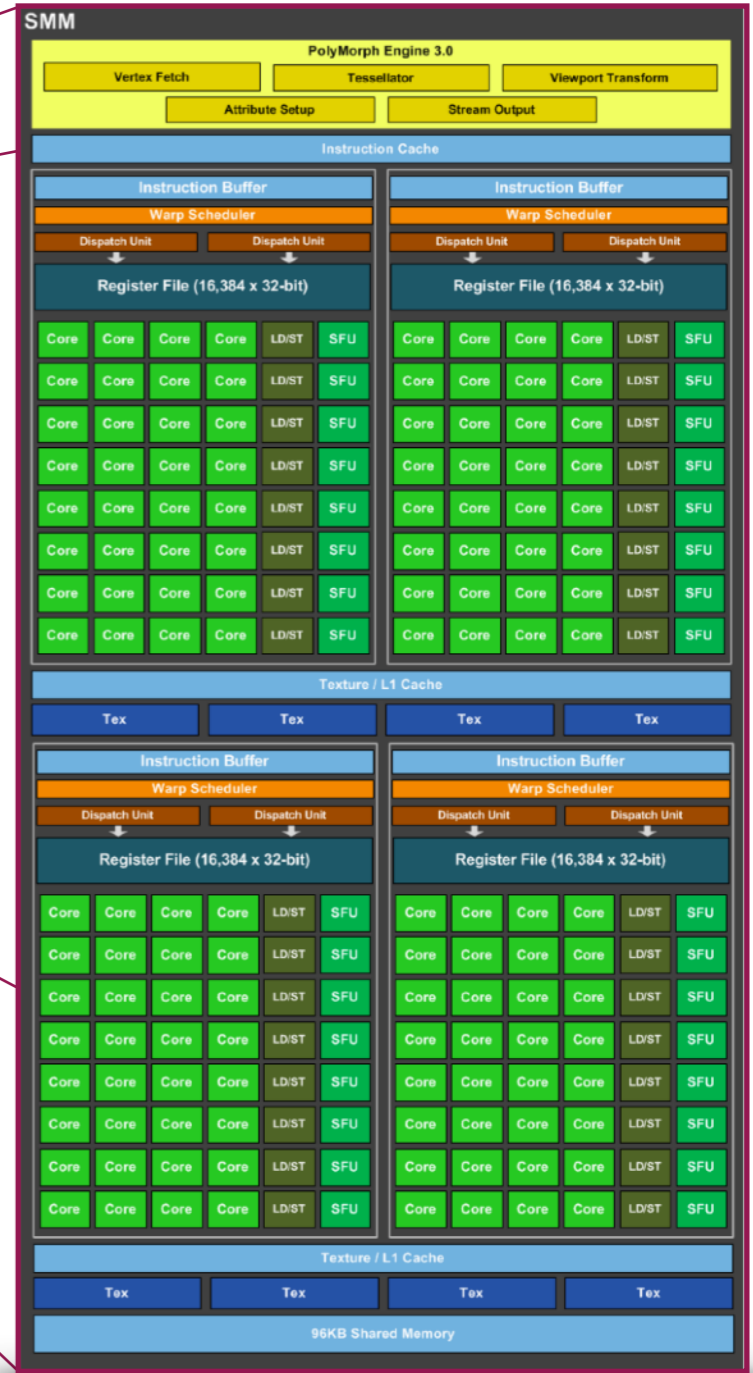
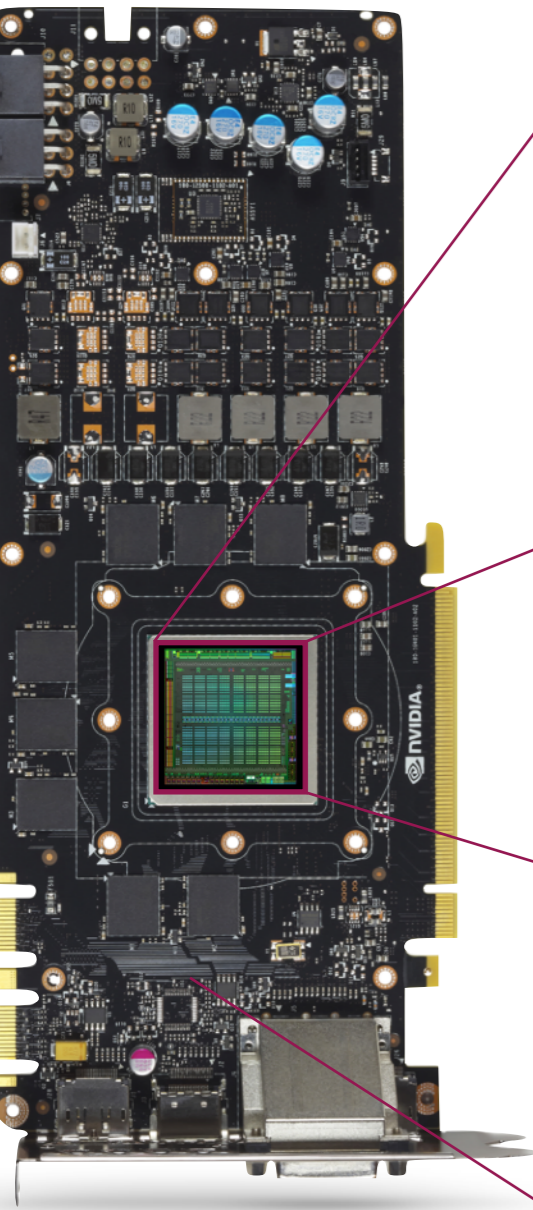
GK110: 15 cores that cluster 192 FPU each.



Each Kepler core (SMX) has six SIMD clusters of 32 ALUs
Data streams at ~70 GFLOAT/s and peak 4+ TFLOP/s (SP)

GPU: Maxwell GPU

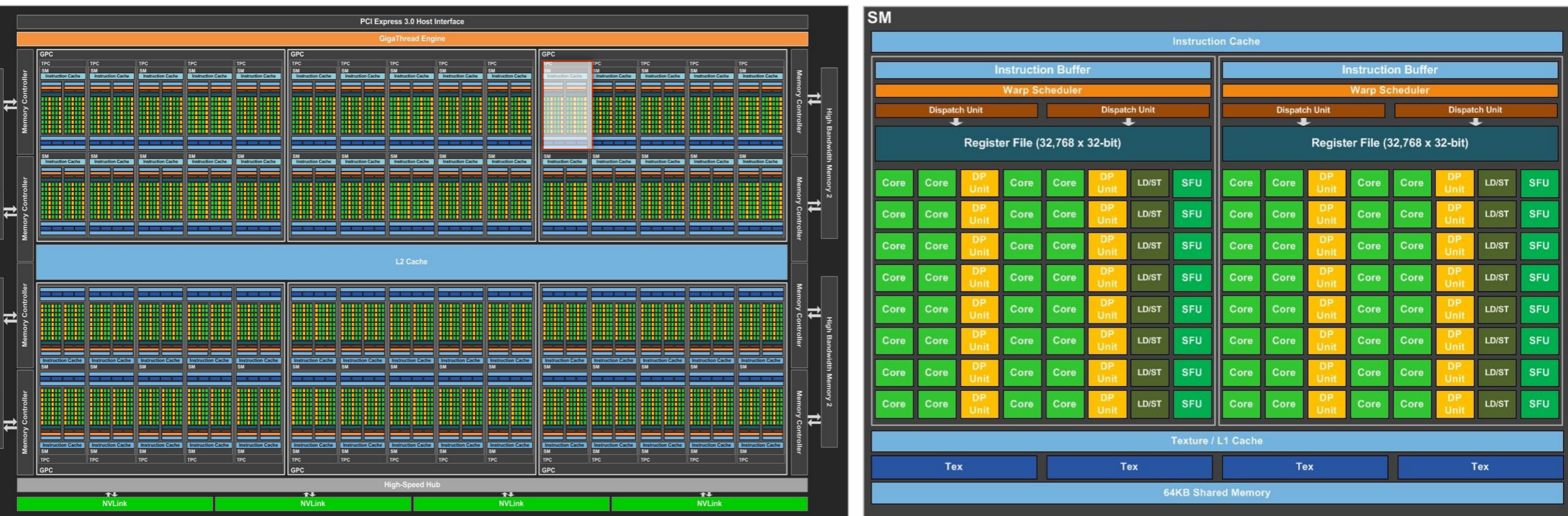
NVIDIA Maxwell GM204 GPU



16 Maxwell cores each have four SIMD clusters with 32 ALUs.
Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)

GPU: Pascal GPU

Professional NVIDIA Pascal GP100 architecture with 60 cores using 16nm fab size



Consumer variants use GDDR5(x) memory with up to 480 GB/s bandwidth and up to 3584 ALUs with peak 10.1 TFLOP/s (SP) 0.3 TFLOP/s (DP)

*60 Pascal GP100 cores each with two SIMD clusters of 32 ALUs (3840 Total).
HBM2 memory streams data at ~1 TB/s and peak 10.6 TFLOP/s (SP), 5.3TFLOP/s (DP)*

GPU: Volta GPU

Professional NVIDIA Pascal GV100 architecture with 84 cores using 12nm fab size



HBM2 memory with **900 GB/s bandwidth** and 5376 ALUs with peak 15.7 TFLOP/s (SP) **7.8 TFLOP/s (DP)**

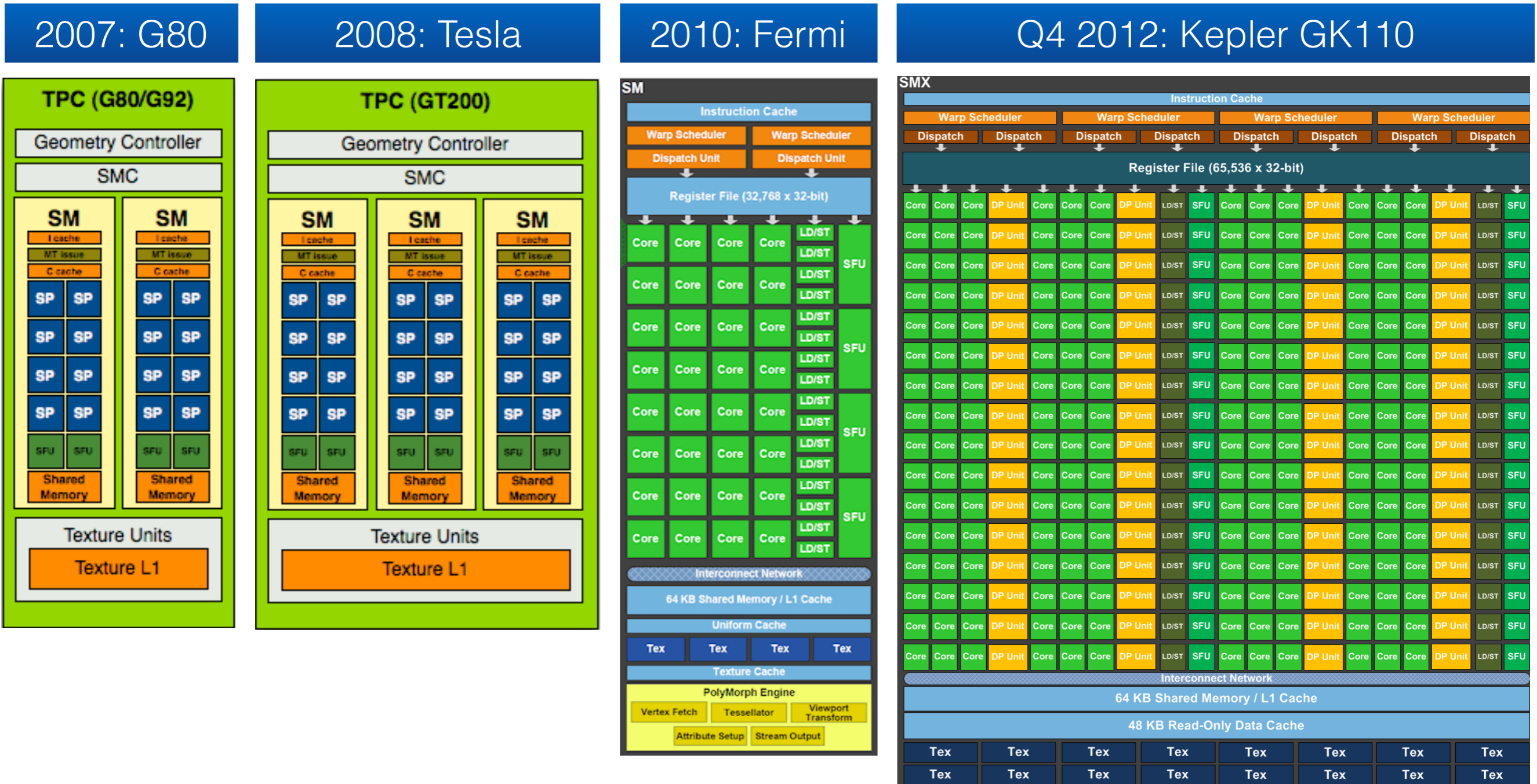


<http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

84 Volta GV100 cores each with four SIMD clusters of 16 ALUs (5120 Total).

GPU: trends in FPU Clusters

The FPU clusters ("core") in 4 NVIDIA generations



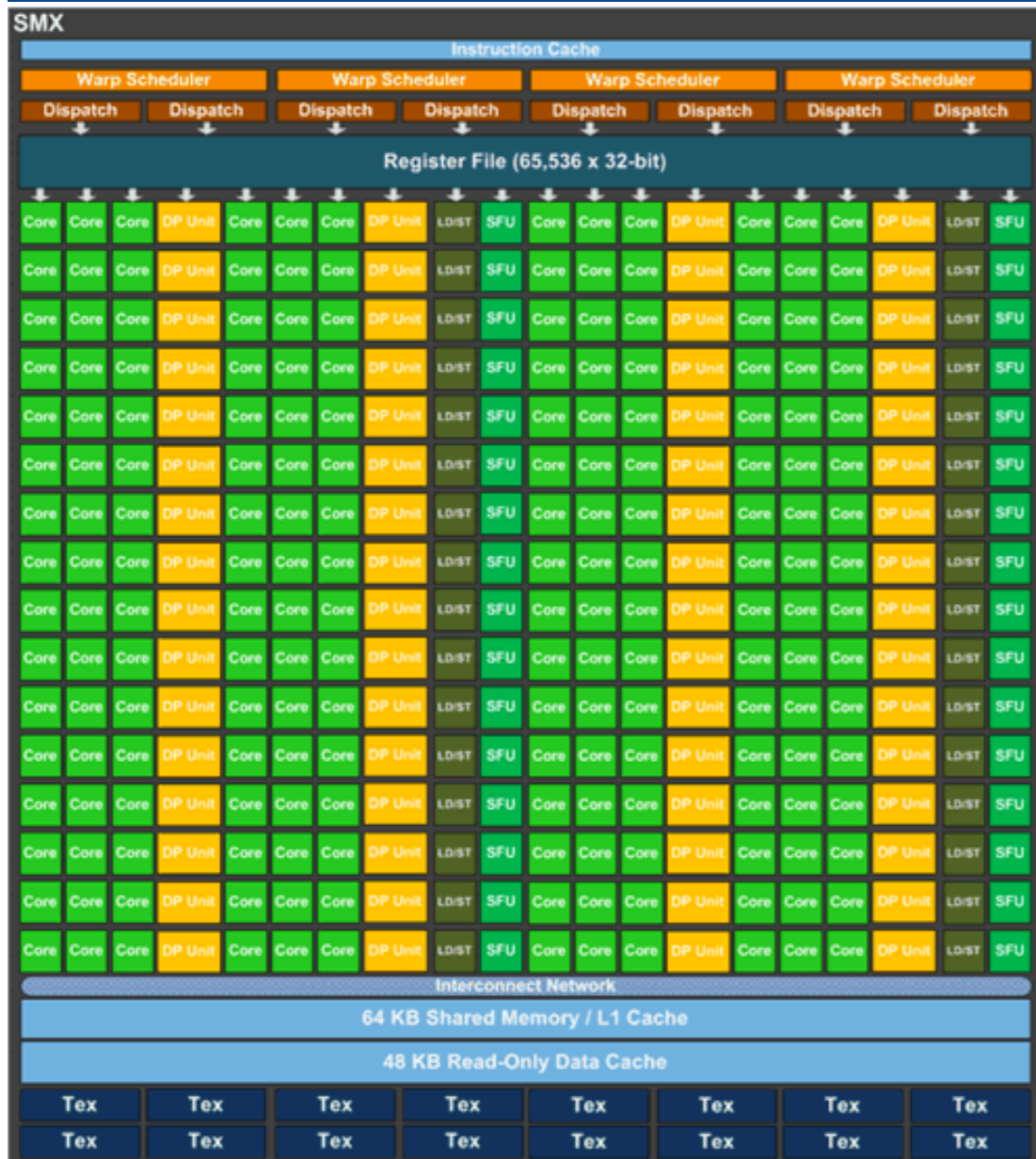
The FPU cluster sizes have ballooned: 16 - 24 - 32 - 192
but the shared memory and register file have not grown accordingly.

Image sources: <http://forum.beyond3d.com/showthread.php?t=58668&page=140>, <http://www.anandtech.com/show/2549/2>, <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>

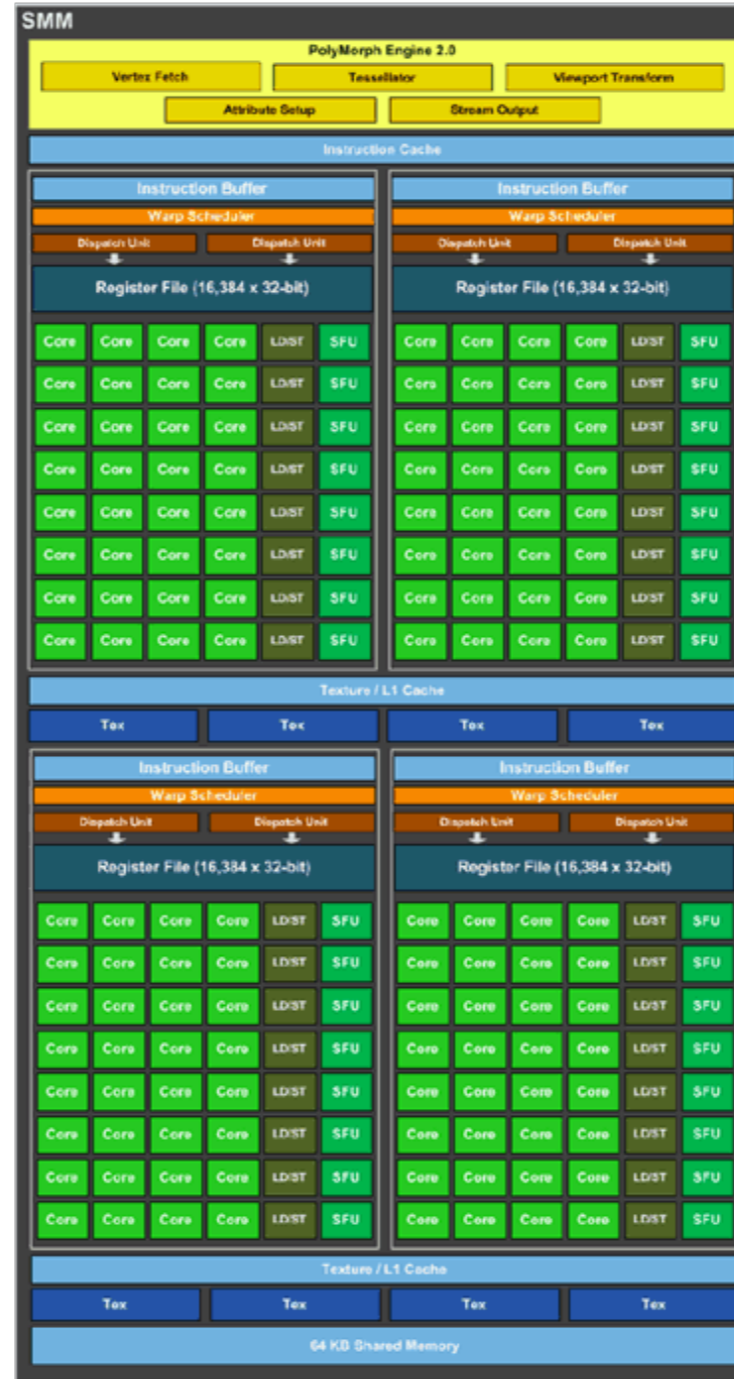
GPU: Kepler to Maxwell to Pascal

The FPU clusters (“core”) in 3 recent NVIDIA processor architectures

Kepler SMX



Maxwell SMM



Pascal SM

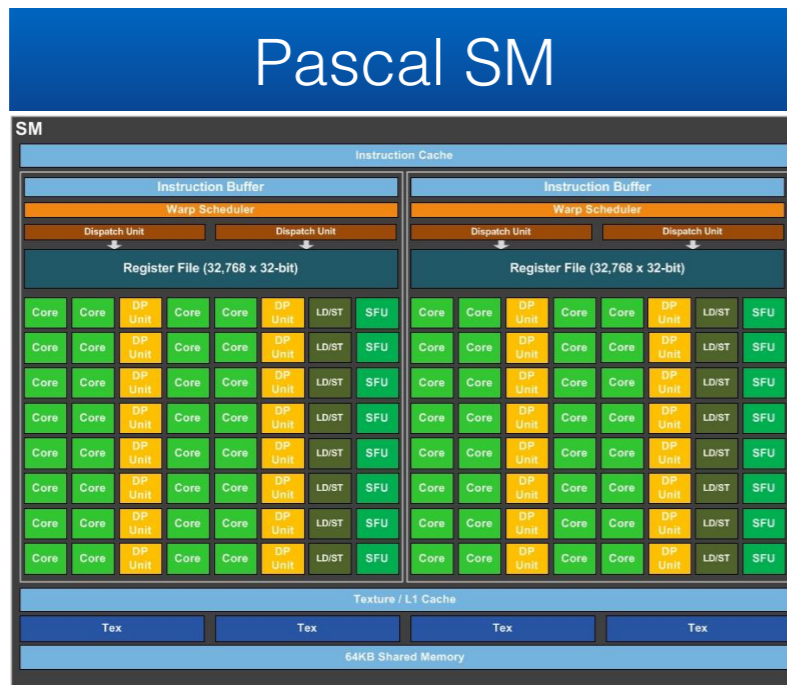


<http://www.ubergizmo.com/2014/02/nvidia-maxwell-gpu-for-geforce-cards/>

Trend: smaller die process yields more space for shared memory and registers.

GPU: Pascal to Volta

The FPU clusters (“core”) in 2 latest NVIDIA processor FP64 heavy architectures

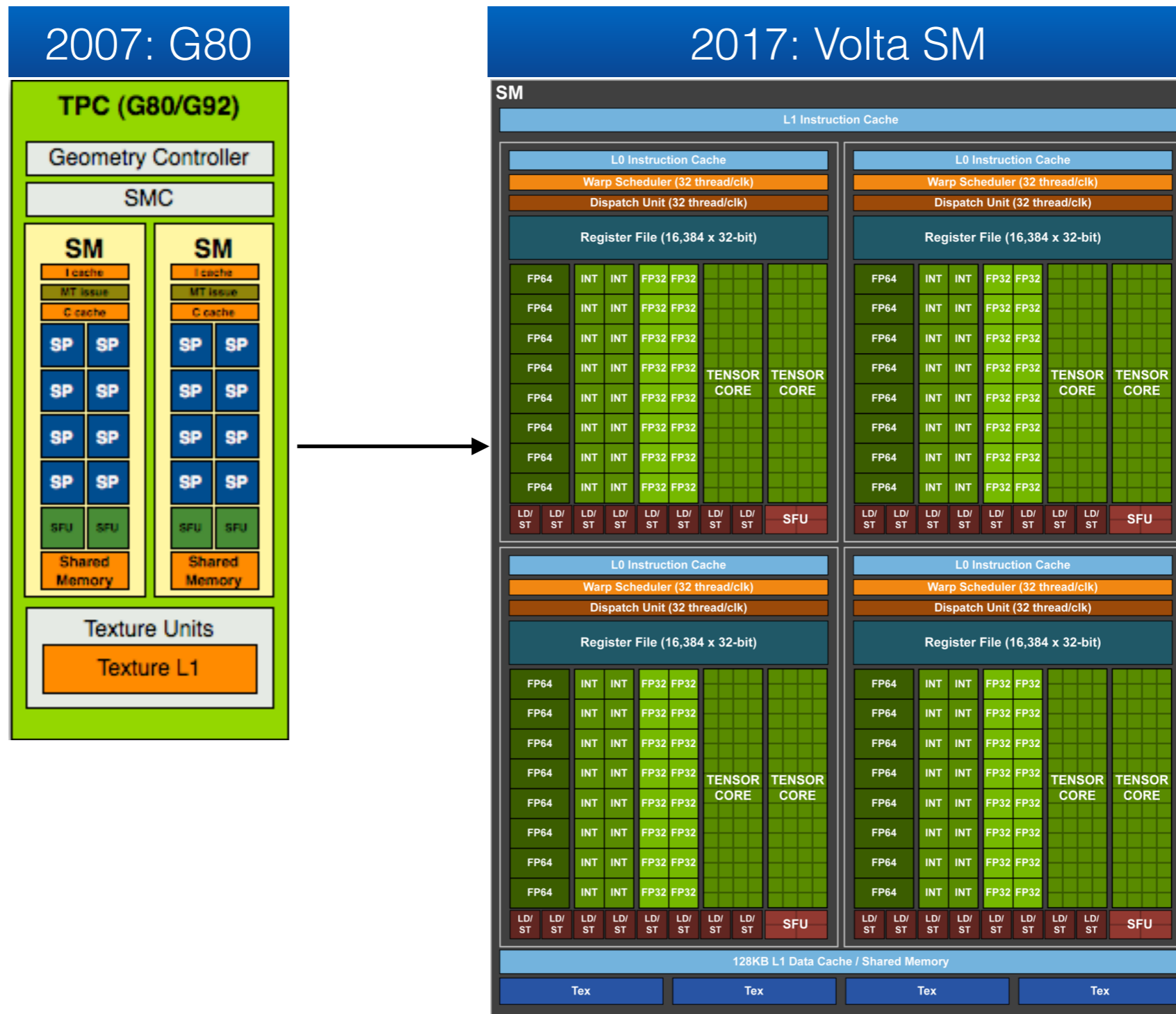


<http://www.ubergizmo.com/2014/02/nvidia-maxwell-gpu-for-geforce-cards/>

Trend: smaller die process yields space for additional half precision “tensor-cores”

GPU: a decade of core architectures

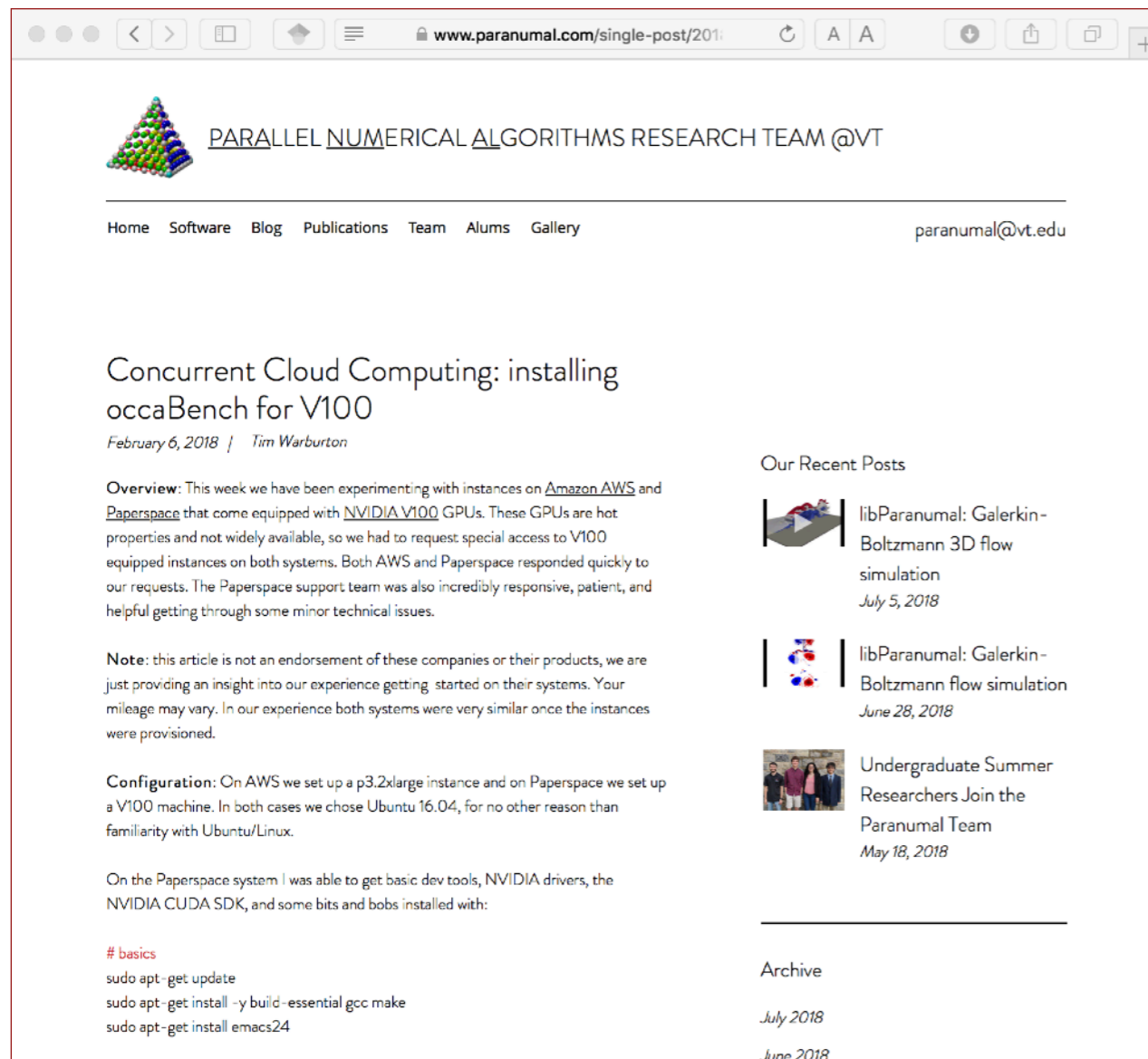
In 10 years the NVIDIA core count & core architecture has scaled remarkably well ...



... their roadmap has been signposted and CUDA codes have scaled
... yet somehow many major HPC codes have not adapted.

Blog: high-order FEM & HPC

Info on using NVIDIA V100 cloud instances/benchmarking/optimizing



www.paranumal.com/single-post/2018-02-06

PARALLEL NUMERICAL ALGORITHMS RESEARCH TEAM @VT

Home Software Blog Publications Team Alums Gallery paranimal@vt.edu

Concurrent Cloud Computing: installing occaBench for V100

February 6, 2018 | Tim Warburton

Overview: This week we have been experimenting with instances on [Amazon AWS](#) and [Paperspace](#) that come equipped with [NVIDIA V100](#) GPUs. These GPUs are hot properties and not widely available, so we had to request special access to V100 equipped instances on both systems. Both AWS and Paperspace responded quickly to our requests. The Paperspace support team was also incredibly responsive, patient, and helpful getting through some minor technical issues.




Note: this article is not an endorsement of these companies or their products, we are just providing an insight into our experience getting started on their systems. Your mileage may vary. In our experience both systems were very similar once the instances were provisioned.

Configuration: On AWS we set up a p3.2xlarge instance and on Paperspace we set up a V100 machine. In both cases we chose Ubuntu 16.04, for no other reason than familiarity with Ubuntu/Linux.

On the Paperspace system I was able to get basic dev tools, NVIDIA drivers, the NVIDIA CUDA SDK, and some bits and bobs installed with:

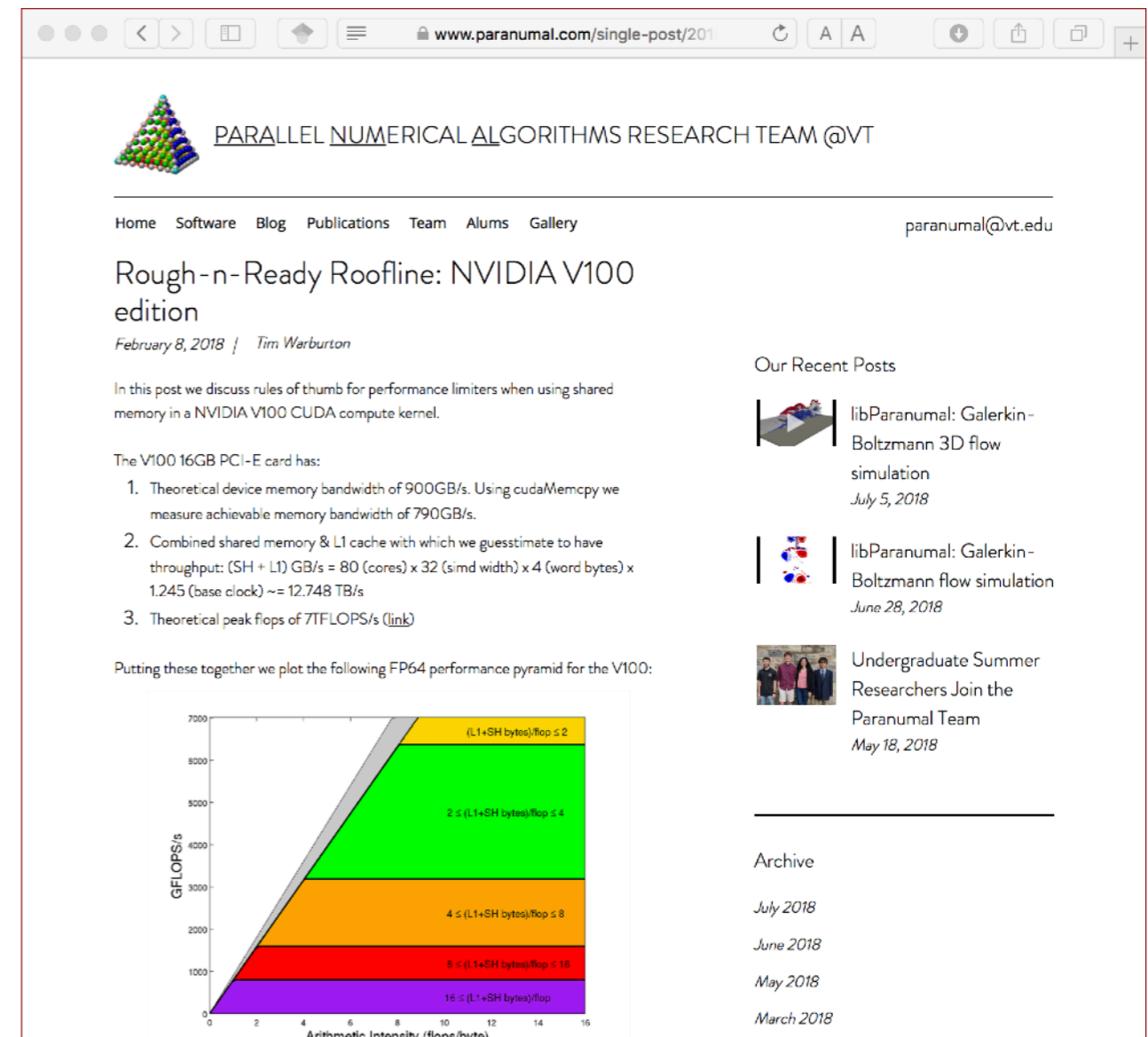
```
# basics
sudo apt-get update
sudo apt-get install -y build-essential gcc make
sudo apt-get install emacs24
```

Our Recent Posts

-  libParanimal: Galerkin-Boltzmann 3D flow simulation
July 5, 2018
-  libParanimal: Galerkin-Boltzmann flow simulation
June 28, 2018
-  Undergraduate Summer Researchers Join the Paranimal Team
May 18, 2018

Archive

- July 2018
- June 2018
- May 2018
- June 2018



www.paranumal.com/single-post/2018-02-08

PARALLEL NUMERICAL ALGORITHMS RESEARCH TEAM @VT

Home Software Blog Publications Team Alums Gallery paranimal@vt.edu

Rough-n-Ready Roofline: NVIDIA V100 edition

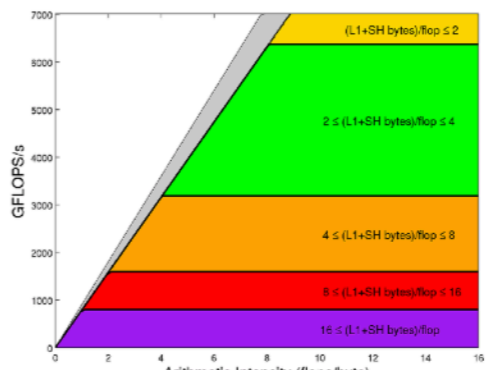
February 8, 2018 | Tim Warburton

In this post we discuss rules of thumb for performance limiters when using shared memory in a NVIDIA V100 CUDA compute kernel.




The V100 16GB PCI-E card has:

- Theoretical device memory bandwidth of 900GB/s. Using cudaMemcpy we measure achievable memory bandwidth of 790GB/s.
- Combined shared memory & L1 cache with which we guesstimate to have throughput: $(SH + L1) \text{ GB/s} = 80 \text{ (cores)} \times 32 \text{ (simd width)} \times 4 \text{ (word bytes)} \times 1.245 \text{ (base clock)} \sim 12.748 \text{ TB/s}$
- Theoretical peak flops of 7TFLOPS/s ([link](#))

Putting these together we plot the following FP64 performance pyramid for the V100:



Our Recent Posts

-  libParanimal: Galerkin-Boltzmann 3D flow simulation
July 5, 2018
-  libParanimal: Galerkin-Boltzmann flow simulation
June 28, 2018
-  Undergraduate Summer Researchers Join the Paranimal Team
May 18, 2018

Archive

- July 2018
- June 2018
- May 2018
- March 2018

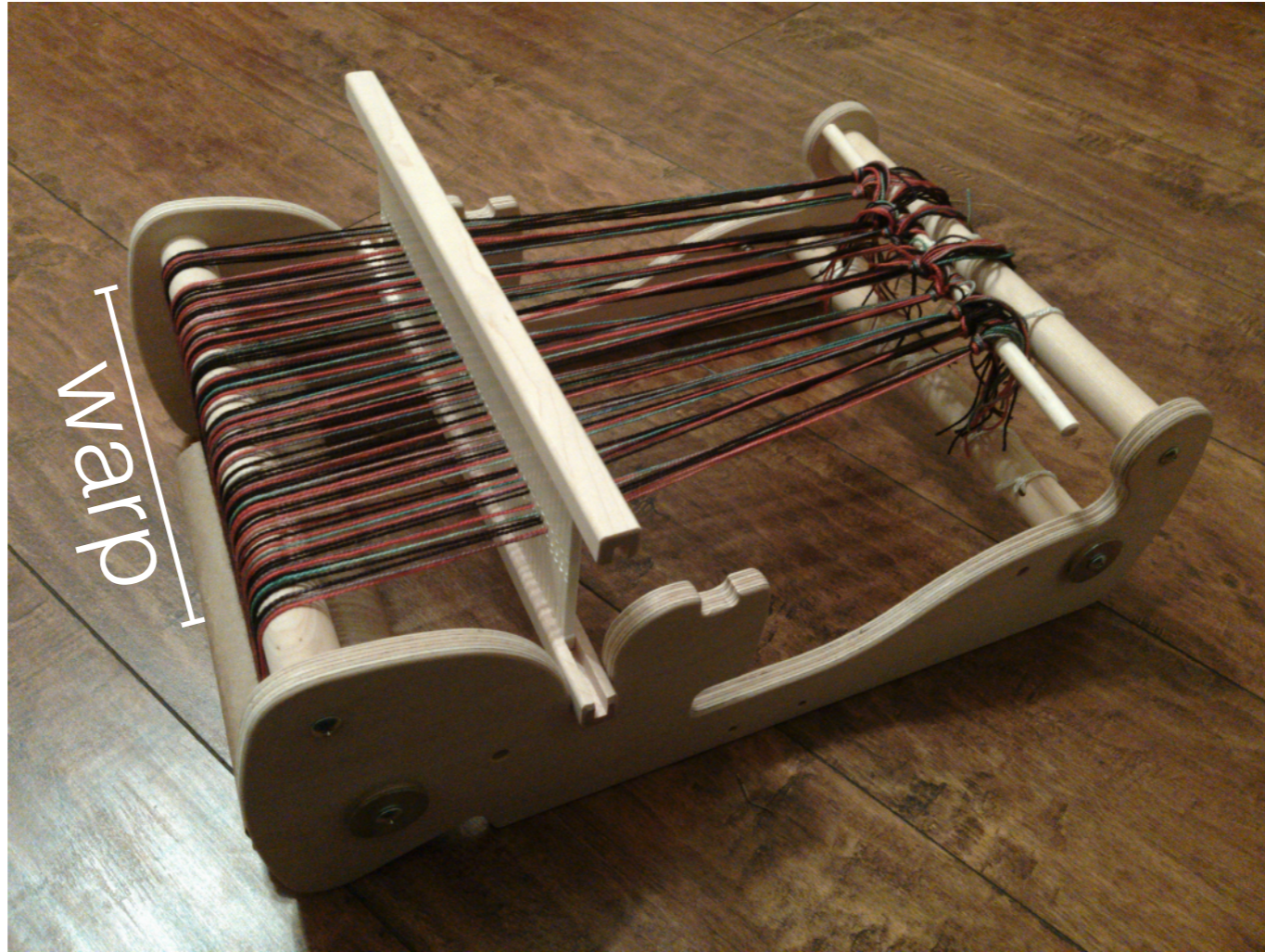
Part 2b: GPU programming with CUDA Threading

NVIDIA's Compute Unified Device Architecture
GPU programming model

Warped Terminology

CUDA

- Is laced (ahem) with terminology derived from weaving like “warp”, “thread”, “texture”.



- We refer instead to a thread array and SIMD groups.

I live with a “weaver” and was just clued in...

CUDA: compute unified device architecture

CUDA was released by NVIDIA in 2007.

The screenshot shows a web browser window with the URL `www.nvidia.com/object/cuda_home_new.html`. The page features the NVIDIA logo, a search bar, and a navigation menu with links for DRIVERS, PRODUCTS, COMMUNITIES, SUPPORT, SHOP, and ABOUT NVIDIA. A prominent green banner reads "CUDA". Below this, a breadcrumb trail indicates the path: NVIDIA Home > Technologies > CUDA Parallel Computing Platform. A "Subscribe" button is visible in the top right. The main content area is divided into three columns. The left column contains sections for "LEARN MORE" (with links to Spotlights, Newsletter, Centers of Excellence, and Women and CUDA), "FOR DEVELOPERS" (with links to Toolkit, Zone, and Conference), and "PRODUCTS" (with links to GeForce and Quadro). The middle column, titled "WHAT IS CUDA?", features a call to action: "Enroll today! Intro to Parallel Programming An open, online course from Udacity" by Dr. John Owens and Dr. David Luebke. Below this is a paragraph defining CUDA as a parallel computing platform. The right column, titled "FOR DEVELOPERS", encourages visiting the CUDA Zone and offers a free test drive of NVIDIA TESLA K40 GPUs, with buttons for "GO TO CUDA ZONE" and "REGISTER".

CUDA is used to program NVIDIA GPUs.

CUDA includes a HOST API and a DEVICE kernel programming language.

CUDA: offload model

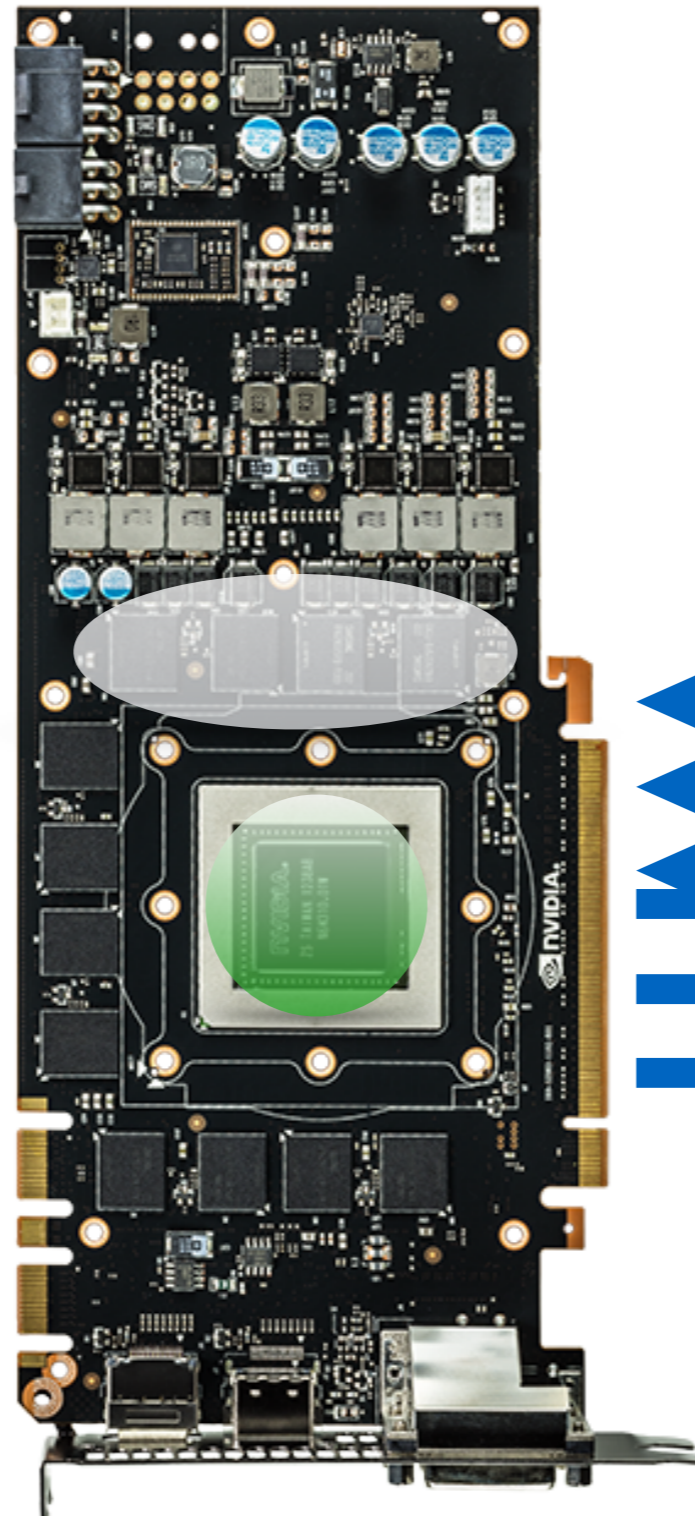
The programmer explicitly moves data between HOST and DEVICE

1. cudaMalloc: allocate memory for a DEVICE array

3. Queue kernel task on DEVICE

2. cudaMemcpy: copy data from HOST to DEVICE array

4. cudaMemcpy: copy data from DEVICE to HOST array



<https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780>

*Key observation: the DEVICE and HOST are asynchronous.
Operations are queued on the DEVICE.*

GPU: natural thread model

The GPU architecture admits a natural parallel threading model

- Programmer partitions a compute task into kernel code:
 - Programmer assigns kernel code to independent work-blocks:
 - Work-block assigned to a core with sufficient resources to process it:
 - Each core processes work-block kernel code with a work-group of “threads”
 - The work-group is batch processed in sub-groups of SIMD* work-items.
 - Each work-item processed by a “thread” passing through a SIMD lane.
 - A stalling SIMD group of “threads” is idled until it can continue.
 - “Threads” in a work-group can collaborate through shared memory.
 - The work-block stays resident until completed by core (using resources).
- Main assumption: same instructions for independent work-groups.

* SIMD here is the number of ALUs in one of the core's vector unit.

CUDA: example HOST code

Overview of C-like CUDA code that runs on the HOST:

simpleKernel.cu

```
#include "cuda.h"

int main(int argc, char **argv){
    int N = 3789; // size of array for this DEMO

    float *d_a; // Allocate DEVICE array
    cudaMalloc((void**) &d_a, N*sizeof(float));

    int B = 512;
    dim3 dimBlock(B,1,1); // 512 threads per thread-block
    dim3 dimGrid((N+B-1)/B, 1, 1); // Enough thread-blocks to cover N

    // Queue kernel on DEVICE
    simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);

    // HOST array
    float *h_a = (float*) calloc(N, sizeof(float));

    // Transfer result from DEVICE array to HOST array
    cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Print out result from HOST array
    for(int n=0;n<N;++n) printf("h_a[%d] = %f\n", n, h_a[n]);
}
```

Note the .cu file extension.

We use NVIDIA's CUDA Compiler nvcc to compile .cu files.

CUDA: host code

Overview of C-like HOST code for a simple *kernel* that fills a vector of length N

1. Allocate array space on DEVICE:

```
float *d_a; // Allocate DEVICE array (pointers used as array handles)
cudaMalloc((void**) &d_a, N*sizeof(float));
```

2. Design thread-array:

```
dim3 dimBlock(512,1,1); // 512 threads per thread-block
dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N
```

3. Queue compute task on DEVICE:

```
// specify number of threads with <<< block count, thread count >>>
simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

4. Copy results from DEVICE to HOST:

```
float *h_a = (float*) calloc(N, sizeof(float));
cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost)
```

Key API calls: cudaMalloc, cudaMemcpy

CUDA: motivating serial function

Before jumping into how to write a CUDA kernel we consider first a serial function that fills an array with entries 0:N-1

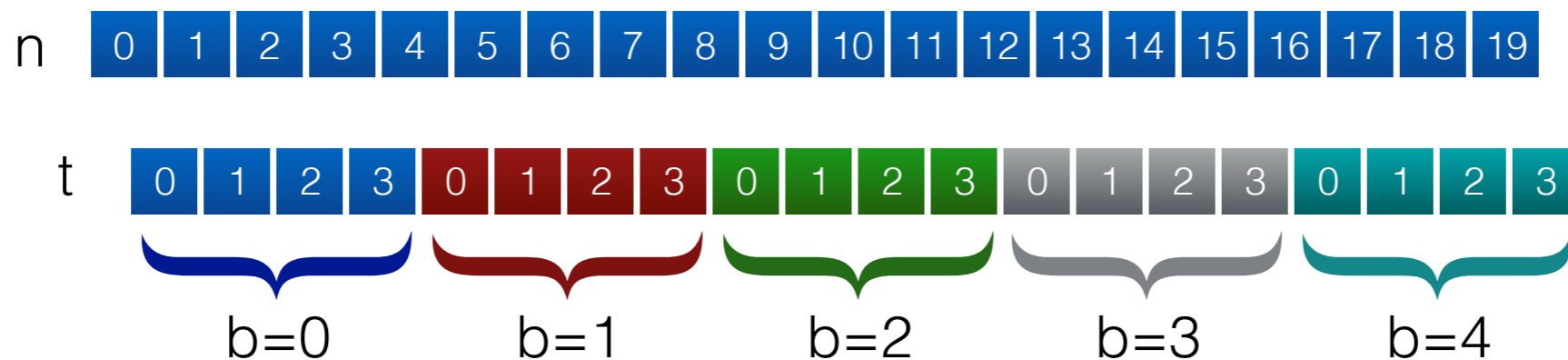
```
void serialSimpleKernel(int N, float *d_a){  
    for(n=0;n<N;++n){ // loop over N entries  
        d_a[n] = n;  
    }  
}
```

To make a two level thread parallel implementation we partition (or chunk) the n-loop

CUDA: motivating serial function

Consider the case with $N=20$ - then break the for loop into independent tiles:

```
void serialSimpleKernel(int N, float *d_a){  
    for(n=0;n<N;++n){ // loop over N entries  
        d_a[n] = n;  
    }  
}
```

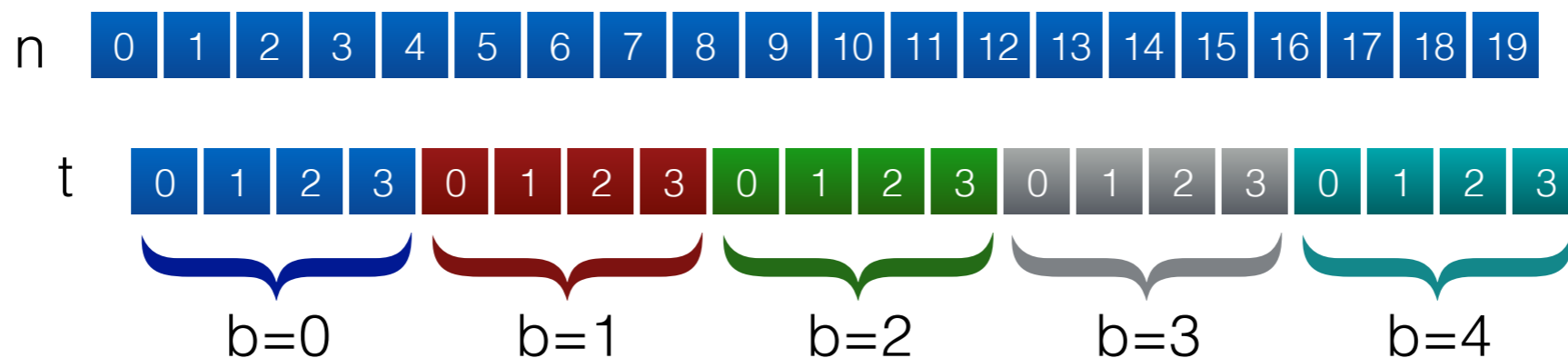


*We can think of splitting the n -loop into tiles of size 4: $n=t+4b$.
Here: block dimension = 4 and grid dimension = 5.*

CUDA: serial function with loop tiling

We tile the n-loop into equal sized tiles (here tile size is blockDim)

```
void tiledSerialSimpleKernel(int N, float *d_a){  
    for(int b=0;b<gridDim;++b){ // loop over blocks  
        for(int t=0;t<blockDim;++t){// loop inside block  
            // Convert thread and thread-block indices into array index  
            const int n = t + b*blockDim;  
            // If index is in [0,N-1] add entries  
            if(n<N) // guard against an inexact tiling  
                d_a[n] = n;  
        }  
    }  
}
```



We assume the loop boundaries (gridDim and blockDim) are externally specified variables. We also assume that: $N \leq \text{gridDim} * \text{blockDim}$. Tiling also referred to chunking sometimes.

CUDA: tiled serial function

We rename variables to conform with CUDA naming convention.
dim3 type intrinsic variables: threadIdx, blockDim, blockIdx, gridDim

```
void tiledSerialSimpleKernel(int N, float *d_a){  
    for(blockIdx.x=0;blockIdx.x<gridDim.x;++blockIdx.x){ // loop over blocks  
        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block  
            // Convert thread and thread-block indices into array index  
            const int n = threadIdx.x + blockDim.x*blockIdx.x;  
  
            // If index is in [0,N-1] add entries  
            if(n<N)  
                d_a[n] = n;  
        }  
    }  
}
```

Key observation: the body of the tiled loop can now be mapped to a thread.

*We also assume that: $N \leq \text{gridDim.x} * \text{blockDim.x}$*

CUDA: multi-dimensional thread rank

Each thread can determine its (multi-dimensional) rank with respect to both its rank in the thread-block and the rank of the thread-block itself.

	Intrinsic variables		
Description	Fastest index		Slowest index
Thread indices in thread-block	threadIdx.x	threadIdx.y	threadIdx.z
Dimensions of thread-block	blockDim.x	blockDim.y	blockDim.z
Block indices.	blockIdx.x	blockIdx.y	blockIdx.z *
Dimensions of grid of thread-blocks	gridDim.x	gridDim.y	gridDim.z *

Remember: we can identify task parallelism by associating tasks with combination of thread-index and block-index.

Best practice: avoid frequent branching based on threadIdx or blockIdx.

** three dimensional grid of thread-blocks supported as of CUDA 2.**

CUDA: limitations

The CUDA compute capability evolves with ongoing NVIDIA GPU hardware revisions.

Technical specifications	Compute capability (version)							
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	5.0
Maximum dimensionality of grid of thread blocks	2				3			
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535					2 ³¹⁻¹		
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512				1024			
Maximum z-dimension of a block	64							
Maximum number of threads per block	512				1024			
Warp size	32							
Maximum number of resident blocks per multiprocessor	8					16		32
Maximum number of resident warps per multiprocessor	24	32		48	64			
Maximum number of resident threads per multiprocessor	768	1024		1536	2048			
Number of 32-bit registers per multiprocessor	8 K	16 K		32 K	64 K			
Maximum number of 32-bit registers per thread	128				63	255		
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB
Number of shared memory banks	16				32			

Table credit: CUDA wikipedia page (<http://en.wikipedia.org/wiki/CUDA>)

CUDA: tiled serial function

We rename variables to conform with CUDA naming convention.
dim3 type intrinsic variables: threadIdx, blockDim, blockIdx, blockDim

```
void tiledSerialSimpleKernel(int N, float *d_a){  
    for(blockIdx.x=0;blockIdx.x<gridDim.x;++blockIdx.x){ // loop over blocks  
        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block  
            // Convert thread and thread-block indices into array index  
            const int n = threadIdx.x + blockDim.x*blockIdx.x;  
  
            // If index is in [0,N-1] add entries  
            if(n<N)  
                d_a[n] = n;  
        }  
    }  
}
```

Key observation: the body of the tiled loop can now be mapped to a thread.

*We also assume that: $N \leq \text{gridDim.x} * \text{blockDim.x}$*

CUDA: simple array operation kernel

```
// HOST code to queue kernel  
simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

`__global__` specifies kernel

```
__global__ void simpleKernel(int N, float *d_a){  
    // Convert thread and thread-block indices into array index  
    const int n = threadIdx.x + blockDim.x*blockIdx.x;  
    // If index is in [0,N-1] add entries  
    if(n<N)  
        d_a[n] = n;  
}
```

ThreadIdx & blockIdx determine thread rank that is mapped to array index

Action performed by each thread

Key observation: the loops are implicitly executed by thread parallelism and *do not* appear in the CUDA kernel code.

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.

Code Along: CUDA Hello World

$$c_n = a_n + b_n \text{ for } n = 0, \dots, N - 1$$

Embarrassingly parallel...

CUDA: offload model

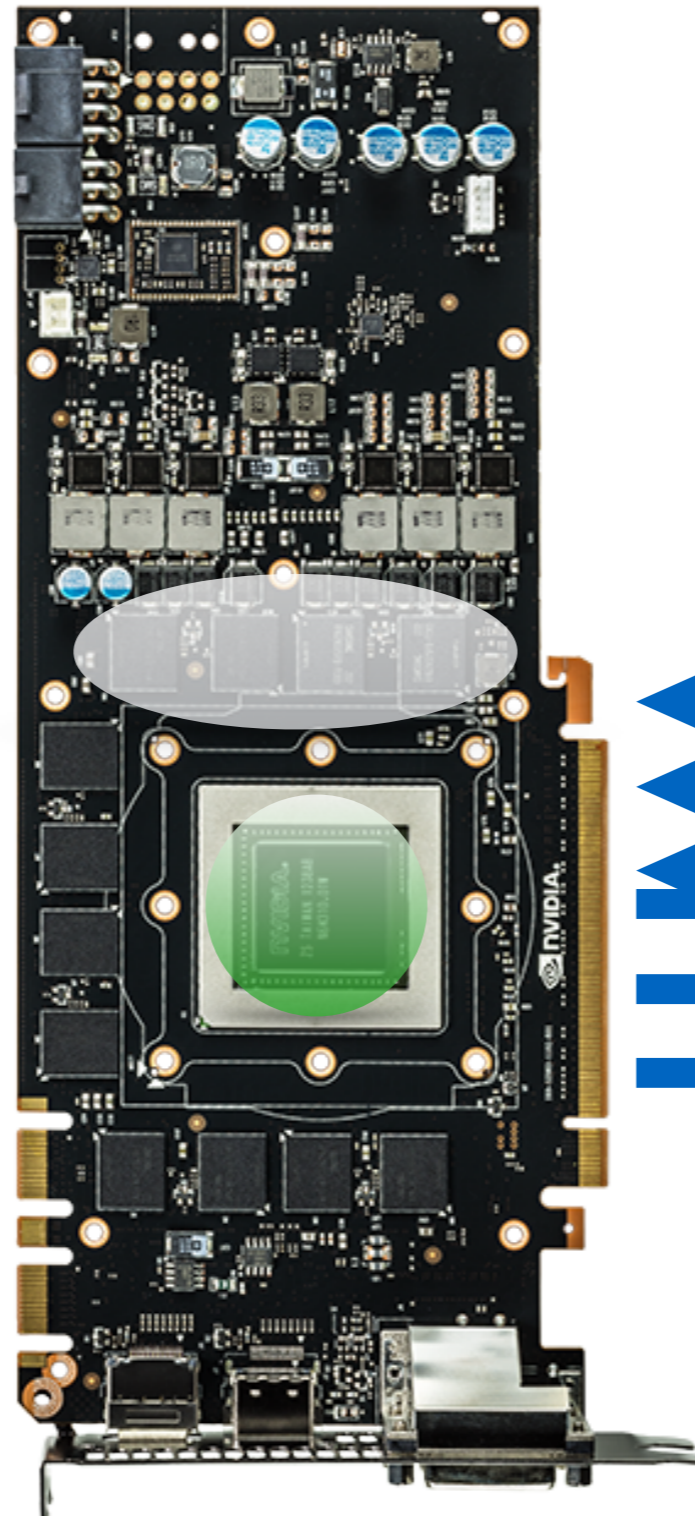
The programmer explicitly moves data between HOST and DEVICE

1. `cudaMalloc`: allocate memory for a DEVICE array

3. Queue kernel task on DEVICE

2. `cudaMemcpy`: copy data from HOST to DEVICE array

4. `cudaMemcpy`: copy data from DEVICE to HOST array



<https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780>

Key observation: the DEVICE and HOST are asynchronous.
Operations are queued on the DEVICE.

Code Along: diving straight into CUDA

Code along demo:

CUDA code to add two vectors together from scratch !!!

You can find a pre-made version here:

<https://github.com/tcew/ATPESC18/tree/master/examples/cuda/addVectors>

Wacky CUDA syntax used:

Thread rank and size info:	threadIdx.x, blockIdx.x, blockDim.x
DEVICE function (kernel) annotation:	__global__
Allocating/freeing a DEVICE array:	cudaMalloc, cudaFree
Copy data between DEVICE and HOST:	cudaMemcpy
Copy direction:	cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost

Kernel launch : `addVectorsKernel <<< dimGrid, dimBlock >>> (N, d_a, d_b, d_c);`

See instruction sheet handout for set up instructions on the cooley system.

Hands On #0: reverse an array in CUDA

Adapt the CUDA code to reverse the entries in an array:

$$b_n = a_{(N-1-n)} \text{ for } n = 0, \dots, N - 1$$

Start with your code along code, or use the pre-baked version:

<https://github.com/tcew/ATPESC18/tree/master/examples/cuda/addVectors>

Things to pay attention to:

1. Make sure you copy back the correct CUDA DEVICE array to the HOST.
2. How many threads should you use to avoid read-write race conflicts ?
3. Change the number of threads.
4. Print the whole b array after the kernel.

10 minutes...

Hands On #1: compiling/running CUDA

This example requires CUDA GPU, drivers, and SDK is installed.

```
cooley.alcf.anl.gov: make sure .soft.cooley includes and resoft
+mvapich2
+cuda-7.5.18
+ffmpeg-1.0.1
@default
```

```
# clone the examples on the login node:
git clone https://github.com/tcew/ATPESC18

# if you haven't already done so, queue an interactive job request:
qsub -A ATPESC2018 -I -n 1 -t 120 -q training

# find the source
cd ATPESC18/examples/cuda/simple

# compile on node with the NVIDIA CUDA compiler (nvcc) installed
nvcc -o simple simple.cu

# run on node with the NVIDIA CUDA runtime libraries installed
./simple
```

Make sure you can complete this exercise now if possible !
Source code: <https://github.com/tcew/ATPESC18/examples/cuda/simple>

CUDA: multi-dimensional tiled serial function

CUDA supports up to 3 nested outer “block” loops,
with a sequence of 3 nested inner “thread” loops

```
void tiledSerialMultidKernel(int N, float *d_a){  
  
    for(blockIdx.z=0;blockIdx.z<gridDim.z;++blockIdx.z){        // loop over z-blocks  
        for(blockIdx.y=0;blockIdx.y<gridDim.y;++blockIdx.y){    // loop over y-blocks  
            for(blockIdx.x=0;blockIdx.x<gridDim.x;++blockIdx.x){ // loop over x-blocks  
  
                // loop over thread indices in thread-block  
                for(threadIdx.z=0;threadIdx.z<blockDim.z;++threadIdx.z){  
                    for(threadIdx.y=0;threadIdx.y<blockDim.y;++threadIdx.y){  
                        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){  
  
                            // Convert thread and thread-block indices into array index  
                            const int nx = threadIdx.x + blockDim.x*blockIdx.x;  
                            const int ny = threadIdx.y + blockDim.y*blockIdx.y;  
                            const int nz = threadIdx.z + blockDim.z*blockIdx.z;  
  
                            // Perform action based on thread-ranks  
                            ...;  
  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Key observation: the iterations in each iteration are assumed to be independent

CUDA: multi-d array operation kernel

```
// HOST code to queue kernel
dim3 dimGrid(GX,GY,GZ), dimBlock(BX,BY, BZ);
multidKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

```
__global__ void multidKernel(int N, float *d_a){
```

```
// Convert thread and thread-block indices into array index
const int nx = threadIdx.x + blockDim.x*blockIdx.x;
const int ny = threadIdx.y + blockDim.y*blockIdx.y;
const int nz = threadIdx.z + blockDim.z*blockIdx.z;
```

```
operations based on thread ranks;
```

```
}
```

Key observation: the loops are implicitly executed by thread parallelism and *do not* appear in the CUDA kernel code.

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.

CUDA: elliptic solver example

We consider a more substantial example: solving the Poisson problem.

Elliptic Poisson problem:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ in } \Omega = [-1,1] \times [-1,1]$$
$$u = 0 \text{ on } \partial\Omega$$

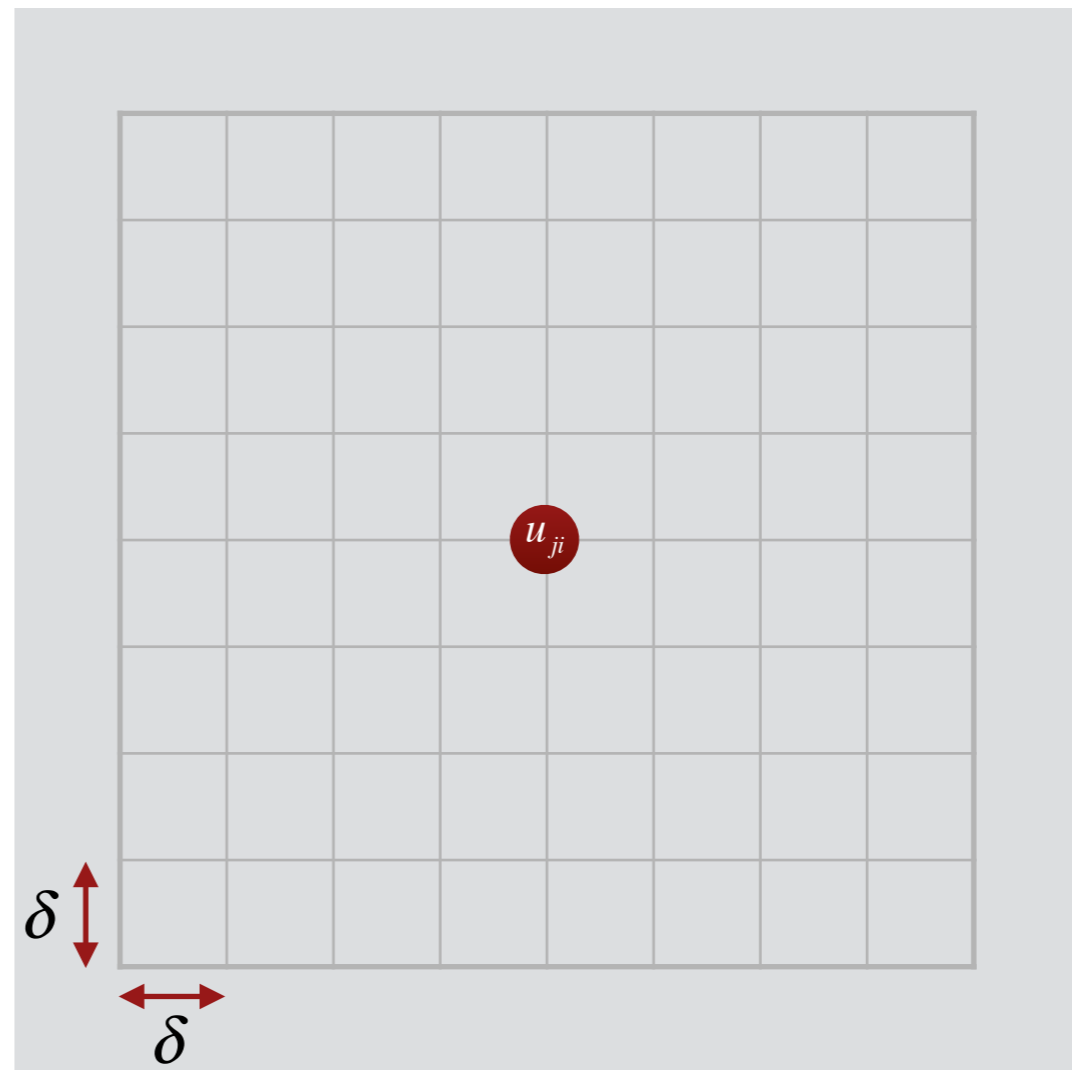
Poisson problem is an archetypal building block for many physics packages.

CUDA: elliptic solver example

Elliptic Poisson problem:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ in } \Omega = [-1,1] \times [-1,1]$$

$$u = 0 \text{ on } \partial\Omega$$



We represent the numerical solution at a regular grid of finite-difference nodes.

CUDA: elliptic solver example

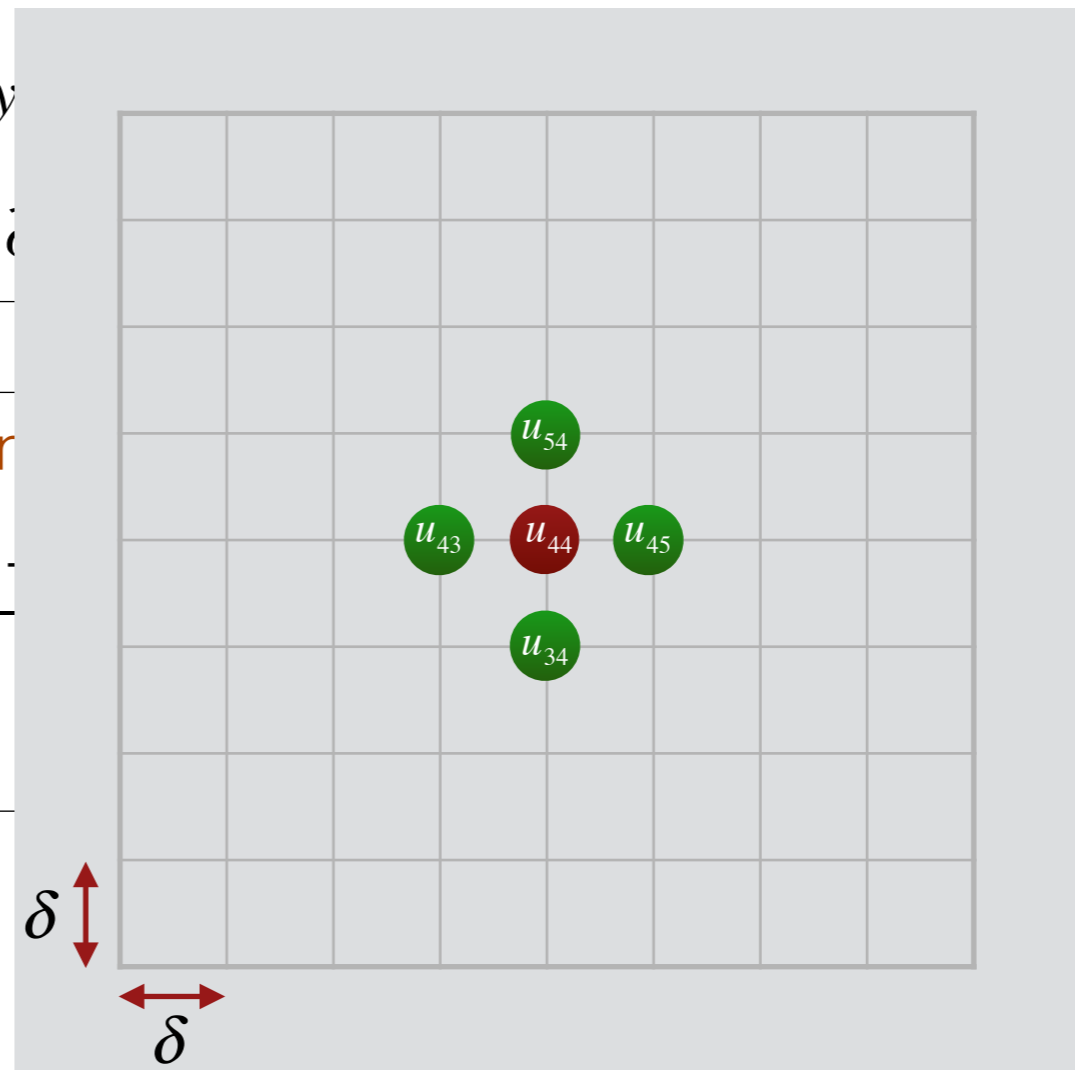
First step discretize the equations into a set of linear constraints.

Elliptic Poisson problem:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$
$$u = 0 \text{ on } \partial \Omega$$

Discrete Poisson problem (assuming

$$\left(\frac{u_{j(i+1)} - 2u_{ji} + u_{j(i-1)}}{\delta^2} \right) + \left(\frac{u_{(j+1)i} - 2u_{ji} + u_{(j-1)i}}{\delta^2} \right)$$



*The derivative operators are approximated by second order differences.
The discrete Poisson problem is approximated at the finite difference nodes.*

CUDA: discrete elliptic example

We solve the linear system for the unknowns using the stationary iterative Jacobi method

Discrete Poisson problem (assuming Cartesian grid):

$$\left(\frac{u_{j(i+1)} - 2u_{ji} + u_{j(i-1)}}{\delta^2} \right) + \left(\frac{u_{(j+1)i} - 2u_{ji} + u_{(j-1)i}}{\delta^2} \right) = f_{ji} \text{ for } i, j = 1, \dots, N$$
$$u_{ji} = 0 \text{ for } i = 0, N+1 \text{ or } j = 0, N+1$$

Jacobi iteration for discrete Poisson problem:

$$\left(\frac{u_{j(i+1)}^k - 2u_{ji}^{k+1} + u_{j(i-1)}^k}{\delta^2} \right) + \left(\frac{u_{(j+1)i}^k - 2u_{ji}^{k+1} + u_{(j-1)i}^k}{\delta^2} \right) = f_{ji} \text{ for } i, j = 1, \dots, N$$
$$u_{ji} = 0 \text{ for } i = 0, N+1 \text{ or } j = 0, N+1$$

*Yes, this is not the best way to solve the problem.
But it is simple.*

CUDA: elliptic solver example

Rearranging we are left with a simple five point recurrence:

Jacobi iteration for discrete Poisson problem:

$$\left(\frac{u_{j(i+1)}^k - 2u_{ji}^{k+1} + u_{j(i-1)}^k}{\delta^2} \right) + \left(\frac{u_{(j+1)i}^k - 2u_{ji}^{k+1} + u_{(j-1)i}^k}{\delta^2} \right) = f_{ji} \text{ for } i, j = 1, \dots, N$$

$$u_{ji} = 0 \text{ for } i = 0, N + 1 \text{ or } j = 0, N + 1$$

Iterate:

$$u_{ji}^{k+1} = \frac{1}{4} \left(-\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \dots, N$$

while:

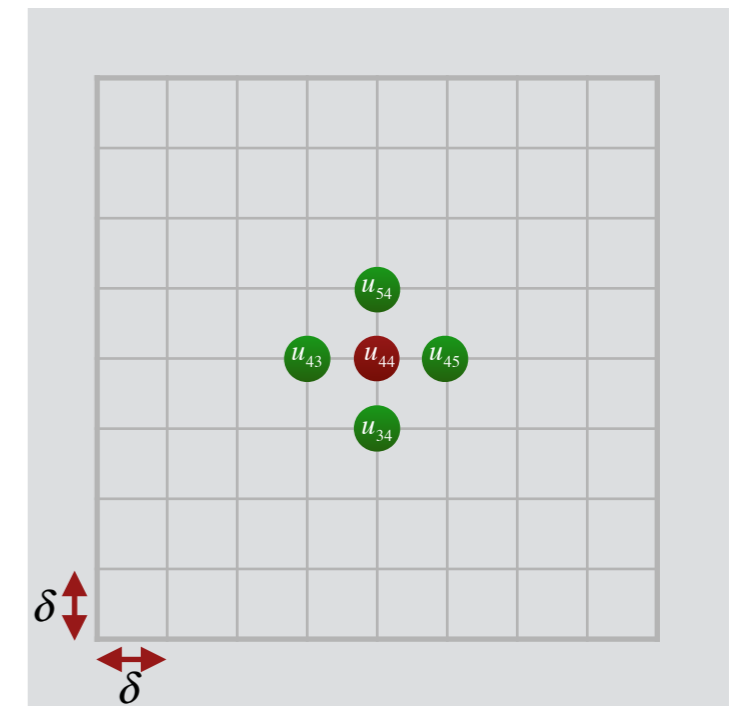
$$\varepsilon := \sqrt{\sum_{i=1}^{i=N} \sum_{j=1}^{j=N} \left(u_{ji}^{k+1} - u_{ji}^k \right)^2} > tol$$

CUDA: parallelism for solver example

For the iterate step we note:
each node can update independently for maximum parallelism.

Iterate:

$$u_{ji}^{k+1} = \frac{1}{4} \left(-\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \dots, N$$



The GPU works best when every thread is doing the same thing.

CUDA: serial Jacobi iteration

The explicit serial loop structure for the Jacobi iteration shows no loop carry dependence:

Iterate:

$$u_{ji}^{k+1} = \frac{1}{4} \left(-\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \dots, N$$

Serial kernel:

```
void jacobi(const int N,
            const datafloat *rhs,
            const datafloat *u,
            datafloat *newu){
    for(int i=0; i<N; ++i){
        for(int j=0; j<N; ++j){

            // Get linear index into NxN
            // inner nodes of (N+2)x(N+2) grid
            const int id = (j + 1)*(N + 2) + (i + 1);

            newu[id] = 0.25f*(rhs[id]
                               + u[id - (N+2)]
                               + u[id + (N+2)]
                               + u[id - 1]
                               + u[id + 1]);
        }
    }
}
```

Note: we use an NxN array of threads and change leave the edge nodes unchanged.

*At the start we set: rhs=-delta*delta*f*

CUDA: parallel Jacobi iteration

For CUDA: each thread can update a node independently for maximum parallelism.

Iterate:

$$u_{ji}^{k+1} = \frac{1}{4} \left(-\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \dots, N$$

CUDA kernel:

```
__global__ void jacobi(const int N,
                      const datafloat *rhs,
                      const datafloat *u,
                      datafloat *newu){

    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;

    // Check that this is a legal node
    if((i < N) && (j < N)){

        // Get linear index onto (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);

        newu[id] = 0.25f*(rhs[id]
                          + u[id - (N+2)]
                          + u[id + (N+2)]
                          + u[id - 1]
                          + u[id + 1]);

    }
}
```

Note: we use an NxN array of threads and leave the edge nodes unchanged.

*At the start we set: rhs=-delta*delta*f*

<https://github.com/tcew/ATPESC18/tree/master/examples/cuda/jacobi>

CUDA: parallelism for solver reduction

To make this more parallel we need to split the termination into CUDA thread-blocks:

Reduction:

$$\varepsilon := \sum_{i=0}^{i=N-1} v_i$$

Block reduction (B blocks)

$$\varepsilon := \sum_{b=0}^{b=B-1} \left(\sum_{i=0}^{i=T-1} v_{i+bT} \right)$$

$$B := \frac{N}{T}$$

Next we need to distribute the inner sum work over the threads in each of the B thread-blocks.

CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```
__global__ void partialReduceResidual(const int entries,
                                     datafloat *u,
                                     datafloat *newu,
                                     datafloat *blocksum){
    __shared__ datafloat s_blocksum[BDIM];
    const int id = blockIdx.x*blockDim.x + threadIdx.x;

    s_blocksum[threadIdx.x] = 0;

    if(id < entries){
        const datafloat diff = u[id] - newu[id];
        s_blocksum[threadIdx.x] = diff*diff;
    }

    int alive = blockDim.x;
    int t = threadIdx.x;

    while(alive>1){
        __syncthreads(); // barrier (make sure s_blocksum is ready)
        alive /= 2; // reduce active threads
        if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
    }

    if(t==0)
        blocksum[blockIdx.x] = s_blocksum[0];
}
```

Step \ Thread	0	1	2	3	4	5	6
0	1	1+7	8	8+11	19	19+17	36
1	3	3+8	11	11+6	17		
2	5	5+6	11				
3	2	2+4	6				
4	7						
5	8						
6	6						
7	4						

Target: $\sum_{i=0}^{i=T-1} v_i$

Here the `__shared__` array is read/writeable only by threads in the same thread-block.
All threads in the thread-block have to enter the `__syncthreads()` before any of them can continue.

CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```
__global__ void partialReduceResidual(const int entries,
                                     datafloat *u,
                                     datafloat *newu,
                                     datafloat *blocksum){

    __shared__ datafloat s_blocksum[BDIM];

    const int id = blockIdx.x*blockDim.x + threadIdx.x;

    s_blocksum[threadIdx.x] = 0;

    if(id < entries){
        const datafloat diff = u[id] - newu[id];
        s_blocksum[threadIdx.x] = diff*diff;
    }

    int alive = blockDim.x;
    int t = threadIdx.x;

    while(alive>1){
        __syncthreads(); // barrier (make sure s_blocksum is ready)

        alive /= 2; // reduce active threads
        if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
    }

    if(t==0)
        blocksum[blockIdx.x] = s_blocksum[0];
}
```

Step \ Thread	0	1	2	3	4	5	6
0	1	1+7	8	8+11	19	19+17	36
1	3	3+8	11	11+6	17		
2	5	5+6	11				
3	2	2+4	6				
4	7						
5	8						
6	6						
7	4						

Target: $\sum_{i=0}^{i=T-1} v_i$

Here the `__shared__` array is read/writeable only by threads in the same thread-block.
All threads in the thread-block have to enter the `__syncthreads()` before any of them can continue.

CUDA: parallel reduction

Source

The screenshot shows a GitHub repository page for 'tcew / ATPESC16'. The browser address bar shows the URL 'github.com/tcew/ATPESC16/tree/master/examples/cuda/reduction'. The repository name is 'tcew / ATPESC16'. The current branch is 'master'. The path is 'ATPESC16 / examples / cuda / reduction /'. The commit history table shows the following files and their commit messages:

File	Commit Message	Time
..		
main.cu	fixed errors and typos	a minute ago
makefile	adding standalone reduction cuda example	10 minutes ago

*The .cu file contains both the partialSum reduction DEVICE kernel and the HOST code.
Note: the HOST code includes event based timing of the kernel execution.*

Hands On #2: CUDA Mandelbrot Area

Converting the Mandelbrot example from Tim Mattson's talk to CUDA.

#1. Retrieve the files:

```
git clone https://github.com/tcew/ATPESC18
```

#2. Complete the skeleton code

```
ATPESC18/handsOn/mandelbrot/mandelbrot.cu
```

[Do things labelled TASK, don't touch things marked FREEBIE]

#3. Hints:

To compile (on cooley.alcf.anl.gov):

```
nvcc -arch=sm_30 -o mandelbrot mandelbrot.cu -lm
```

To run (on a cooley compute node):

```
./mandelbrot
```

Useful CUDA keywords (google for details) :

thread rank: threadIdx.x, threadIdx.y, blockIdx.x, blockIdx.y, blockDim

keywords: __device__, __global__

[to turn off CUDA kernel optimization: `nvcc -Xptxas -O3 -arch sm_30 -o mandelbrot mandelbrot.cu -lm`]

Part 3: Interlude on CUDA performance

Dark Arts Indeed

Classic Definition of “Supercomputer”

This is a well known definition of a “supercomputer”

“A supercomputer is a device for turning compute-bound problems into I/O-bound problems.”

Ken Batchers*

... Another Cool Quote...

In much the same vain...

“Arithmetic is cheap, bandwidth is money, latency is physics.”

Mark Hoemmen*

NVIDIA can be viewed as a company that sells expensive GDDR memory.

**Student of Jim Demmel: thesis web link*

CUDA: memory options

The different memory spaces on the GPU have different characteristics

Memory	Location	Latency	Cached	Access	Scope	Lifetime
Register	On-chip	1	N/A	Read/write	One thread	Thread
Local	Off-chip	1000	No	Read/write	One thread	Thread
Shared	On-chip	2	N/A	Read/write	All threads in a block	Block
Global	Off-chip	1000	Yes*	Read/write	All threads & host	Application
Constant	Off-chip	1-1000	Yes	Read	All threads & host	Application
Texture	Off-chip	1000	Yes	Read	All threads in a block	Application
Read-only Cache	On-chip	Low	Yes	Read/write	?	?

CUDA: limitations

Recall the table showing that CUDA compute capabilities have evolved over time

Technical specifications	Compute capability (version)							
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	5.0
Maximum dimensionality of grid of thread blocks	2				3			
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535					2 ³¹⁻¹		
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512				1024			
Maximum z-dimension of a block	64							
Maximum number of threads per block	512				1024			
Warp size	32							
Maximum number of resident blocks per multiprocessor	8					16		32
Maximum number of resident warps per multiprocessor	24	32		48	64			
Maximum number of resident threads per multiprocessor	768	1024		1536	2048			
Number of 32-bit registers per multiprocessor	8 K	16 K		32 K	64 K			
Maximum number of 32-bit registers per thread	128				63	255		
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB
Number of shared memory banks	16				32			

There are several interesting tidbits here.

Table credit: CUDA wikipedia page (<http://en.wikipedia.org/wiki/CUDA>)

CUDA: occupancy calculator

The amount of register space is highly constrained:
kernels with high register count will have low occupancy

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit <http://developer.nvidia.com/cuda>

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 3.5 (Help)

1.b) Select Shared Memory Size Config (bytes): 49152

2.) Enter your resource usage:

Threads Per Block: 256 (Help)

Registers Per Thread: 32

Shared Memory Per Block (bytes): 4096

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor: 2048 (Help)

Active Warps per Multiprocessor: 64

Active Thread Blocks per Multiprocessor: 8

Occupancy of each Multiprocessor: 100%

Physical Limits for GPU Compute Capability: 3.5

Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	4096	49152	12

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	8	8	64
Limited by Shared Memory per Multiprocessor	12		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

Impact of Varying Block Size

Impact of Varying Register Count Per Thread

CUDA Occupancy Calculator: (download) spreadsheet tallies up register count, shared memory count, and thread count per thread-block to estimate how many thread-blocks can be resident.

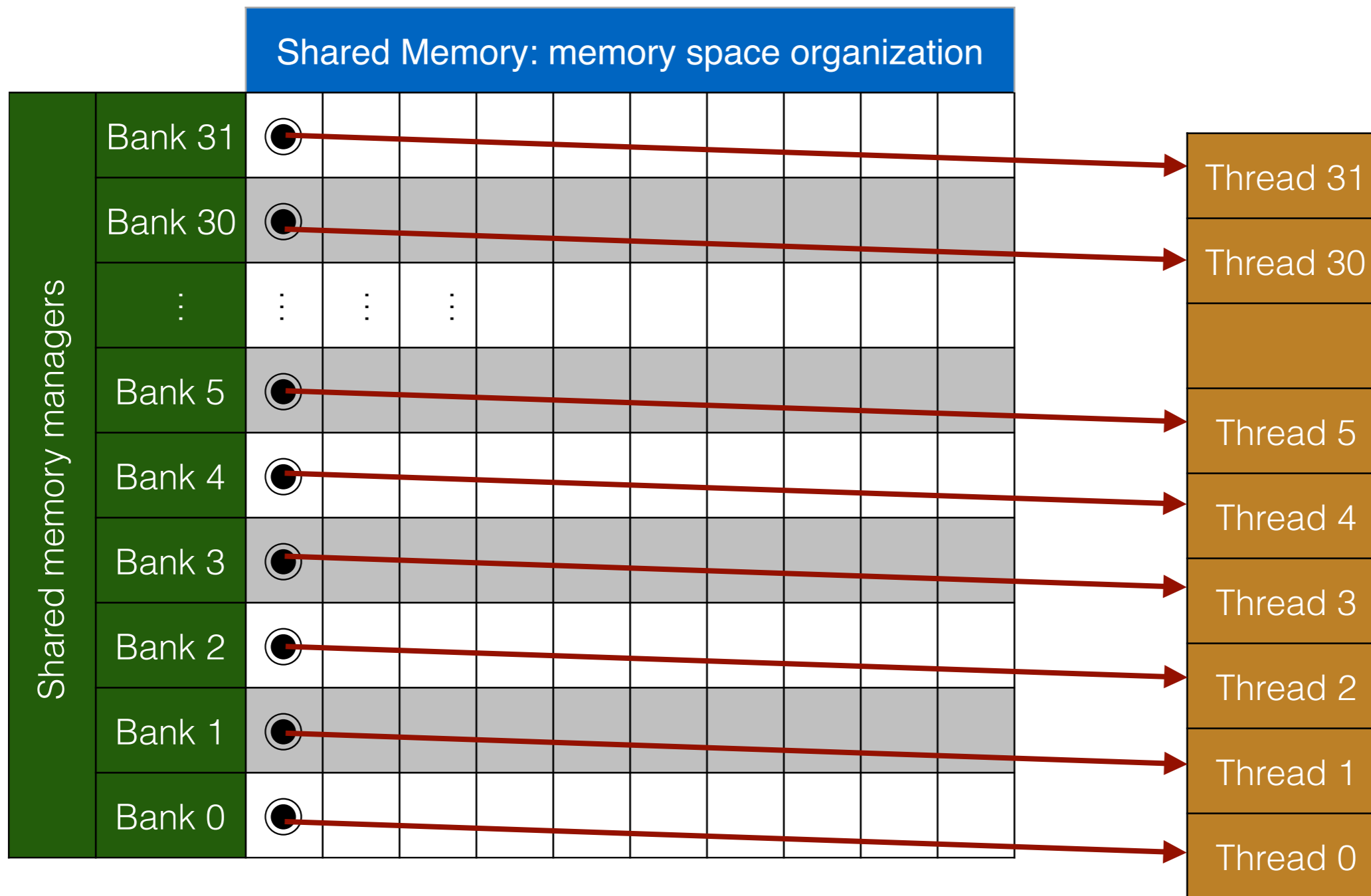
CUDA: shared memory banks

Shared memory is organized as interwoven “memory banks” with separate managers.
A shared memory array spans up to 32 independent memory banks.

		Shared Memory: memory space organization									
Shared memory managers	Bank 31	31	63	95							
	Bank 30	30	62	94							
	⋮	⋮	⋮	⋮							
	Bank 5	5	37	69							
	Bank 4	4	36	68							
	Bank 3	3	35	67							
	Bank 2	2	34	66							
	Bank 1	1	33	65	...						
	Bank 0	0	32	64	128						

CUDA: shared memory banks

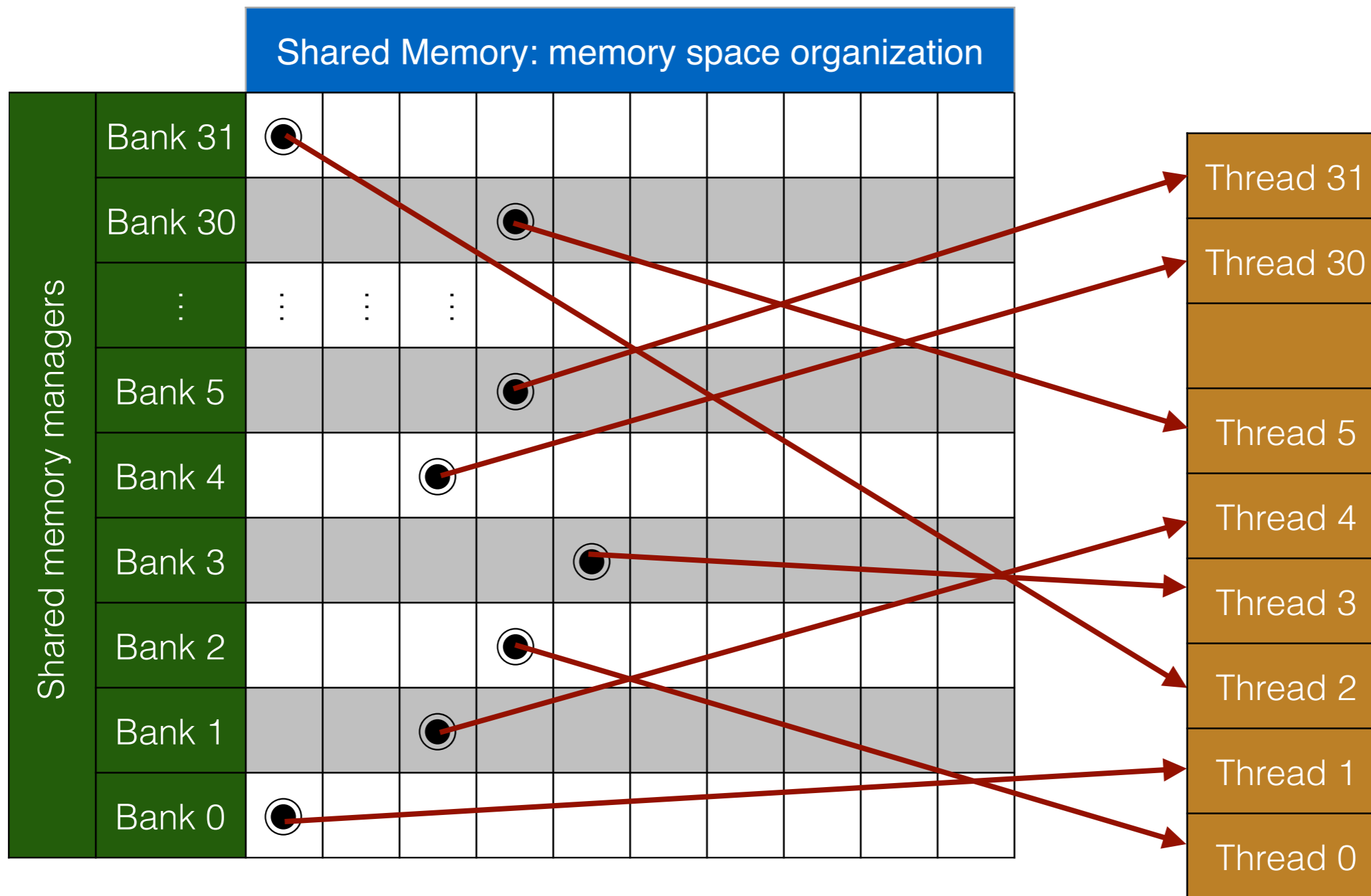
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.



OK: all threads in the SIMD group access different shared memory banks

CUDA: shared memory banks

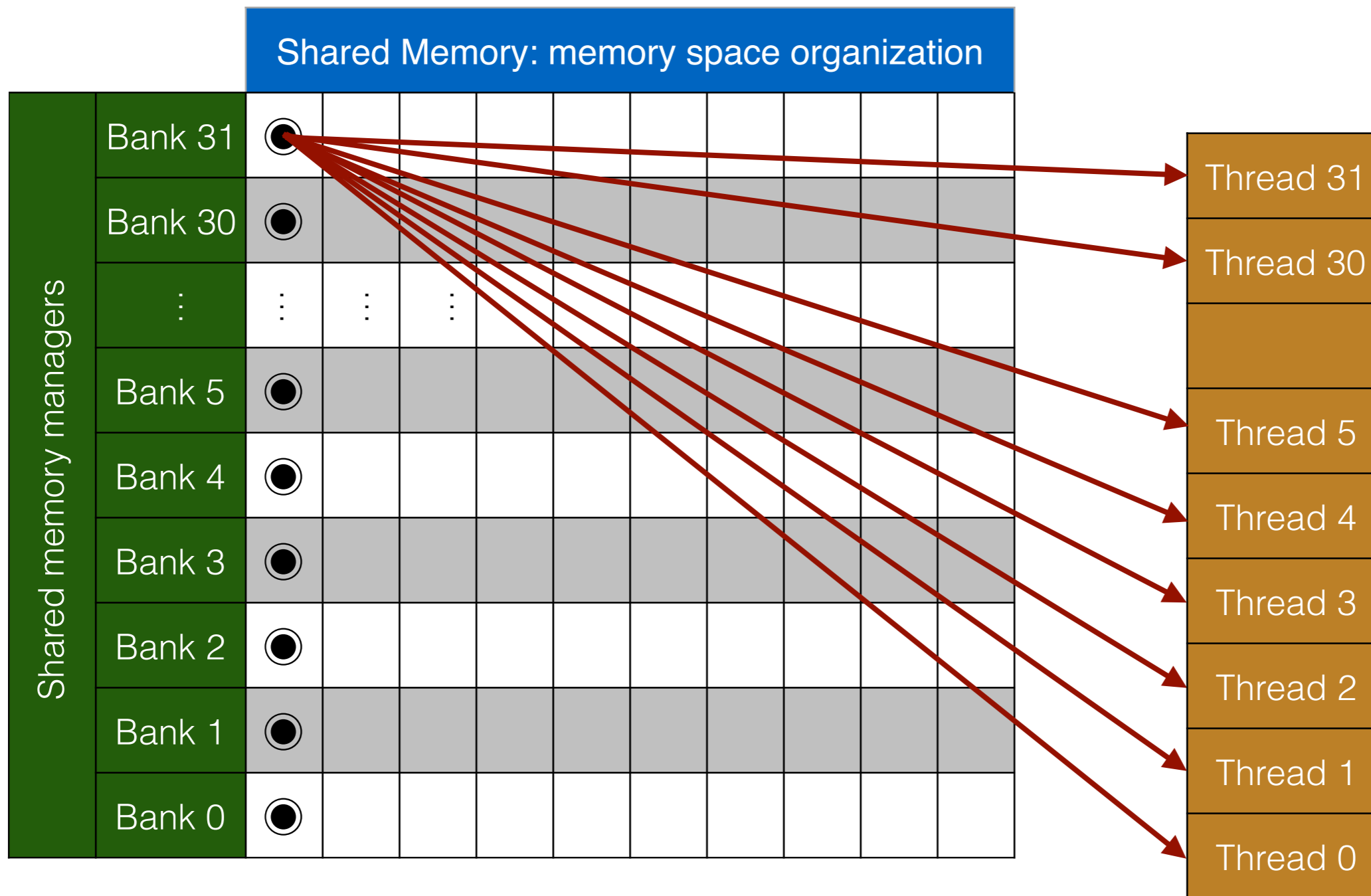
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.



OK: all threads in the SIMD group access different shared memory banks

CUDA: shared memory broadcast

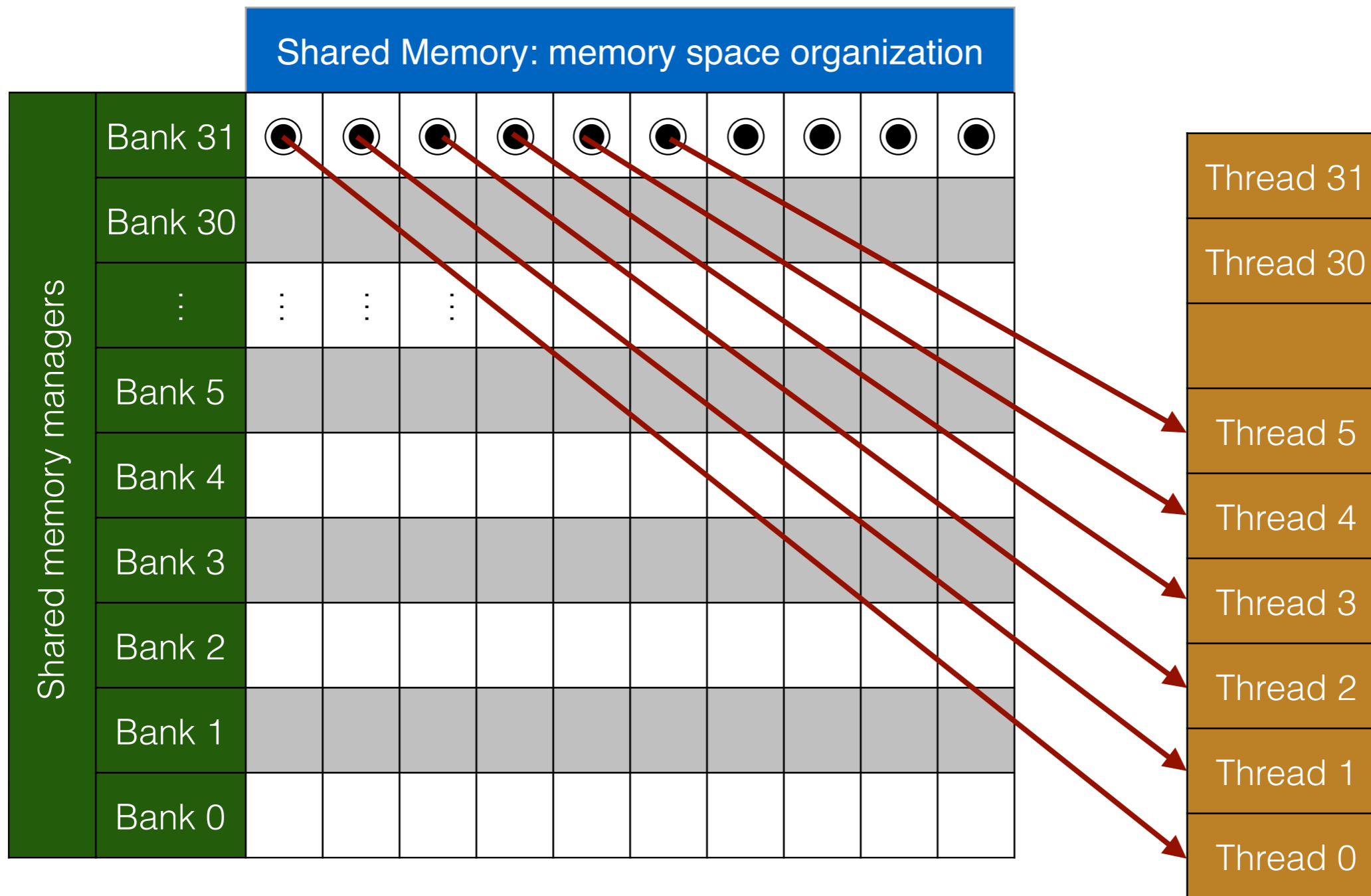
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.



OK: all threads in the SIMD group access the same entry results in an efficient broadcast.

CUDA: shared memory broadcast

To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.



BAD: all threads in the SIMD group access the same bank resulting in serialization.

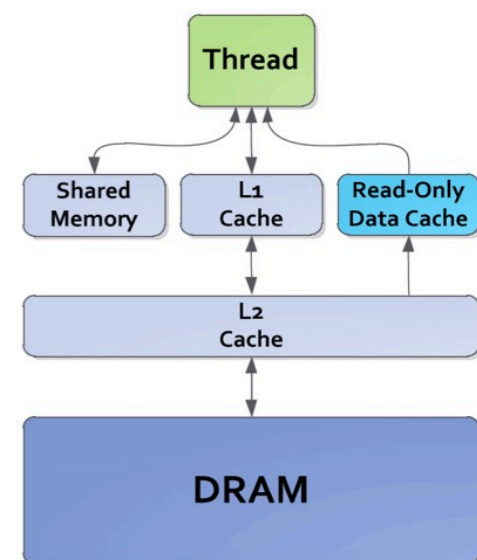
CUDA: accessing device memory

High end NVIDIA GPUs either have 256 or 384 bit wide memory bus to device memory



1. GPU has a “coalescer” that collects DRAM memory requests.
2. The coalescer efficiently streams contiguous, aligned blocks of memory by avoiding repeated address setup.
3. The GPU bus to DRAM consists of 6x 64 bit busses.
4. Each bus has an independent memory controller.

Kepler Memory Hierarchy

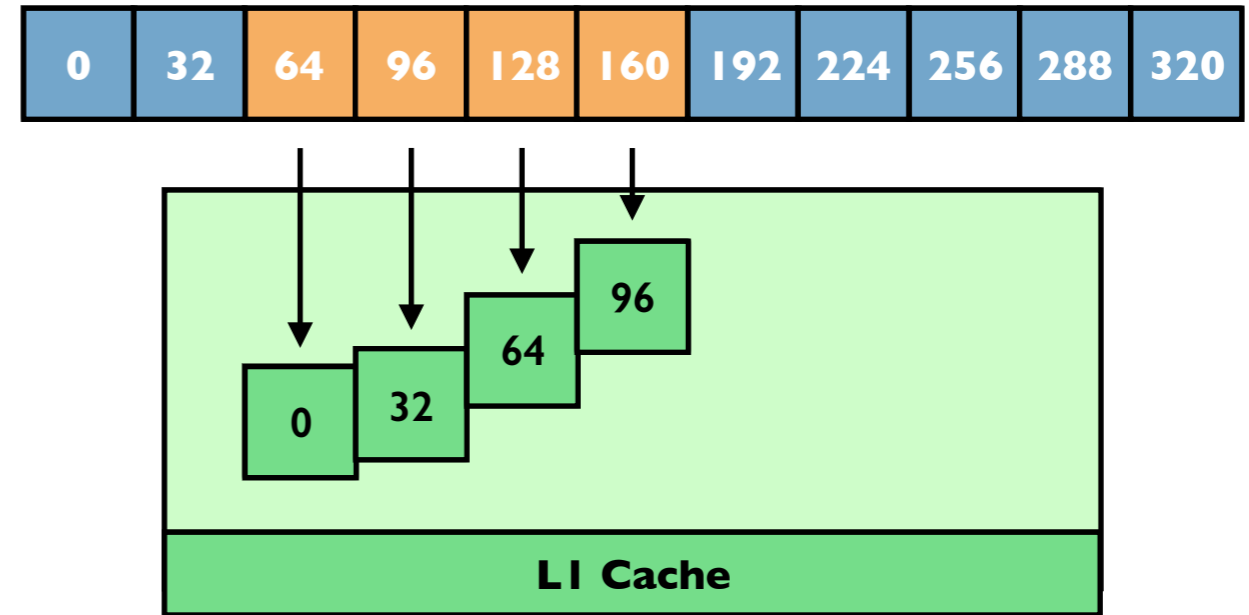


*Rule of thumb: avoid non unitary stride DEVICE (DRAM) array access.
Useful slides , these , and image credit: [link](#)*

CPU Optimization Techniques

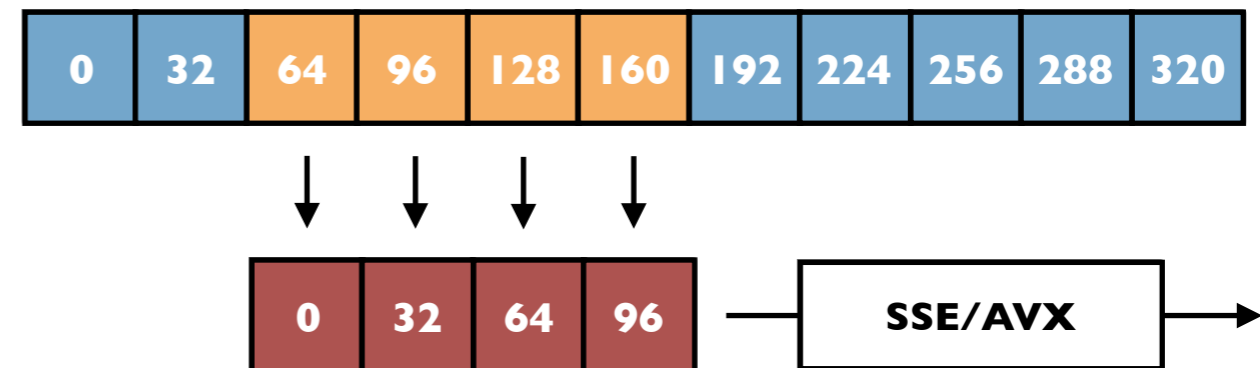
Cache

- Data loaded into cache from aligned contiguous blocks (cache lines)



Vectorization

- Use large registers instructions to perform operations in parallel.
- Also uses continuous load instructions to vectorize efficiently.

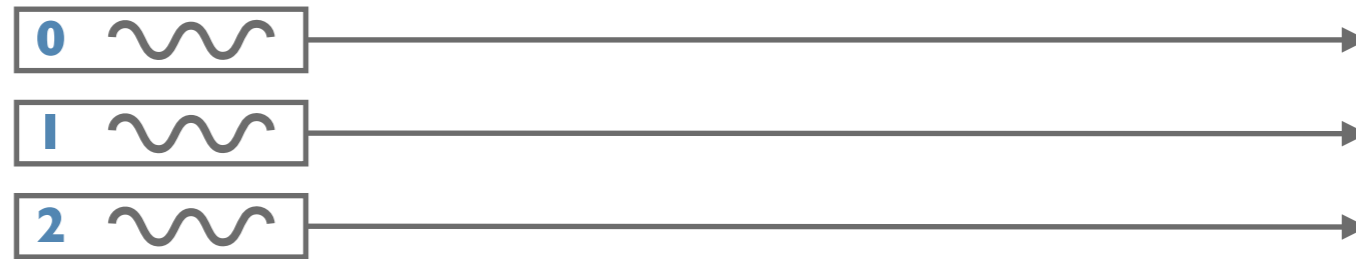


Continuous memory accesses are used for both, cache storage and vectorization

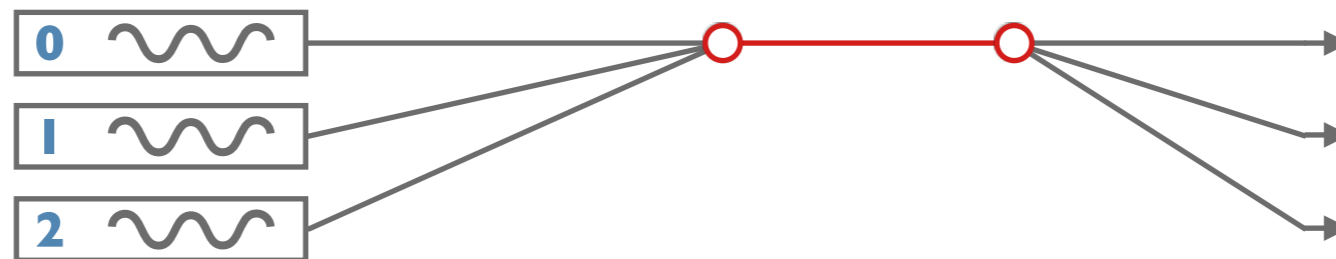
CPU Optimization Techniques

Multithreading

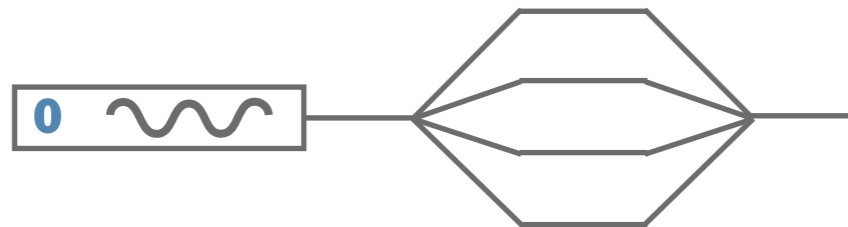
- Threads capable of fully parallelizing **generic** instructions (ignoring bandwidth).



- Perfect scaling ... **without** barriers, joins, or other types of thread-dependencies.



- SIMD Lanes

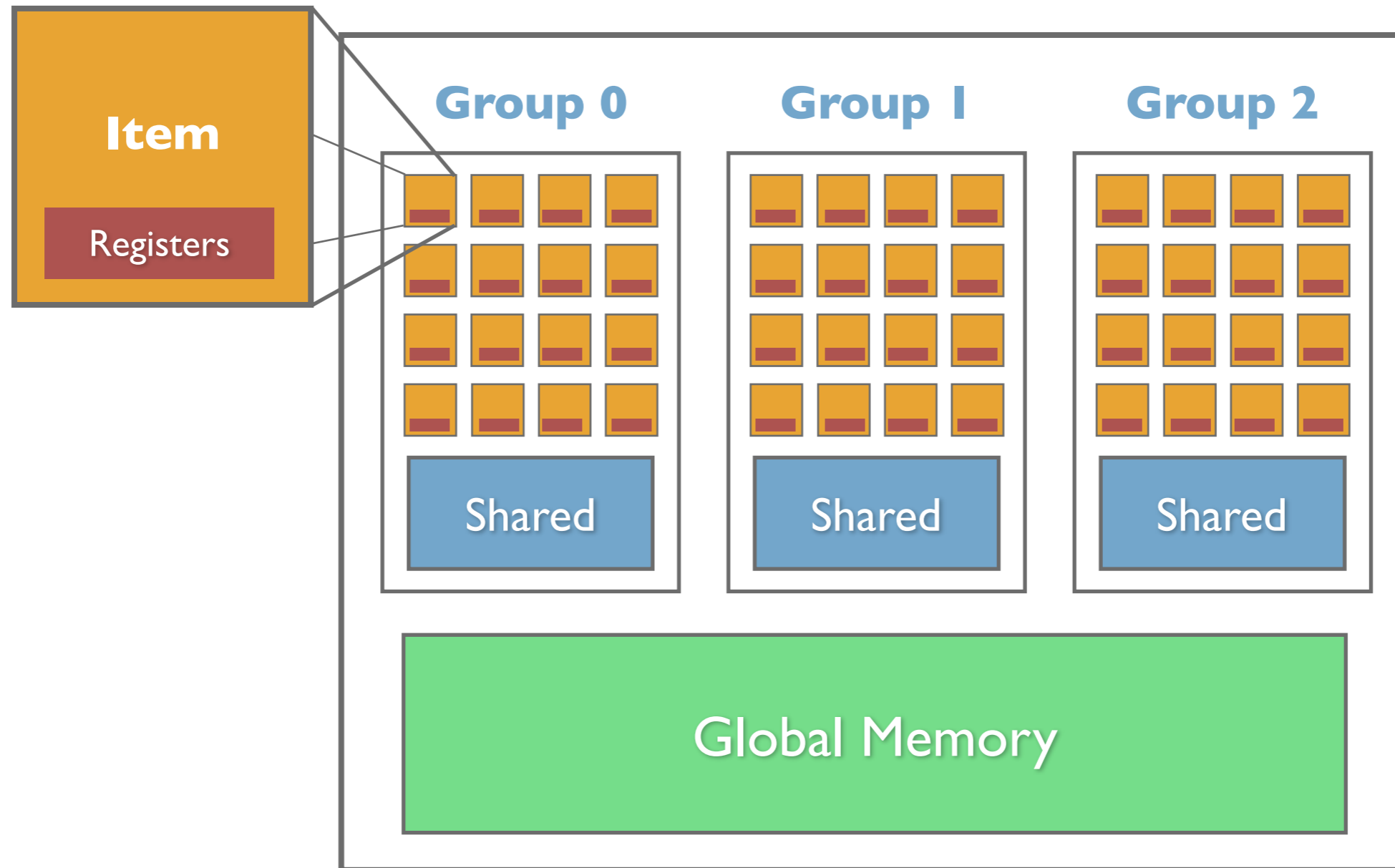


Executing parallel instructions using multithreading

GPU Optimization Techniques

GPU Architecture

- Independent work-groups are launched.
- Work-groups contain groups of work-items, “parallel” threads.

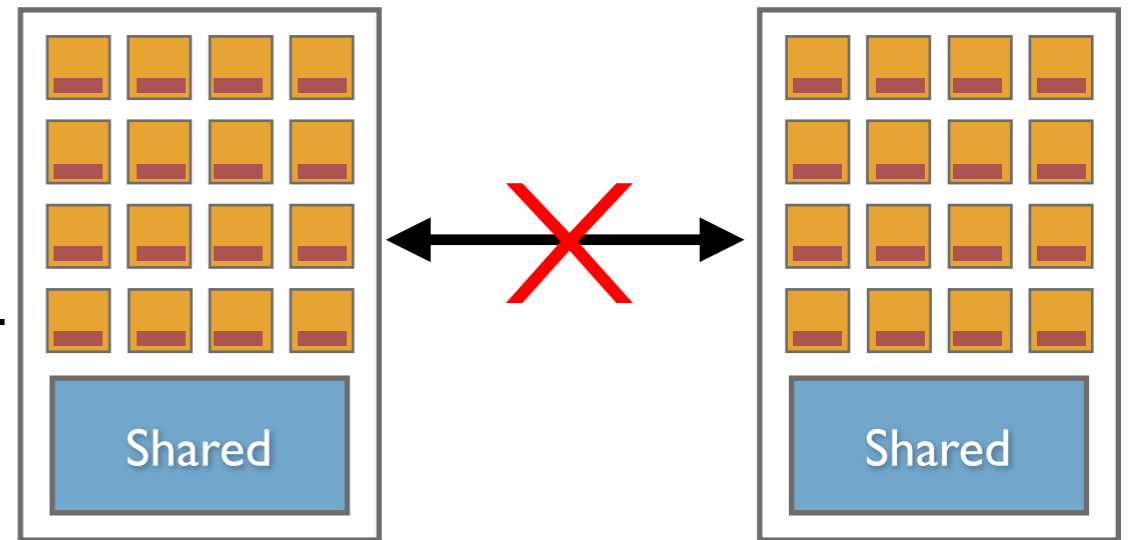


Kernel code describes the work-item operations

GPU Optimization Techniques

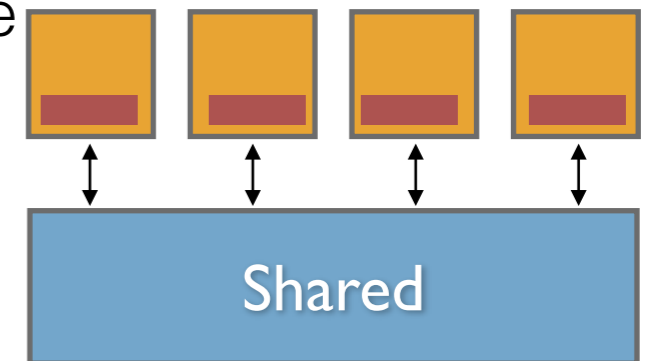
Work-groups

- Groups of work-items.
- No communication between work-groups.
- Designed for independent group parallelism.
- Avoid inter-block synchronization (deadlocks).
- Avoid data race dependencies between blocks.



Work-items

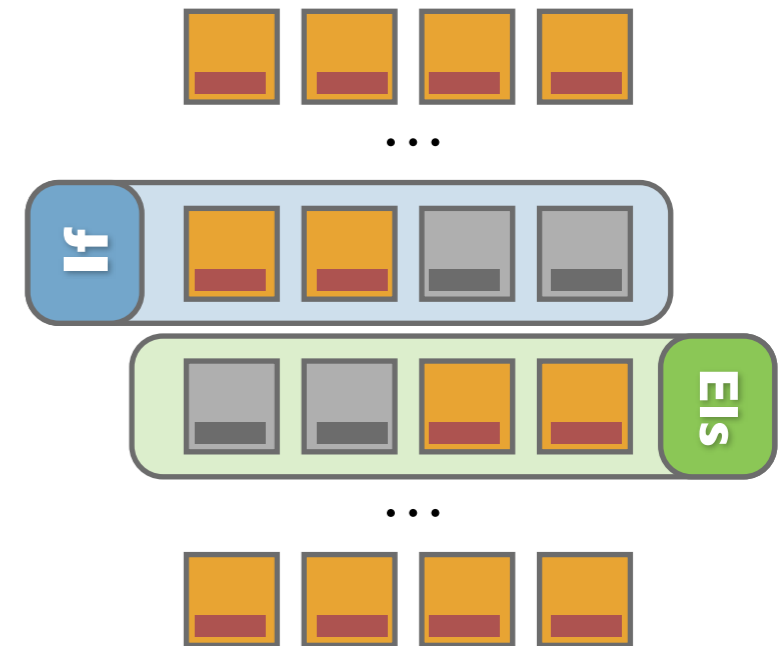
- Work-items are executed in parallel, able to barrier and share data using shared memory (& CUDA's shuffle).
- Avoid data race dependencies between work-items.



GPU Optimization Techniques

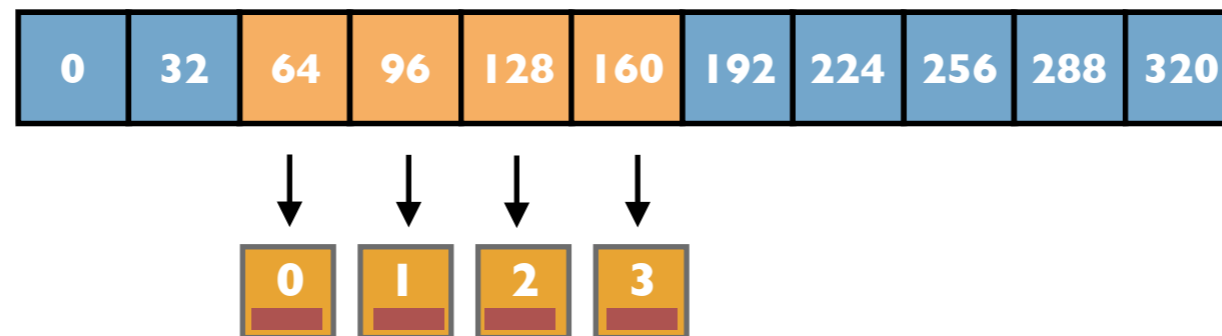
Parallel Work-item Execution

- Work-items are launched in subsets of 32 or 64.
- Each set of work-items execute same instructions.
- No parallel branching (**in the subset**).



Data Transfer

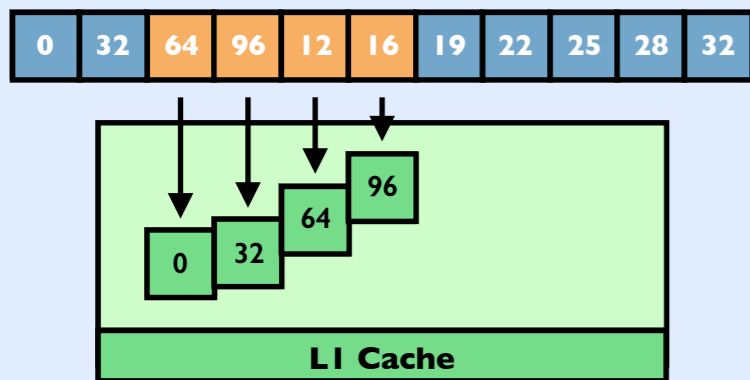
- Low individual bandwidth and high latency.
- Coalesced memory access on contiguous and aligned work-items.



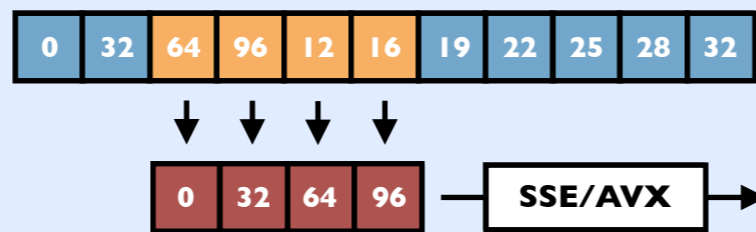
CPU & GPU Similarities

CPU Optimizations

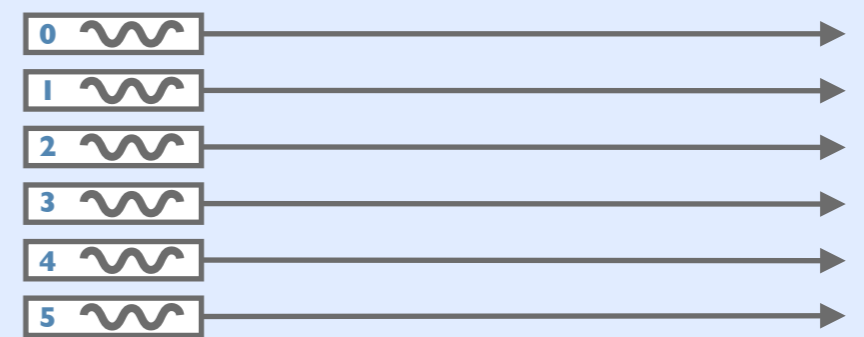
Cache



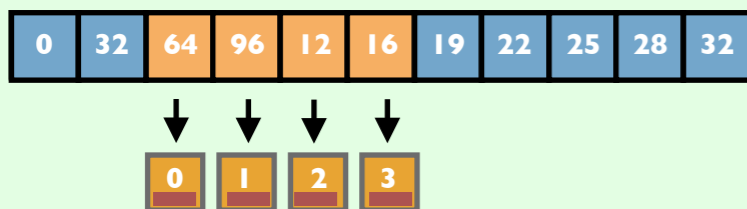
Vectorization



Thread Independence



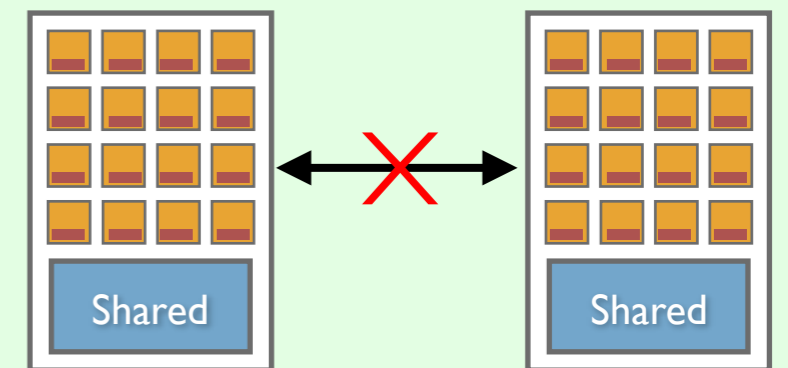
Coalescing



No Branching



Work-group Independence



GPU Optimizations

Exposing vectorization / SIMD parallelism are vital in both architectures

Part 4: Portable programming models

RIP: Blinky 01/15/14-08/03/15

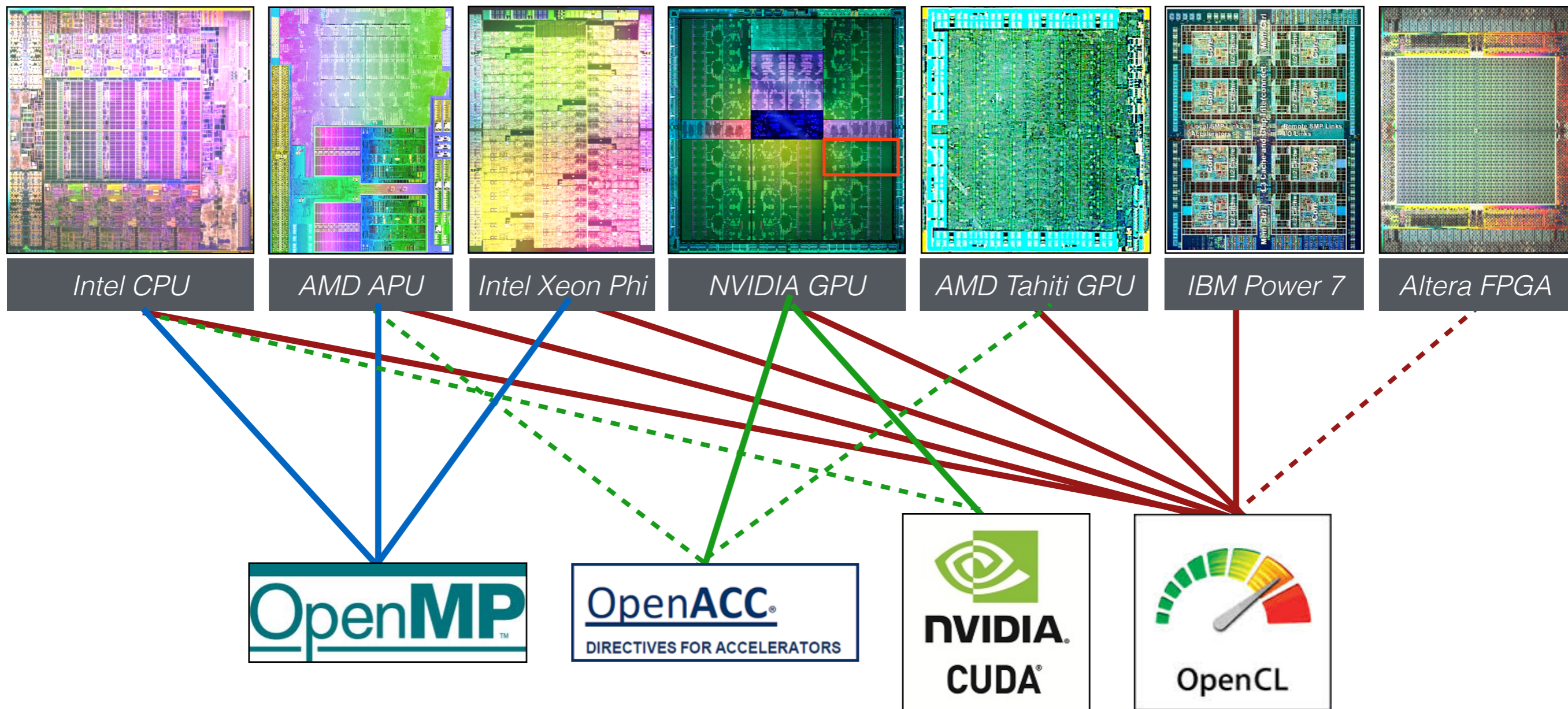


Blinky was a MacBook Pro with discrete NVIDIA GPU and OpenMP, OpenCL, CUDA ...

Latest MBP comes with AMD GPU and/or Intel Iris GPU, no CUDA, and default clang compilers omit OpenMP 96

Many-core: fragmentation

Zoo of competing architectures and programming models (with vendor bias)



Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming

Part 4a:
Open Compute
Language (OpenCL)

OpenCL: standards committee

OpenCL - The open stand: x
www.khronos.org/opencl/

English 中文 | 日本語 | 한국어 Login Members | Adopters | Implementers Search Khronos.org

KHRONOS
GROUP
CONNECTING SOFTWARE TO SILICON

Developers ▾ Conformance ▾ Membership ▾ News ▾ Events ▾ Forums ▾

OpenCL	OpenGL	OpenGL ES	WebGL	WebCL	COLLADA	glTF	EGL
OpenSL ES	OpenMAX	SPIR	SYCL	StreamInput	OpenVX	Camera	Other

Home ▸ OpenCL

OpenCL SPIR SYCL Overview PDF Overview Forums Adopters Resources

The open standard for parallel programming of heterogeneous systems

OpenCL™ is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL (Open Computing Language) greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and medical software.

OpenCL 2.0

Share It
212
Like
96
Tweet
102
g+1
21
in s

Quick-reference-card for OpenCL 2.0: ([link](#))

The Khronos Group administers the OpenCL standard.

OpenCL: standard for multicore

OpenCL allows us to write cross platform code
(customization need for best performance)

KHRONOS
GROUP



OpenCL



The Khronos Group administers the OpenCL standard.

OpenCL: who ?

OpenCL Working Group

- **Diverse industry participation**
 - Processor vendors, system OEMs, middleware vendors, application developers
- **Many industry-leading experts involved in OpenCL's design**
 - A healthy diversity of industry perspectives
- **Apple made initial proposal and is very active in the working group**
 - Serving as specification editor

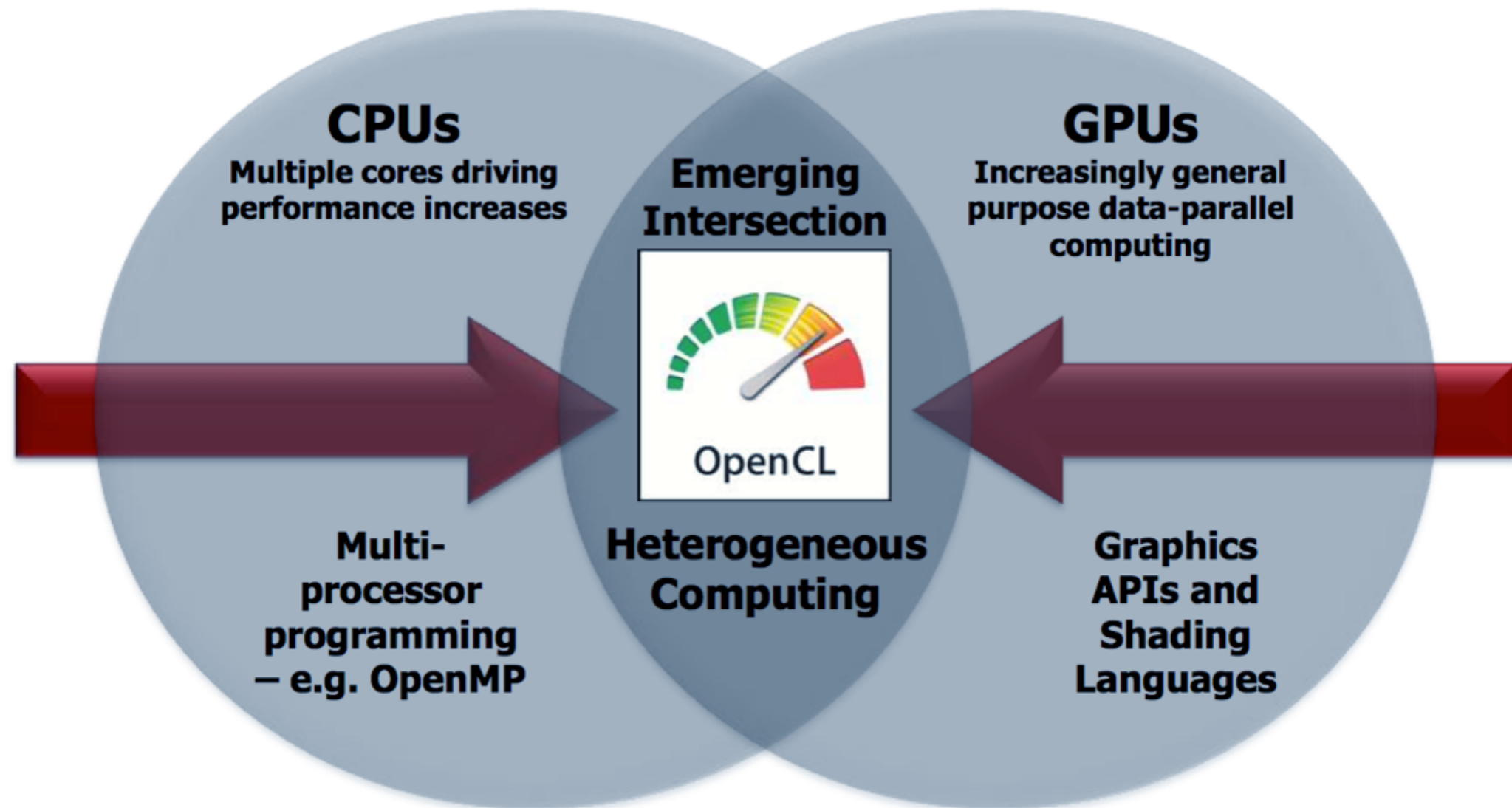


KHRONOS GROUP

*The OpenCL standard changes relatively slowly over time compared to CUDA.
Credit: Khronos Group*

OpenCL: why ?

Processor Parallelism



OpenCL is a programming framework for heterogeneous compute resources

© Copyright Khronos Group, 2010 - Page 3

*Emphasis on heterogeneous computing.
Credit: Khronos Group*

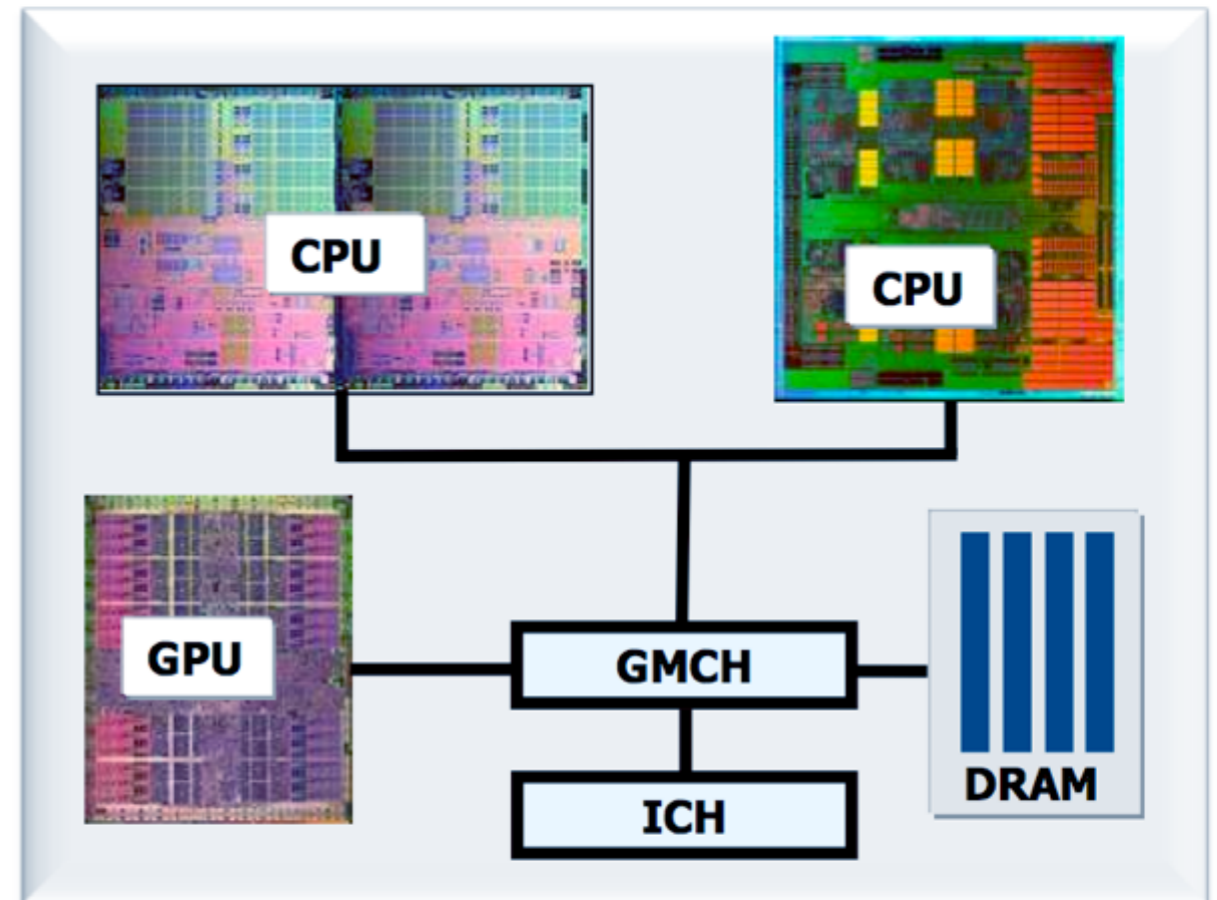
OpenCL: why ?

It's a Heterogeneous World

- **A modern platform Includes:**

- One or more CPUs
- One or more GPUs
- DSP processors
- ... other?

OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

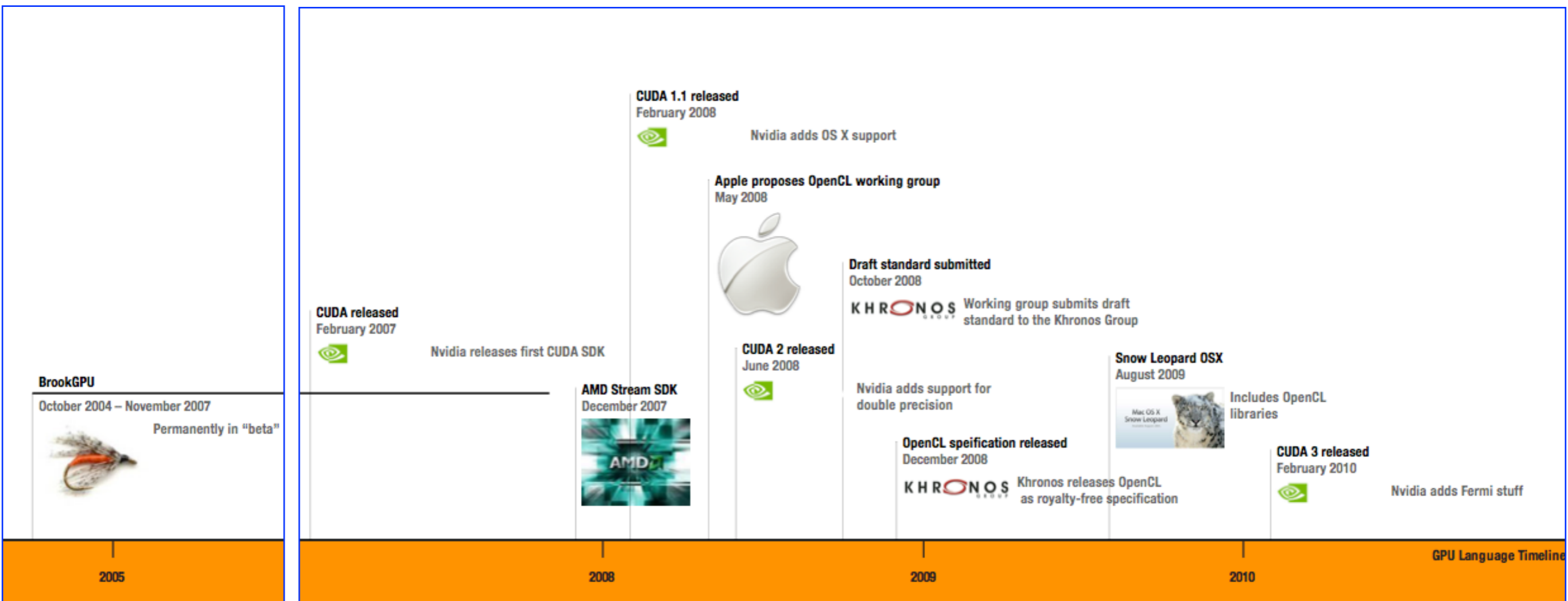


GMCH = graphics memory control hub

ICH = Input/output control hub

OpenCL: when ?

CUDA and OpenCL are competing standards for GPGPU programming



GPGPU "quiet time"

Only a few hardy souls tried GPU computing before CUDA was released.

OpenCL: terminology ?

OpenCL is **very** closely related to CUDA

CUDA	OpenCL
Kernel	Kernel
Host program	Host program
Thread	Work item
Thread block	Work group
Grid	NDRange (index space)

The rapid development of OpenCL helps explain the similarities

OpenCL: thread indexing

OpenCL is **very** closely related to CUDA

CUDA		OpenCL	
Local indices:		Local indices:	
threadIdx.x	threadIdx.y	get_local_id(0)	get_local_id(1)
Global indices:		Global indices:	
$\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$	$\text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$	get_global_id(0)	get_global_id(1)

The rapid development of OpenCL helps explain the similarities

OpenCL: thread array dimensions

OpenCL is **very** closely related to CUDA

CUDA	OpenCL
gridDim.x	get_num_groups(0)
blockIdx.x	get_group_id(0)
blockDim.x	get_local_size(0)
gridDim.x*blockDim.x	get_global_size(0)

The rapid development of OpenCL helps explain the similarities

OpenCL: kernel language qualifiers

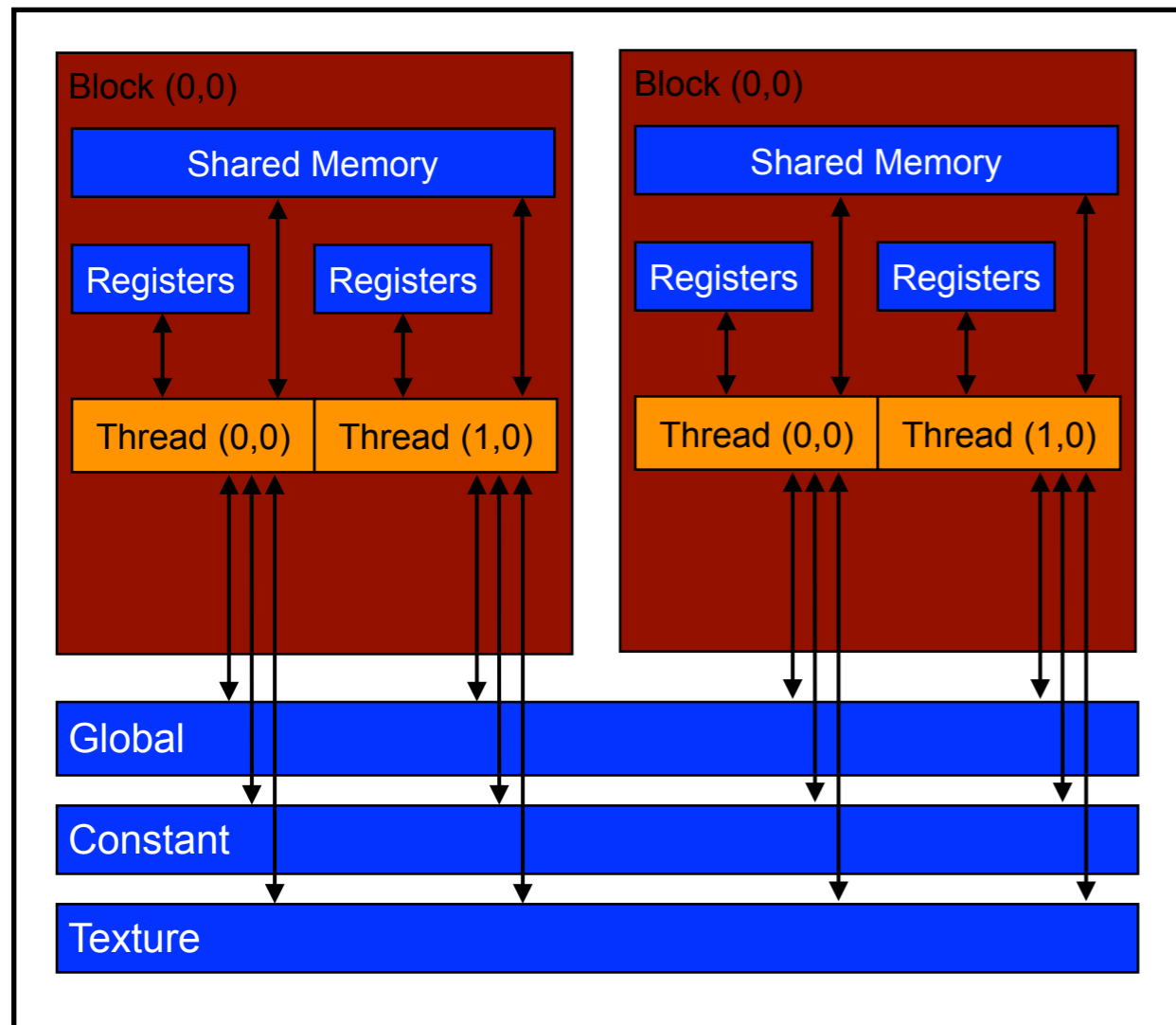
OpenCL is **very** closely related to CUDA

CUDA	OpenCL
<code>__global__</code> function	<code>__kernel</code> function
<code>__device__</code> function	function
<code>__constant__</code> variable	<code>__constant</code> variable
<code>__device__</code> variable	<code>__global</code> variable
<code>__shared__</code> variable	<code>__local</code> variable

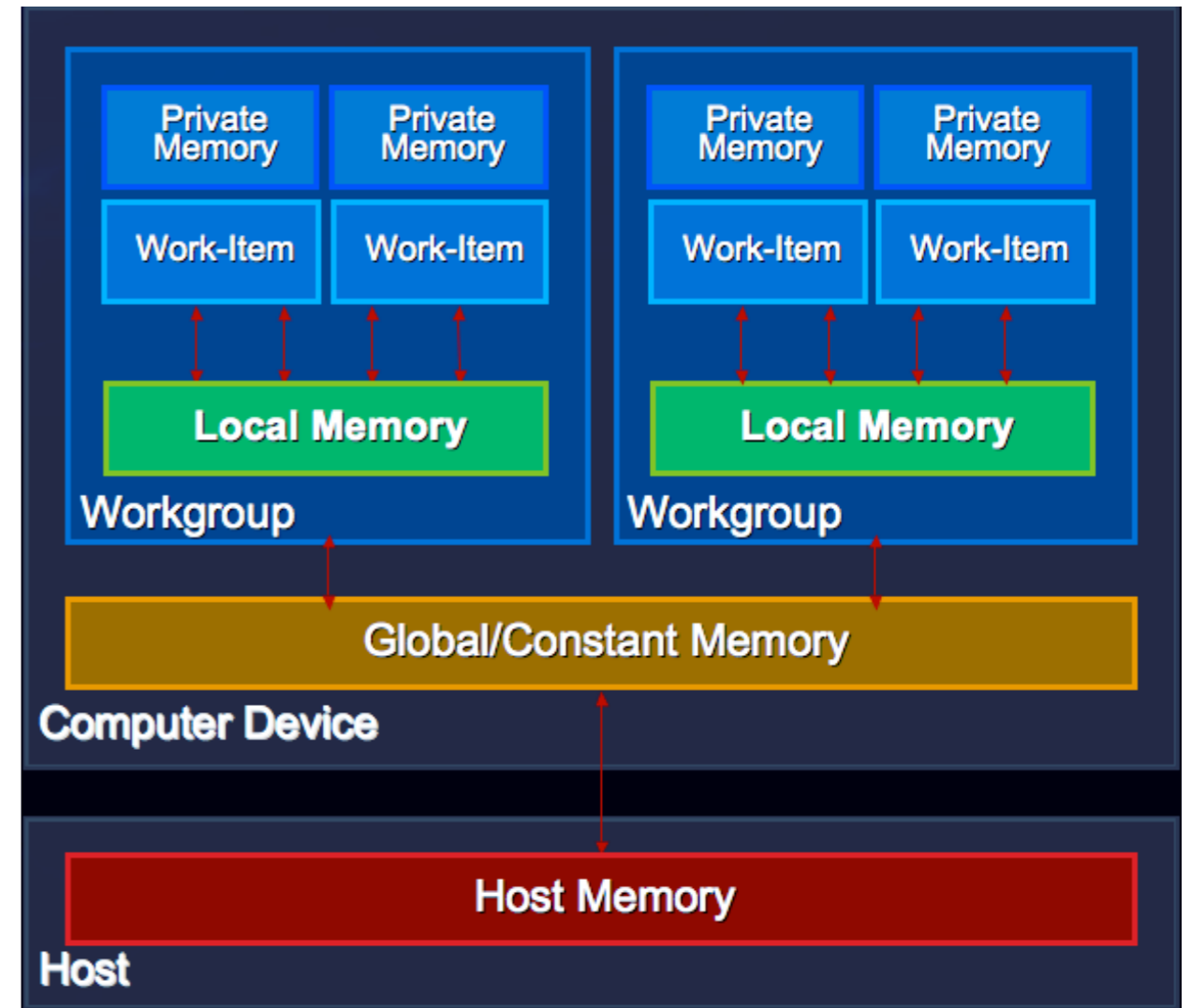
The rapid development of OpenCL helps explain the similarities

OpenCL: memory model

Again, the memory model for CUDA and OpenCL are very similar



CUDA



OpenCL

Image system not shown
[AMD OpenCL slides](#)

The rapid development of OpenCL helps explain the similarities

OpenCL: setting up a DEVICE

OpenCL is very flexible, allowing simultaneous heterogeneous computing with possibly multiple implementations, command queues, & devices in one system [CPU+GPUs]

To set up a device:

1. Choose platform (implementation of OpenCL) from list of platforms:
 - `clGetPlatformIDs`
2. Choose device on that platform (for instance a specific CPU or GPU):
 - `clGetDeviceIDs`
3. Create a context on the device (manager for tasks):
 - `clCreateContext`
4. Create command queue on a context on the chosen device:
 - `clCreateCommandQueue`

With flexibility can come complexity.

OpenCL: HOST code API headers

The include files ...

CUDA

```
#include <cuda.h>
```

OpenCL

```
#ifdef __APPLE__  
#include <OpenCL/opencl.h>  
#else  
#include <CL/cl.h>  
#endif
```

OpenCL: setting up a platform

For flexibility we first have to choose the OpenCL “platform”

```
#include <cuda.h>

int main()
{

// nothing special to do (really only one CUDA platform)
```

```
...
cl_platform_id    platforms[100];
cl_uint          platforms_n;

/* get list of platforms(platform == OpenCL implementation) */
clGetPlatformIDs(100, platforms, &platforms_n);
...
```

*Any given system may have multiple OpenCL platforms from different vendors installed.
We will choose one of the returned platform IDs.*

OpenCL: choosing a device

Next we choose a device supported by the platform.

```
...  
  
int dev = 0;  
cudaSetDevice(dev);  
  
...
```

```
...  
cl_device_id    devices[100];  
cl_uint         ndevices;  
  
clGetDeviceIDs(platforms[plat], CL_DEVICE_TYPE_ALL, 100, devices, &ndevices);  
  
if(dev >= ndevices){ printf("invalid device\n"); exit(0); }  
  
// choose user specified device  
cl_device_id device = devices[dev];  
  
...
```

Each OpenCL platform can interact with one or more compute devices.

OpenCL: setting up a context

Next we choose a context (manager) for the chosen device.

```
...  
// nada  
...
```

```
cl_context context;  
  
// make compute context on device (pfn_notify is an error callback function)  
context = clCreateContext((cl_context_properties *)NULL, 1, &device,  
                        &pfn_notify, (void*)NULL, &err);
```

OpenCL: setting up a common queue

Next we choose a context (manager) for the chosen device.

```
..  
// not necessary although you may wish to use cudaStreamCreate  
..
```

```
// make compute context on device (pfn_notify is an error callback function)  
cl_command_queue queue =  
    clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
```

OpenCL: compiling a DEVICE kernel

Since the platform+device+context is chosen at runtime
it is customary to build compute kernels at runtime.

To set up a kernel on a DEVICE:

1. Represent kernel source code as a C character array:
 - `clCreateProgramWithSource`
2. Create a “program” from the source code:
 - `clBuildProgram`
3. Compile and build the “program”:
 - `clGetProgramBuildInfo`
4. Check for compilation errors:
 - `clGetProgramBuildInfo`
5. Build executable kernel:
 - `clCreateKernel`

```
const char *source =  
    "__kernel void foo(int N, __global float *x){"  
    "  
    "    int id = get_global_id(0);  
    "  
    "    if(id<N)  
    "        x[id] = id;  
    "  
    "};";
```

OpenCL: building a kernel

We now need to build the kernel [some steps skipped for brevity]

```
..  
// not necessary  
// nvcc compiles the kernel code when you compile the executable  
..
```

```
/* create program from source */  
cl_program program = clCreateProgramWithSource(context, 1,  
                                              (const char **) & source, (size_t*) NULL, &err);  
  
/* compile and build program */  
const char *allFlags = " ";  
err = clBuildProgram(program, 1, &device, allFlags,  
                   (void (*)(cl_program, void*)) NULL, NULL);  
  
/* omitted error checking */  
..  
/* create runnable kernel */  
cl_kernel kernel = clCreateKernel(program, functionName, &err);
```

And we have to do that for each kernel.

OpenCL: are we there yet ?

Unbelievably no.

To execute the kernel:

1. Just like CUDA we need to allocate storage on the DEVICE:
 - `clCreateBuffer`
2. We need to add the input arguments one at a time to the kernel:
 - `clSetKernelArg`
3. Specify the local work-group size and global thread array sizes.
4. Queue the kernel
 - `clEnqueueNDRangeKernel`
5. Wait for the kernel to finish:
 - `clFinish`

Nearly there ?

OpenCL: thanks for the memory

We next allocate array space on the DEVICE:

```
int N = 100; /* vector size */

/* size of array */
size_t sz = N*sizeof(float);

float *d_a; // CUDA uses pointer for array handles

cudaMalloc((void**) &d_a, N*sizeof(float));
```

```
int N = 100; /* vector size */

/* size of array */
size_t sz = N*sizeof(float);

/* create device buffer and copy from host buffer */
cl_mem c_x = clCreateBuffer(context,
                           CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sz, h_x, &err);
```

In this case we have provided CL with a host pointer and clCreateBuffer copies from h_x to c_x.

OpenCL: kernel good to go ?

Not quite: we now need to specify each kernel argument one by one.

```
...  
dim3 dimBlock(256,1,1);           // 512 threads per thread-block  
dim3 dimGrid((N+255)/256, 1, 1); // Enough thread-blocks to cover N  
  
// Queue kernel on DEVICE  
simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);  
...
```

```
/* now set kernel arguments one by one */  
clSetKernelArg(kernel, 0, sizeof(int), &N);  
clSetKernelArg(kernel, 1, sizeof(cl_mem), &c_x);  
  
/* set thread array */  
int dim = 1;  
size_t local[3] = {256,1,1};  
size_t global[3] = {256*((N+255-1)/256),1,1};  
  
/* queue up kernel */  
clEnqueueNDRangeKernel(queue, kernel, dim, 0, global, local, 0,  
                        (cl_event*)NULL, NULL);
```

*Note: CUDA uses block sizes + number of blocks.
OpenCL uses block sizes and global number of threads.*

OpenCL: simple kernel example

The kernel programming languages are similar:

CUDA

```
__global__ void simpleKernel(int N,  
                             float *a)  
{  
  
    /* get thread coordinates */  
    int i = threadIdx.x +  
           blockIdx.x*blockDim.x;  
  
    /* do simple task */  
    if(i<N)  
        a[i] = i;  
}
```

OpenCL

```
__kernel void simpleKernel(int N,  
                            __global float *a)  
{  
  
    /* get thread coordinates */  
    int i = get_global_id(0);  
  
    /* do simple task */  
    if(i<N)  
        a[i] = i;  
}
```

Some minor differences in syntax & identifiers

OpenCL: summary

OpenCL seems to be a panacea: it works on everything...

- OpenCL has a bit of a bad reputation:
 - CUDA has a richer ecosystem of tools & libraries.
 - CUDA has more extensive documentation and tutorials.
 - Platform/device/context/queue complexity.
 - Competing vendor priorities.
 - The vendors offer differing levels of support.
 - OpenCL:Intel:CPU vectorization is flaky.
 - OpenCL:OS X:CPU limited work-items per work-group
 - OpenCL:NVIDIA:GPU trails CUDA in performance.
 - Rumors constantly circulate about EOL.
- On the other hand:
 - Runtime compilation adds several optimization opportunities without templating.
 - OpenCL is library based, so no special compilers are required.
 - Vendor independence is important.

OpenCL: comparing Jacobi kernels

Recalling the Poisson example: side by side comparison of serial v. CUDA v. OpenCL kernel

Iterate:

$$u_{ji}^{k+1} = \frac{1}{4} \left(-\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \dots, N$$

Serial kernel:

```
void jacobi(const int N,
           const double *rhs,
           const double *u,
           double *newu){
    for(int i=0;i<N;++i){
        for(int j=0;j<N;++j){

            // Get linear index into NxN
            // inner nodes of (N+2)x(N+2) grid
            const int id = (j + 1)*(N + 2) + (i + 1);

            newu[id] = 0.25f*(rhs[id]
                            + u[id - (N+2)]
                            + u[id + (N+2)]
                            + u[id - 1]
                            + u[id + 1]);
        }
    }
}
```

CUDA kernel:

```
__global__ void jacobi(const int N,
                      const double *rhs,
                      const double *u,
                      double *newu){

    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;

    // Check that this is a legal node
    if((i < N) && (j < N)){

        // Get linear index onto (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);

        newu[id] = 0.25f*(rhs[id]
                        + u[id - (N+2)]
                        + u[id + (N+2)]
                        + u[id - 1]
                        + u[id + 1]);
    }
}
```

OpenCL kernel:

```
?? void jacobi(    const int N,
                 ?? const double *rhs,
                 ?? const double *u,
                 ?? double *newu){

    // Get thread indices
    const int i = ??;
    const int j = ??;

    if((i < N) && (j < N)){

        ??;
    }
}
```

Note explicit loops in serial kernel and hidden loops in CUDA and OpenCL kernels.

OpenCL: comparing Jacobi kernels

Recalling the Poisson example: side by side comparison of serial v. CUDA v. OpenCL kernel

Iterate:

$$u_{ji}^{k+1} = \frac{1}{4} \left(-\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \dots, N$$

Serial kernel:

```
void jacobi(const int N,
           const double *rhs,
           const double *u,
           double *newu){
    for(int i=0;i<N;++i){
        for(int j=0;j<N;++j){
            // Get linear index into NxN
            // inner nodes of (N+2)x(N+2) grid
            const int id = (j + 1)*(N + 2) + (i + 1);
            newu[id] = 0.25f*(rhs[id]
                            + u[id - (N+2)]
                            + u[id + (N+2)]
                            + u[id - 1]
                            + u[id + 1]);
        }
    }
}
```

CUDA kernel:

```
__global__ void jacobi(const int N,
                      const double *rhs,
                      const double *u,
                      double *newu){
    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;
    // Check that this is a legal node
    if((i < N) && (j < N)){
        // Get linear index onto (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);
        newu[id] = 0.25f*(rhs[id]
                        + u[id - (N+2)]
                        + u[id + (N+2)]
                        + u[id - 1]
                        + u[id + 1]);
    }
}
```

OpenCL kernel:

```
__kernel void jacobi(const int N,
                    __global const double *rhs,
                    __global const double *u,
                    __global double *newu){
    // Get thread indices
    const int i = get_global_id(0);
    const int j = get_global_id(1);
    if((i < N) && (j < N)){
        // Get linear index into (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);
        newu[id] = 0.25f*(rhs[id]
                        + u[id - (N+2)]
                        + u[id + (N+2)]
                        + u[id - 1]
                        + u[id + 1]);
    }
}
```

Note explicit loops in serial kernel and hidden loops in CUDA and OpenCL kernels.

OpenCL: partial reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```
__global__ void partialReduceResidual(const int entries,
                                     double *u,
                                     double *newu,
                                     double *blocksum){

    __shared__ double s_blocksum[BDIM];

    const int id = blockIdx.x*blockDim.x + threadIdx.x;

    int alive = blockDim.x;
    int t = threadIdx.x;

    s_blocksum[threadIdx.x] = 0;

    if(id < entries){
        const double diff = u[id] - newu[id];
        s_blocksum[threadIdx.x] = diff*diff;
    }

    while(alive>1){

        __syncthreads(); // barrier (make sure s_blocksum is ready)

        alive /= 2;
        if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
    }

    if(t==0)
        blocksum[blockIdx.x] = s_blocksum[0];
}
```

OpenCL partial reduction kernel:

```
__kernel void partialReduce(    const int entries,
                               ?? const double *u,
                               ?? const double *newu,
                               ?? double *blocksum){

    __local double s_blocksum[BDIM];

    const int id = get_global_id();

    int alive = ??;
    int t = ??;

    s_blocksum[t] = 0;

    // load global data into local memory if in range
    if(id < entries){
        const double diff = u[id] - newu[id];
        s_blocksum[t] = diff*diff;
    }

    while(alive>1){

        barrier(CLK_LOCAL_MEMFENCE); // barrier (make sure s_blocksum is ready)

        alive /= 2;
        if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
    }

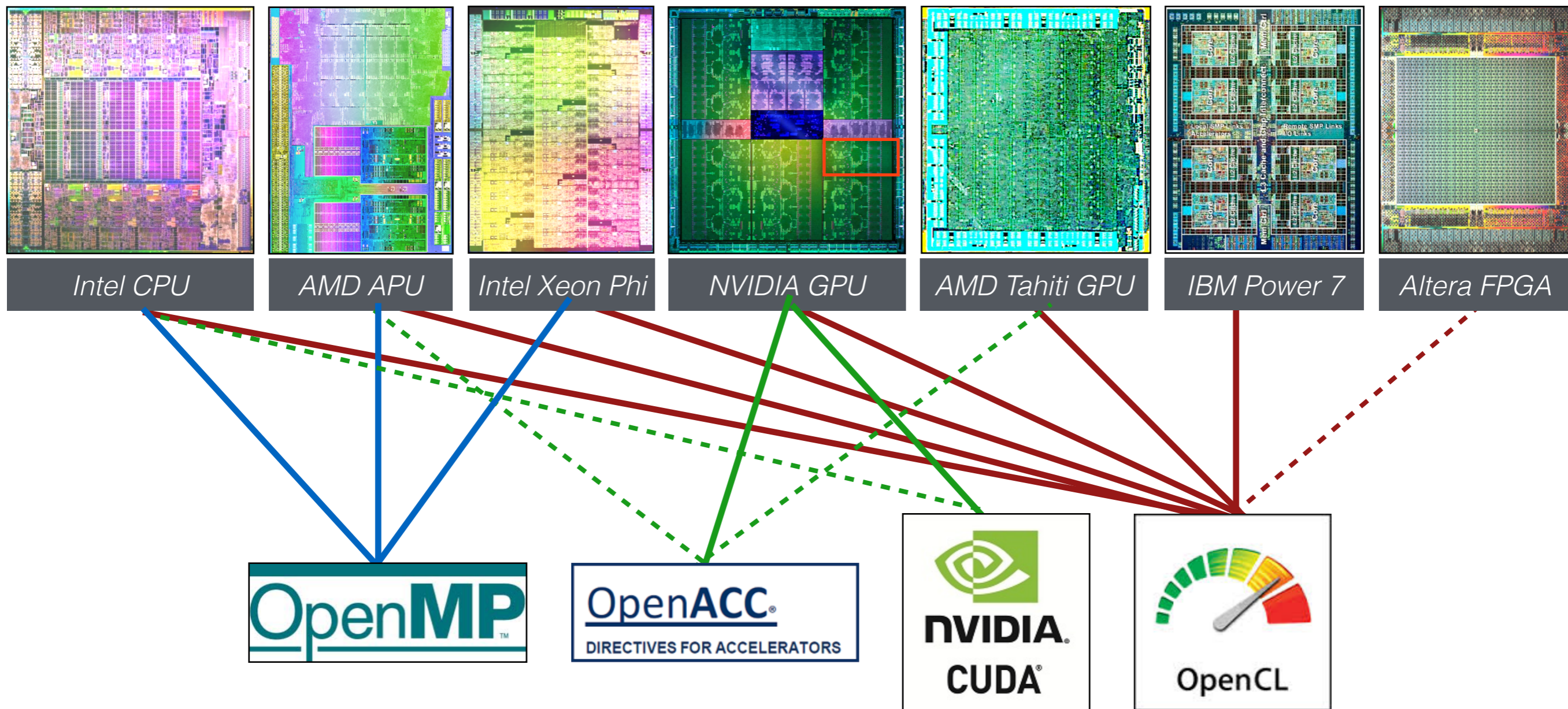
    if(t==0)
        blocksum[get_group_id(0)] = s_blocksum[0];
}
```

More details on OpenCL reduction next time.

Part 4b: Portability alternatives to OpenCL

Many-core: updates

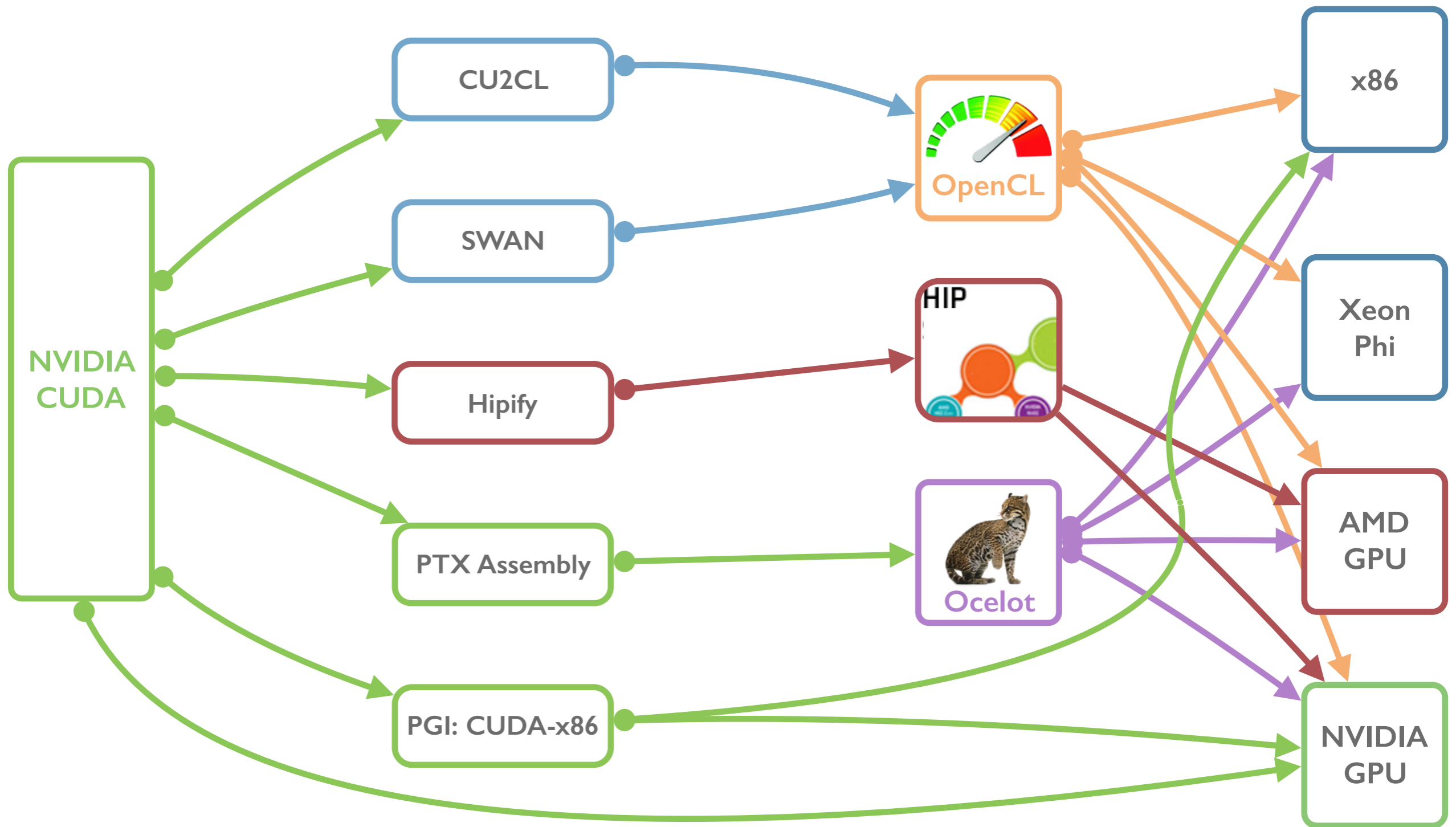
There are additional options



Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming

Many-core: porting from CUDA

Existing CUDA code can be ported to other frameworks.



*Translation to intermediate languages happens at the level of source code or assembly code.
However, this is predicated on using CUDA as the source language.*

Portability Approaches: directives

Directive approach

- Use of optional [`#pragma`]'s to give compiler transformation hints
- Aims for portability, **performance** and programmability



- Introduced for accelerator support through directives (2012)
- Compilers with OpenACC support:
 - gcc 6.1 (<https://gcc.gnu.org/wiki/OpenACC>), OpenACC toolkit (<https://developer.nvidia.com/openacc>), omni-compiler (<http://omni-compiler.org>)



- OpenMP has been around for a while (1997)
- OpenMP 4.0 specifications (2013) includes accelerator support



```
#pragma omp target teams distribute parallel for
for(int i = 0; i < N; ++i){
    y[i] = a*x[i] + y[i];
}
```

Portability: directives & data movement

Directive approach

- Not centralized anymore due to the offload model
- OpenACC and OpenMP begin to resemble an API rather than code decorations

OpenACC
Directives For Accelerators

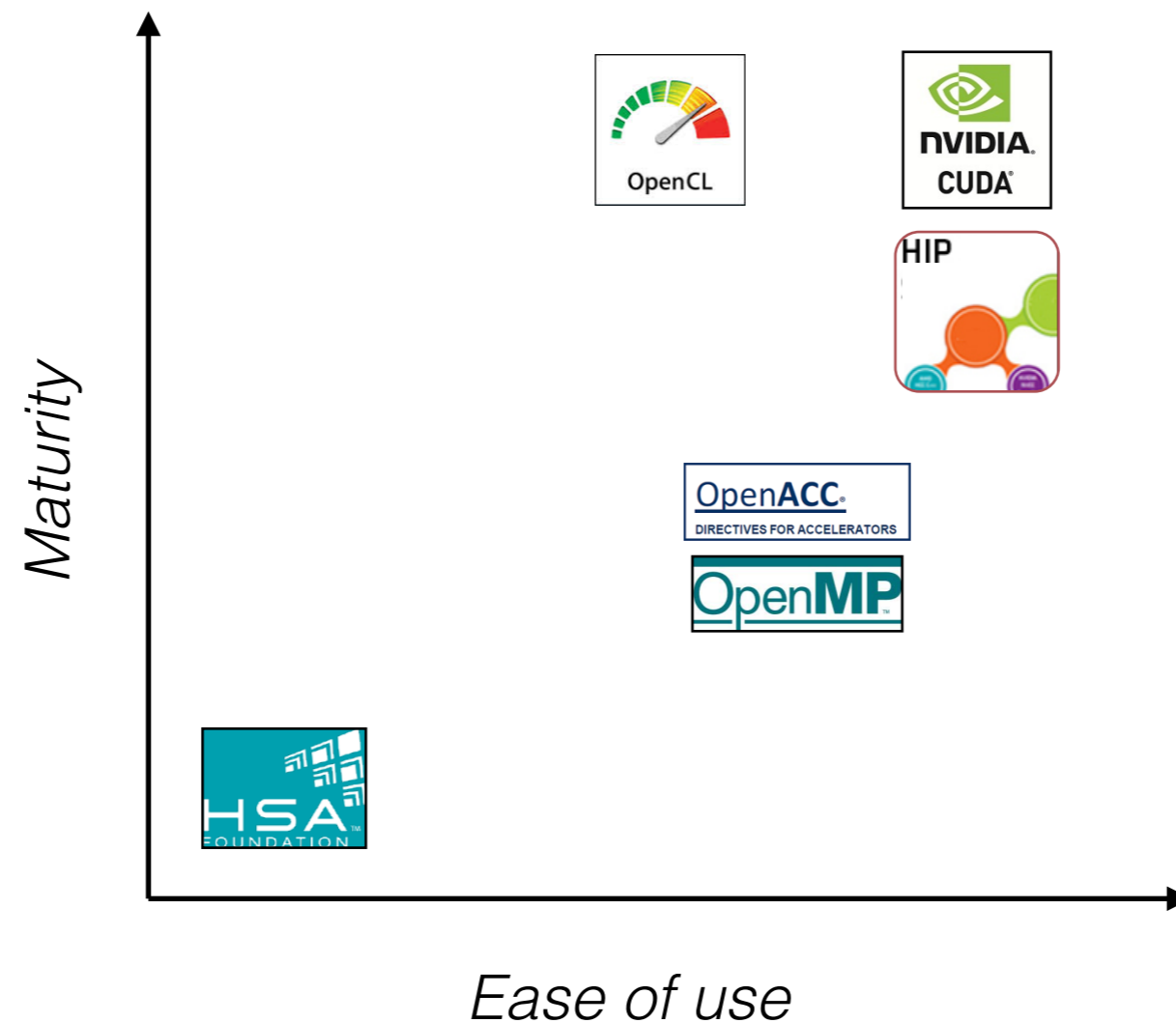
```
double a[100];  
#pragma acc enter data copyin(a)  
// OpenACC code  
#pragma acc exit data copyout(a)
```

OpenACC
Directives For Accelerators

```
class Matrix {  
    double *v;  
    int len  
    Matrix(int n) {  
        len = n;  
        v = new double[len];  
        #pragma acc enter data create(v[0:len])  
    }  
    ~Matrix() {  
        #pragma acc exit data delete(v[0:len])  
        delete[] v;  
    }  
};
```

Portability: ease of use

My opinion on “Maturity” balanced against “Ease of use”
for portable many-core programming



Still need an easy, efficient, durable, portable, open-source, vendor-independent approach for many-core programming

Step Back: MPI + X ?

Which “X” is going to dominate on-node threaded computing ?

MPI +
MPI

MPI +
OpenMP

MPI +
pThreads

MPI +
CUDA

MPI +
OpenCL

MPI +
OpenACC

MPI +
TBB

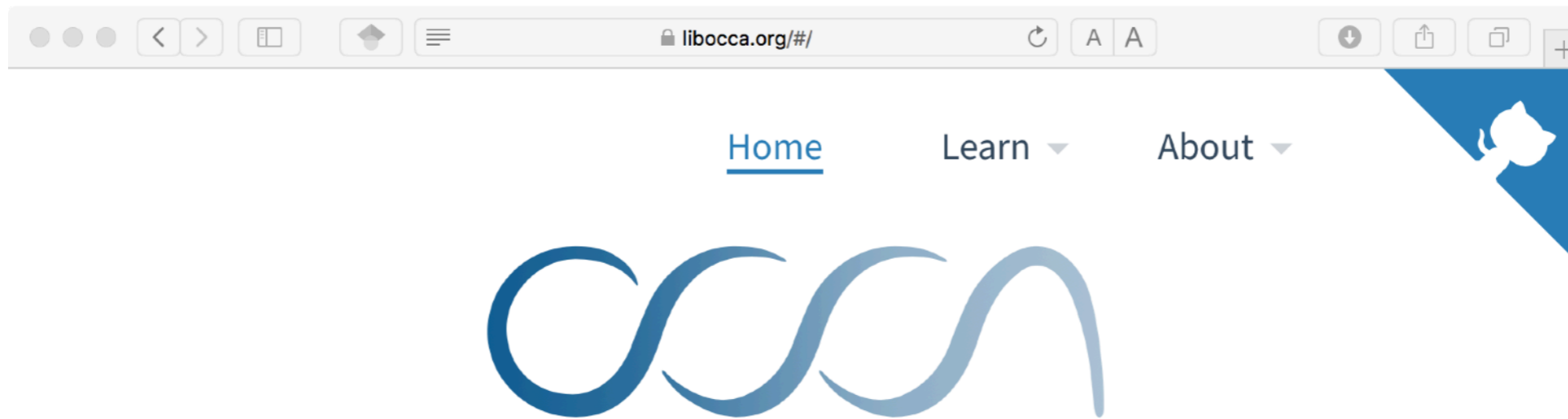
MPI +
Cilk Plus

MPI +
?

Should it even matter what “X” is ?

Part 4c: OCCA
Open Concurrent
Compute Abstraction

OCCA: easy portability



What is OCCA?

In a nutshell, OCCA (like *oca-rina*) is an open-source library which aims to

- Make it easy to program different types of devices (e.g. *CPU*, *GPU*, *FPGA*)
- Provide a **unified API** for interacting with backend device APIs (e.g. *OpenMP*, *CUDA*, *OpenCL*)
- Use just-in-time compilation to build backend kernels
- Provide a **kernel language**, a minor extension to C, to abstract programming for each backend

Quick Navigation

github.com/libocca/occa
libocca.org

GPU: Marmite[®] of the HPC world



People love or hate GPUs
& the source is messy

Open Concurrent Compute Abstraction (OCCA)

Goals:

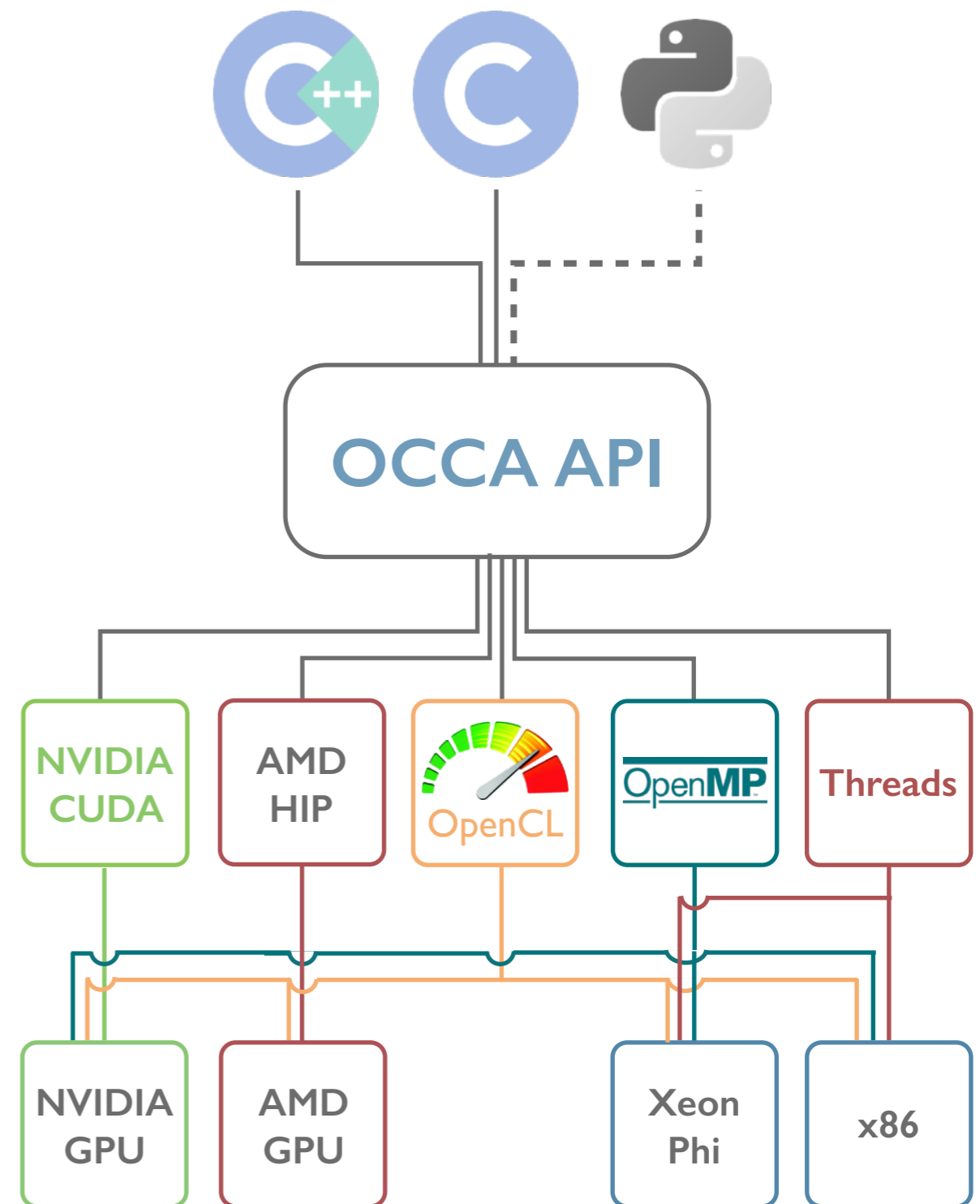
- Portability.
- Native code performance.
- Insulate simulation codes from HPC churn.
- Reduce the Marmite-ness of GPUs.

Design Principles:

- Simplicity.
- Unified interface.
- Limited dependencies.
- Explicit offload compute model.
- Kernel language: lightly annotated C.

Codes Exploring OCCA:

- libParanumal, ESDGSEM, NUMA, GNuMe, Nek5K*, libCEED, MFEM, laghos...



What does OCCA **not** do ?

Open Concurrent Compute Architecture, no magic unicorns.

Auto-parallelize:

- Some programmer intervention is required to identify parallel for loops.

Auto-optimize:

- Programmer knowledge of architecture is still invaluable.

Auto-layout:

- The programmer needs to decide how data is arranged in memory.

Auto-distribute:

- You can use MPI+OCCA but you have to write the MPI code.
- We considered M-OCCA but it devolves quickly into a PGAS.

Low-level code:

- We do not circumvent the vendor compilers.

OCCA: give it a spin & live demo

Building the OCCA library:

```
git clone https://github.com/libocca/occa -b 0.2
cd occa
export OCCA_DIR=`pwd`
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OCCA_DIR/lib
make -j
```

Building example:

```
cd examples/addVector/cpp
make
./main
```

Try changing the threading model to OpenCL, CUDA, or OpenMP:

```
emacs main.cpp
```

Portability: approaches of use

Numerous approaches to portability

API	Type	Front-ends	Kernel	Back-ends
Kokkos	ND arrays	C++	Custom	CUDA, OpenMP, & ROCm
VexCL	Vector class	C++	-	CUDA & OpenCL
RAJA	Library	C++	C++ Lambdas	CUDA, OpenMP, TBB std::thread, ROCm
→ OCCA	API, Source-to-source, Kernel Languages	C,C++	OpenCL, CUDA,& custom unified kernel language	CUDA, OpenCL, Threads, OpenMP, HIP (ROCm)
CU2CL *	Source-to-source	App	CUDA	OpenCL
Insieme	Source-to-source compiler	C	OpenMP, Cilk, MPI, OpenCL	OpenCL, MPI, Insieme IR runtime
Trellis	Directives	C/C++	#pragma trellis	OpenMP, OpenACC, CUDA
OmpSs	Directives + kernels	C,C++	Hybrid OpenMP, OpenCL, CUDA	OpenMP, OpenCL, CUDA
Ocelot	PTX Translator	CUDA	CUDA	OpenCL

DSL

Compiler

OCCA emphasis: lightweight and extensible.

** Wu Feng et al @ VT !*

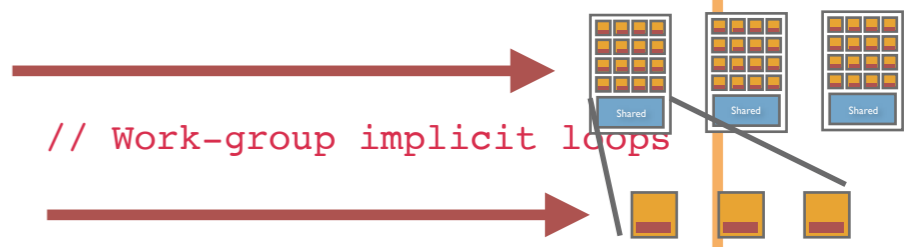
OCCA:OKL kernel language

Description

- Minimal extensions to C, familiar for regular programmers
- Explicit loops expose parallelism for modern multicore CPUs and accelerators
- Parallel loops are explicit through the fourth for-loop **inner** and **outer** labels

```
kernel void kernelName(...){
  ...
  for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
    for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
      for(int groupX = 0; groupX < xGroups; ++groupX; outer0){

        for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
          for(int itemY = 0; itemY < yItems; ++itemY; inner1){
            for(int itemX = 0; itemX < xItems; ++itemX; inner0){
              // GPU Kernel Scope
            }
          }
        }
      }
    }
  }
  ...
}
```



// Work-group implicit loops

// Work-item implicit loops



```
dim3 blockDim(xGroups, yGroups, zGroups);
dim3 threadDim(xItems, yItems, zItems);
kernelName<<< blockDim , threadDim >>>(...);
```



OCCA:OKL kernel language

Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL **kernels**
- Extension: allow for multiple outer-loops per kernel

```
kernel void kernelName(...){
    ...

    for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
        for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
            for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
                for(outer){
                    for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
                        for(int itemY = 0; itemY < yItems; ++itemY; inner1){
                            for(int itemX = 0; itemX < xItems; ++itemX; inner0){ // Work-item implicit loops
                                // GPU Kernel Scope
                            }
                        }
                    }
                }
            }
        }
    }
    ...
}
```

OKL

OCCA:OKL kernel language

Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL **kernels**
- Extension: allow for multiple outer-loops per kernel

```
kernel void kernelName(...){  
  for(outer){  
    for(inner){  
    }  
  }  
  
  for(outer){  
    for(inner){  
    }  
  }  
  
  for(outer){  
    for(inner){  
    }  
  }..  
}
```

A blue circular logo with the text "OKL" in white, positioned in the bottom right corner of the code block's container.

OCCA:OKL kernel language

Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL **kernels**
- Extension: allow for multiple outer-loops per kernel

```
kernel void kernelName(...){
kernel void kernelName(...){
  if(expr){
    for(outer){
      for(inner){
        } }
    } }
  else{
    for(outer){
      for(inner){
        } }
    } }
  }
  for(outer){
  while(expr){
    for(outer){
    } for(inner){
  } }
  }
  }
}
```

OKL

OCCA:OKL kernel language

Shared memory

```
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
  shared int sharedVar[16];

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    sharedVar[itemX] = itemX;
  }

  // Auto-insert [barrier(localMemFence);]

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
  }
}
```

OKL

Exclusive memory

```
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
  exclusive int exclusiveVar, exclusiveArray[10];

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    exclusiveVar = itemX; // Pre-fetch
  }

  // Auto-insert [barrier(localMemFence);]

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    int i = exclusiveVar; // Use pre-fetched data
  }
}
```

OKL

OCCA:OKL kernel language

Shared memory

```
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
  shared int sharedVar[16];

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    sharedVar[itemX] = itemX;
  }

  // Auto-insert [barrier(localMemFence);]

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
  }
}
```

OKL

Exclusive memory (similar to threadPrivate)

```
exclusiveVar = 0;
exclusiveVar = 1;
exclusiveVar = 2;
.
.
.
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
  exclusive int exclusiveVar, exclusiveArray[10];

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    exclusiveVar = itemX; // Pre-fetch
  }

  // Auto-insert [barrier(localMemFence);]

  for(int itemX = 0; itemX < 16; ++ itemX; inner0){ // Work-item implicit loops
    int i = exclusiveVar; // Use pre-fetched data
  }
}
```

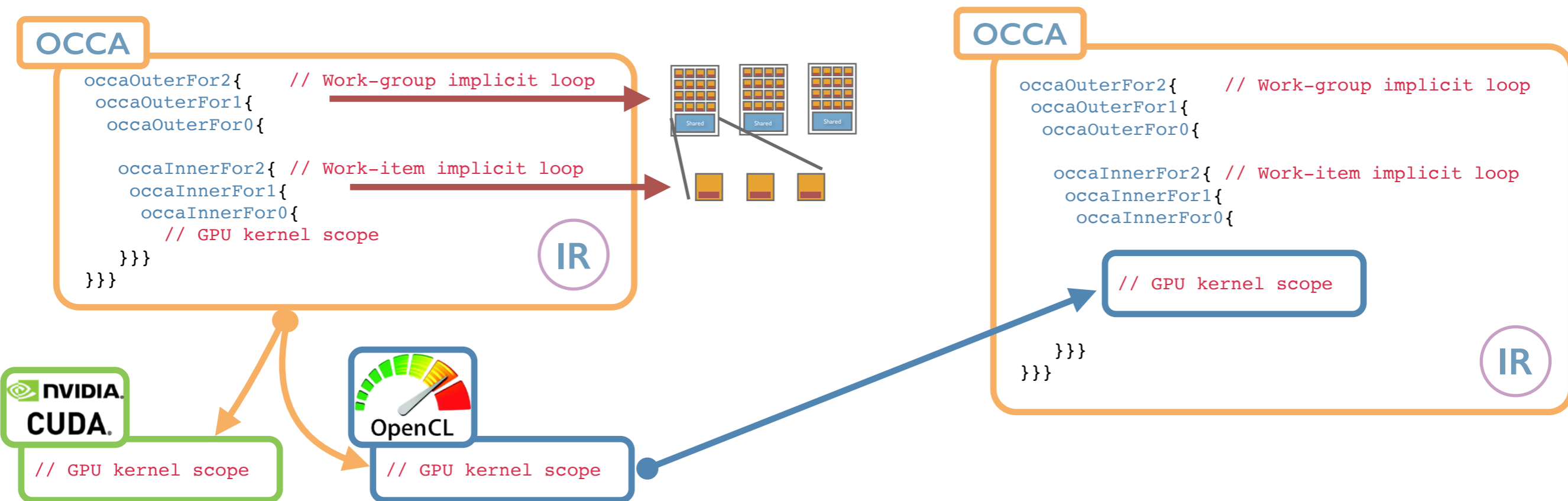
OKL

Local barriers are auto-inserted

OpenCL/CUDA to OCCA IR

Description

- Parser can translate OpenCL/CUDA kernels to OCCA IR*
- Although OCCA IR was derived from the GPU model, there are complexities



Since we derived OCCA IR from the GPU model, the inverse should be easy ... right?

OCCA: example adding two vectors

```
#include <iostream>

#include "occa.hpp"

int main(int argc, char **argv){
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i){
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << ": " << ab[i] << '\n';
}
```

OCCA: example adding two vectors

```
#include <iostream>

#include "occa.hpp"

int main(int argc, char **argv)
{
    int N = 50;
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i)
        a[i] = i;
    for(int i = 0; i < N; ++i)
        b[i] = 1 - i;
    for(int i = 0; i < N; ++i)
        ab[i] = 0;

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("OpenCL", 0, 0); // (Platform, Device) = (0, 0)

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", 'addVectors');

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << ": " << ab[i] << '\n';
}
```

```
kernel void addVectors(const int entries,
                       const float * a,
                       const float * b,
                       float * ab){
    for(int b=0;b<entries;b+=10;outer0){
        for(int n=b;n<b+10;++n;inner0){
            if(n < entries)
                ab[n] = a[n] + b[n];
        }
    }
}
```



Native

```
#include "stdlib.h"
#include "st

#include

int main(
    int i;

    float *a;
    float *b;
    float *ab;

    for(i = 0; i < 5; ++i)
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occaDev_t dev;
    occaKernel_t kernel;
    occaMem_t mem;

    device = occa.device('OpenCL', 0, 0);

    o_a = occa.malloc(device, a, 'single');
    o_b = occa.malloc(device, b, 'single');
    o_ab = occa.malloc(device, ab, 'single');

    addVectors = occa.buildKernelFromSource('addVectors.occa', ...
        'addVectors');

    dims = 1;
    itemsPerGroup = 2;
    groups = (entries + itemsPerGroup - 1)/itemsPerGroup;

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(occa.type(entries, 'int32'), ...
        o_a, o_b, o_ab);

    ab = o_ab(:);

    addVectors.s

    addVectors([c_int(entries),
        o_a, o_b, o_ab])

    o_ab.copyTo(ab, c_float)

    print ab

    occa.runKernel(addVectors,
        (entries,
        o_a, o_b, o_ab)

    occa.memcpy(ab, o_ab)

    println(ab)

    occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

    for(i = 0; i < 5; ++i)
        printf("%d = %f\n", i, ab[i]);
    }
```

```
from ctypes
import occa
```

```
require( bytestri
```

```
entries = 5
```

```
device = occa.dev
```

```
# Dynamic range?
```

```
a = Float32[1 -
```

```
b = Float32[i
```

```
ab = Float32[0
```

```
o_a = occa.mallo
```

```
o_b = occa.mallo
```

```
o_ab = occa.mallo
```

```
addVectors = occa
```

```
dims = 1;
```

```
itemsPerGroup = 2
```

```
groups = (entries
```

```
occa.setWorkingDi
```

```
occa.runKernel(ad
```

```
occa.memcpy(ab, o_ab)
```

```
println(ab)
```

```
entries = 5
```

```
a = [i
```

```
b = [1 - i
```

```
ab = [0
```

```
device = occa
```

```
o_a = devic
```

```
o_b = devic
```

```
o_ab = devic
```

```
addVectors =
```

```
dims = 1
```

```
itemsPerGrou
```

```
groups = (en
```

```
addVectors.s
```

```
addVectors([c_int(entries),
```

```
o_a, o_b, o_ab])
```

```
o_ab.copyTo(ab, c_float)
```

```
print ab
```

```
entries = 5;
```

```
a = ones(entries, 1);
```

```
b = ones(entries, 1);
```

```
ab = zeros(entries, 1);
```

```
device = occa.device('OpenCL', 0, 0);
```

```
o_a = device.malloc(a , 'single');
```

```
o_b = device.malloc(b , 'single');
```

```
o_ab = device.malloc(ab, 'single');
```

```
addVectors = device.buildKernelFromSource('addVectors.occa', ...
    'addVectors');
```

```
dims = 1;
```

```
itemsPerGroup = 2;
```

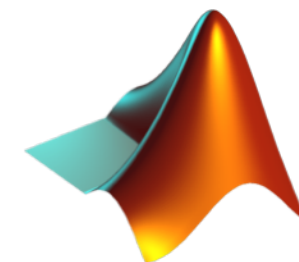
```
groups = (entries + itemsPerGroup - 1)/itemsPerGroup;
```

```
addVectors.setWorkingDims(dims, itemsPerGroup, groups);
```

```
addVectors(occa.type(entries, 'int32'), ...
    o_a, o_b, o_ab);
```

```
ab = o_ab(:);
```

```
ab
```



All the HOST codes use the same kernel.

Example HOST code: <https://github.com/tcew/OCCA2/tree/master/examples/addVectors>

OCCA: comparing Jacobi kernels

Simple Poisson example: comparison of serial v. CUDA v. OpenCL v. OCCA kernels

Serial kernel:

```
void jacobi(const int N,
            const datafloat *rhs,
            const datafloat *u,
            datafloat *newu){
    for(int i=0;i<N;++i){
        for(int j=0;j<N;++j){
```

CUDA kernel:

```
__global__ void jacobi(const int N,
                       const datafloat *rhs,
                       const datafloat *u,
                       datafloat *newu){
    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;
```

OpenCL kernel:

```
__kernel void jacobi(const int N,
                     __global const datafloat
                     __global const datafloat
                     __global datafloat *newu){
    // Get thread indices
    const int i = get_global_id(0);
    const int j = get_global_id(1);
```

OCCA IR kernel:

```
occaKernel void jacobi(occaKernelInfoArg,
                       const int occaVariable N,
                       occaPointer const datafloat *rhs,
                       occaPointer const datafloat *u,
                       occaPointer datafloat *newu){
    occaOuterFor1{
        occaOuterFor0{
            occaInnerFor1{
                occaInnerFor0{
                    // Get thread indices
                    const int i = occaGlobalId0;
                    const int j = occaGlobalId1;
                    if((i < N) && (j < N)){
                        // Get linear index into (N+2)x(N+2) grid
                        const int id = (j + 1)*(N + 2) + (i + 1);
                        newu[id] = 0.25f*(rhs[id]
                                           + u[id - (N+2)]
                                           + u[id + (N+2)]
                                           + u[id - 1]
                                           + u[id + 1]);
                    }
                }
            }
        }
    }
}
```

OKL kernel:

```
kernel void jacobi(const int N,
                   const datafloat *rhs,
                   const datafloat *u,
                   datafloat *newu){
    for(int start1=0; start1<N; start1+=BY; outer1){
        for(int start0=0; start0<N; start0+=BX; outer0){
            for(int j=start1; j<start1+BY; ++j; inner1){
                for(int i=start0; i<start0+BX; ++i; inner0){
                    if((i < N) && (j < N)){
                        // Get linear index into (N+2)x(N+2) grid
                        const int id = (j + 1)*(N + 2) + (i + 1);
                        newu[id] = 0.25f*(rhs[id]
                                           + u[id - (N+2)]
                                           + u[id + (N+2)]
                                           + u[id - 1]
                                           + u[id + 1]);
                    }
                }
            }
        }
    }
}
```

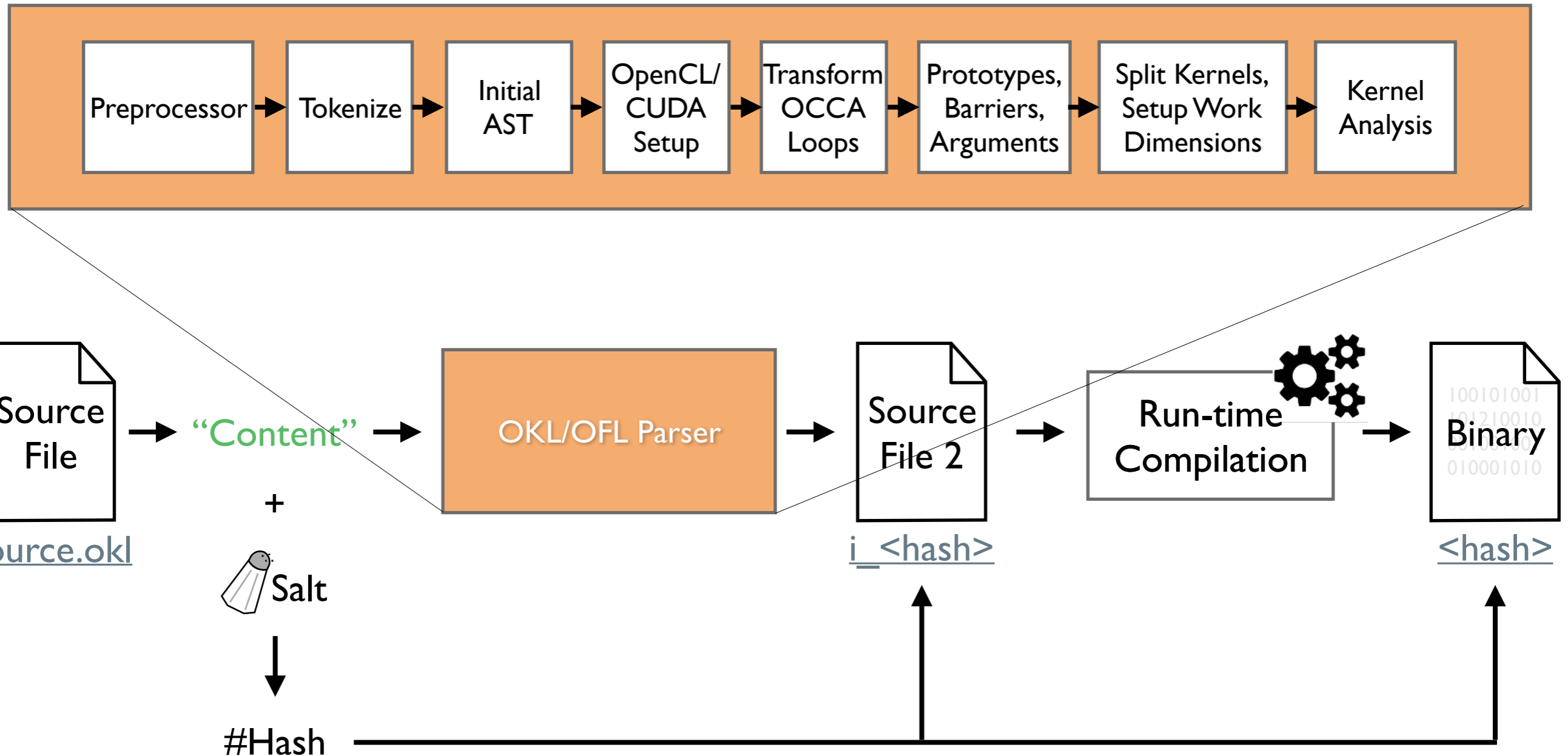
In OCCA we split the i and j loops both into outer and inner loops.

From the OCCA kernel we can reproduce the serial, CUDA, and OpenCL kernels (also pthreads, openmp...)

Online Compilation

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism, making use of the OCCA IR

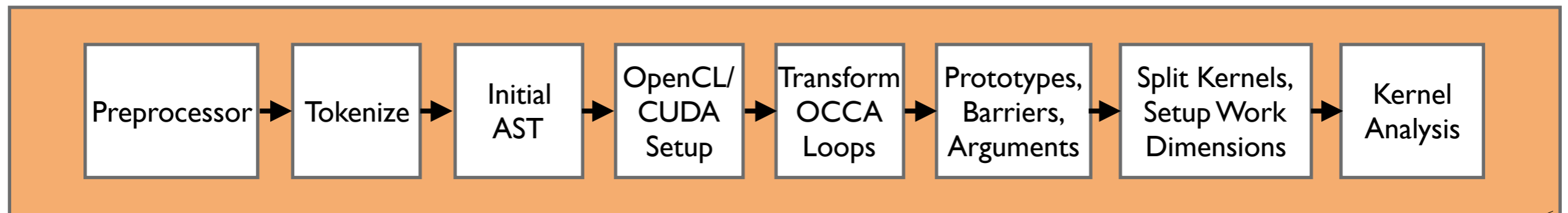


Custom compilation tools tailored for code manipulation and analysis

Behind the Scenes: caching and hashing

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism & make use of OCCA IR



```
#define N 10
```

```
int i = N;
```

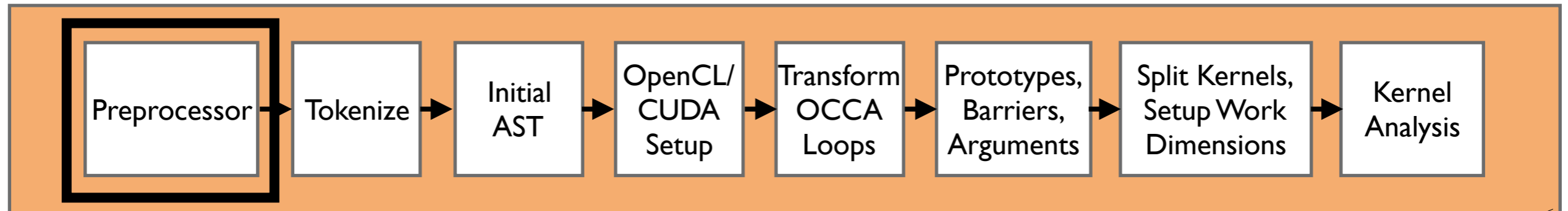
```
int i = 10;
```

#1 1a511

OCCA Infrastructure

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism & make use of OCCA IR



```
#define N 10
```

```
int i = N;
```

```
int i = 10;
```

```
int
```

```
i
```

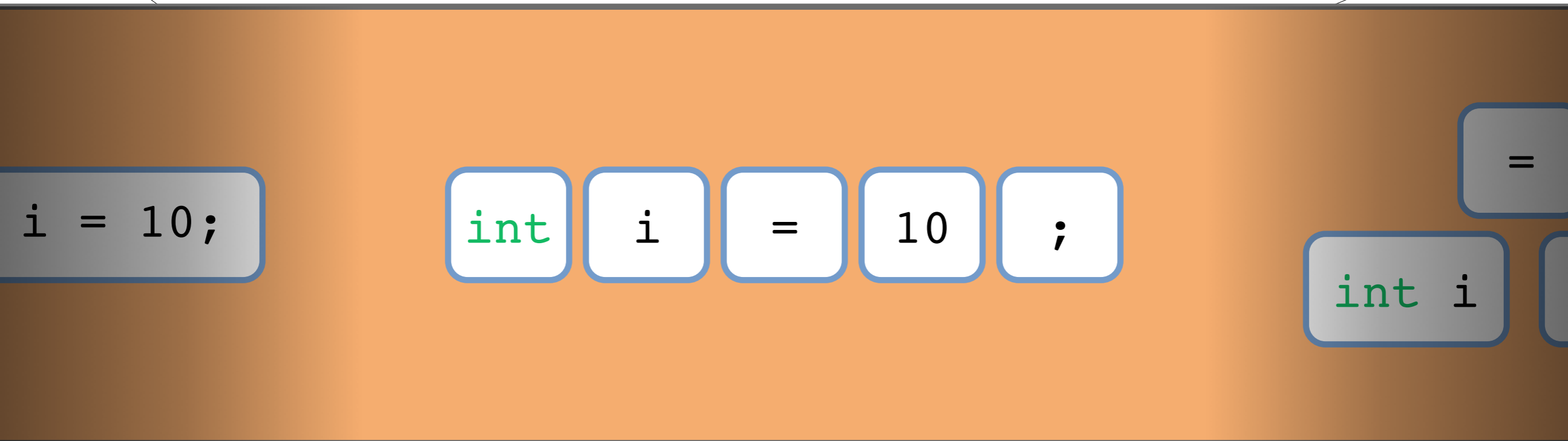
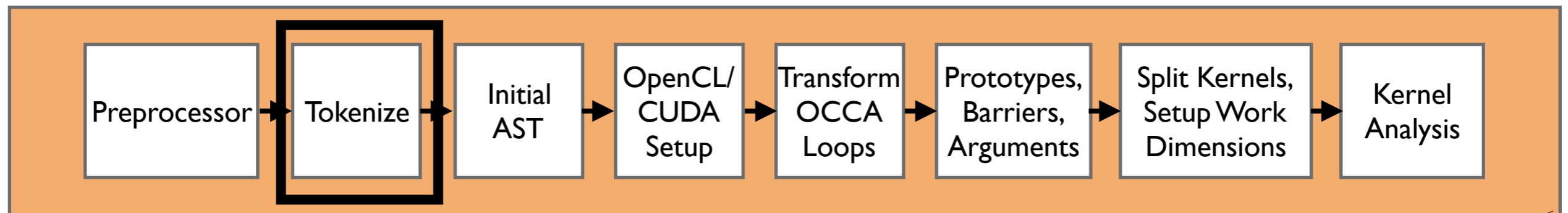
```
=
```

```
#if 1a511
```

OCCA Infrastructure

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism & make use of OCCA IR

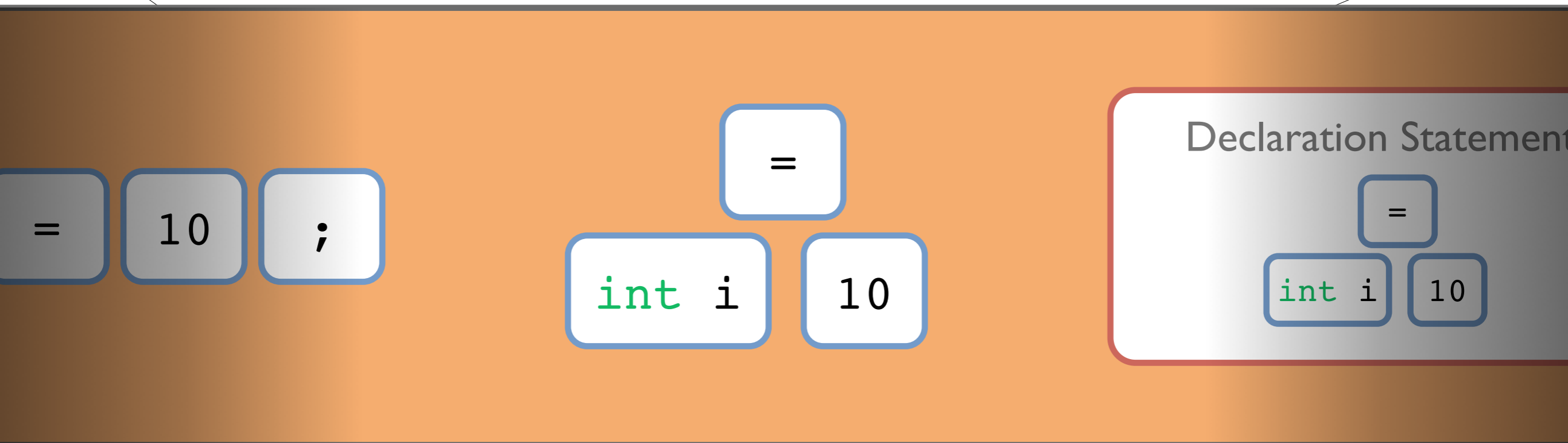
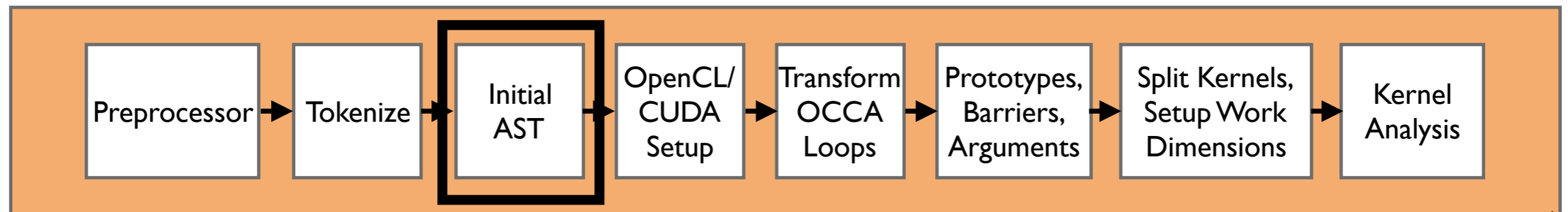


#1 | AST

OCCA Infrastructure

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism & make use of OCCA IR

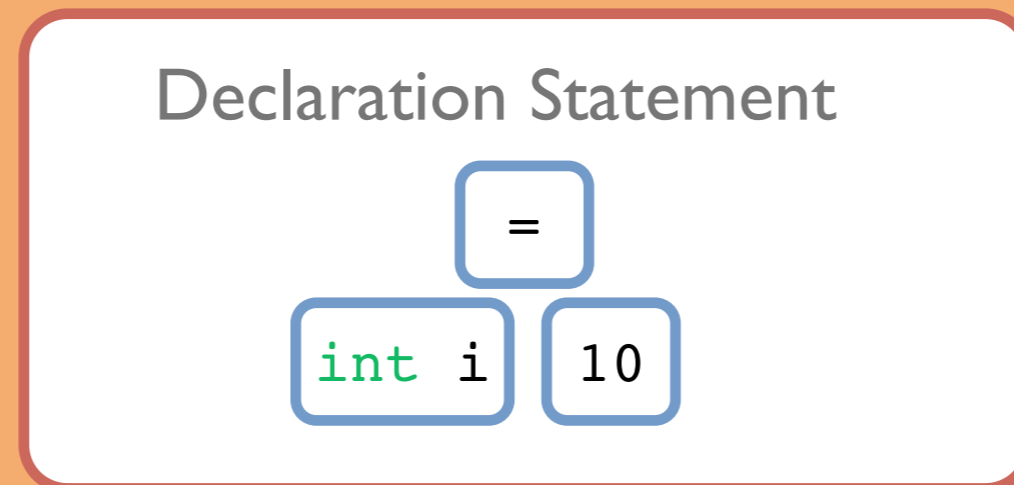
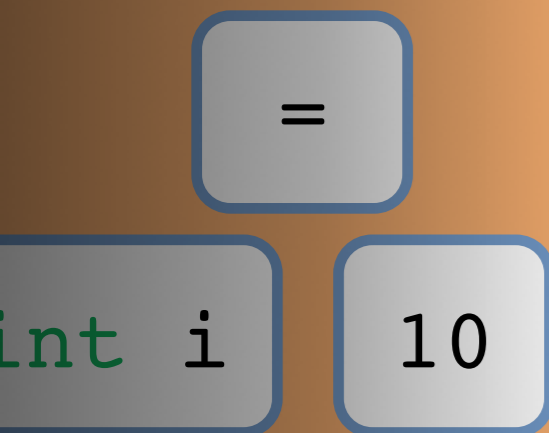
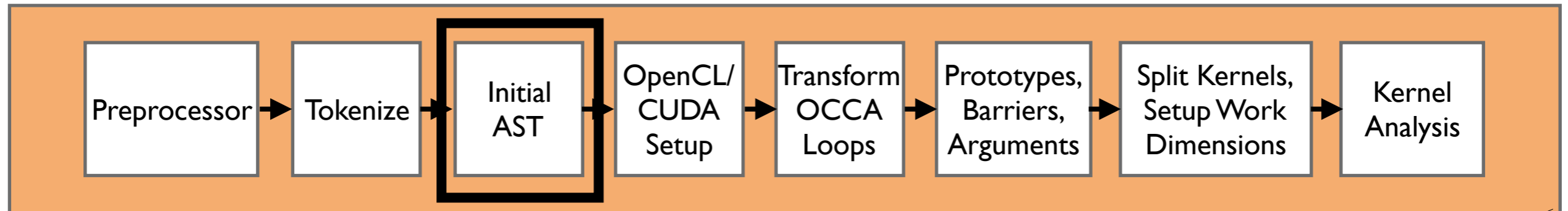


#1 | AST

OCCA Infrastructure

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism & make use of OCCA IR

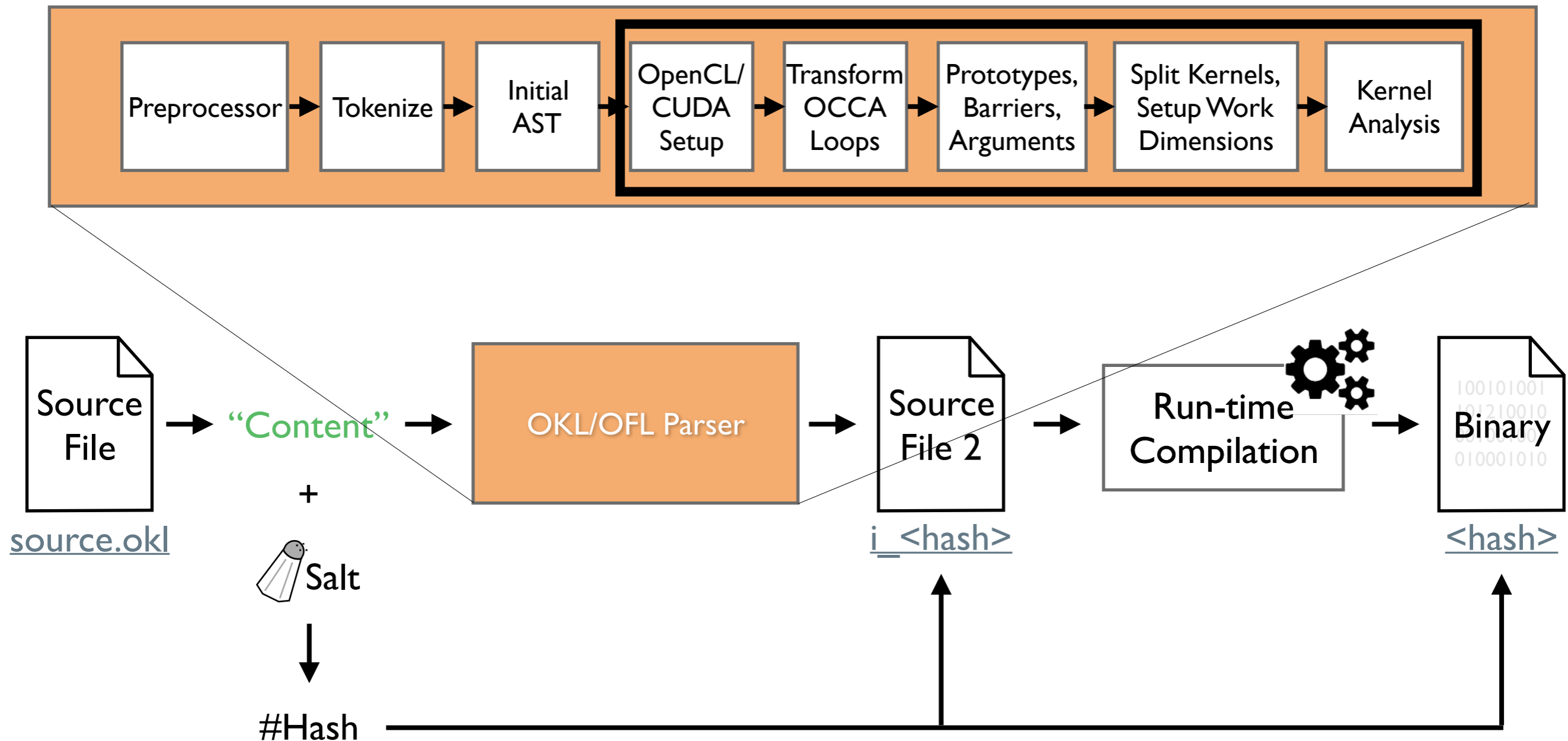


#1 1a511

OCCA Infrastructure

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism & make use of OCCA IR



Custom compilation tools tailored for code manipulation and analysis

Hands On #3: *OCCA flow simulation*

In this exercise you will create a flow simulation



Work in teams of two.

OCCA Flow Simulation: instructions

#1. build the OCCA library:

```
# login node: clone the OCCA repo
git clone https://github.com/libocca/occa -b 0.2

# compute node: cd to the OCCA directory
cd occa
# build OCCA
make -j

# add OCCA_DIR to env and add dynamic library path
export OCCA_DIR=`pwd`
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OCCA_DIR/lib
```

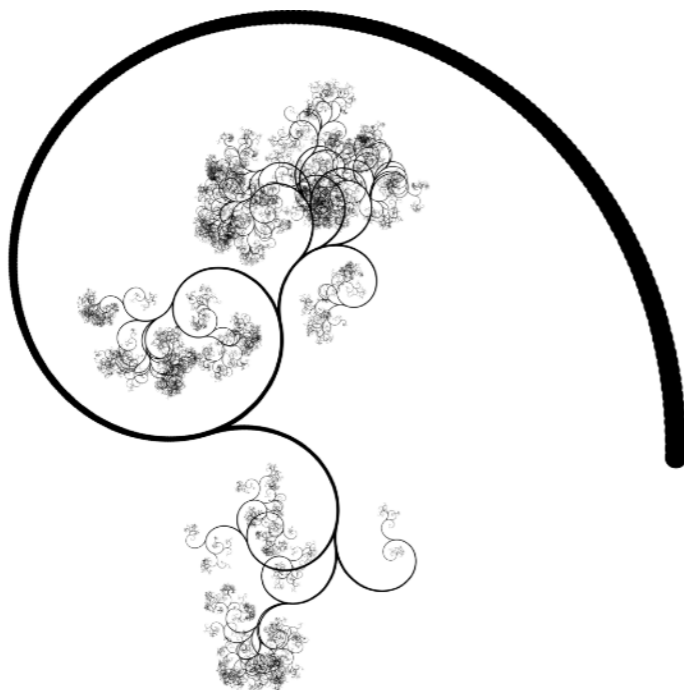
#2. build the OCCA LBM code:

```
# login node: clone the ATPESC18 repo
git clone https://github.com/tcew/ATPESC18

# compute node: cd to the lbm directory
cd ATPESC18/handsOn/lbm

# build OCCA lbm solver
make -f makefile.occa
```

#3. login node: save png image with white background to the lbm directory:

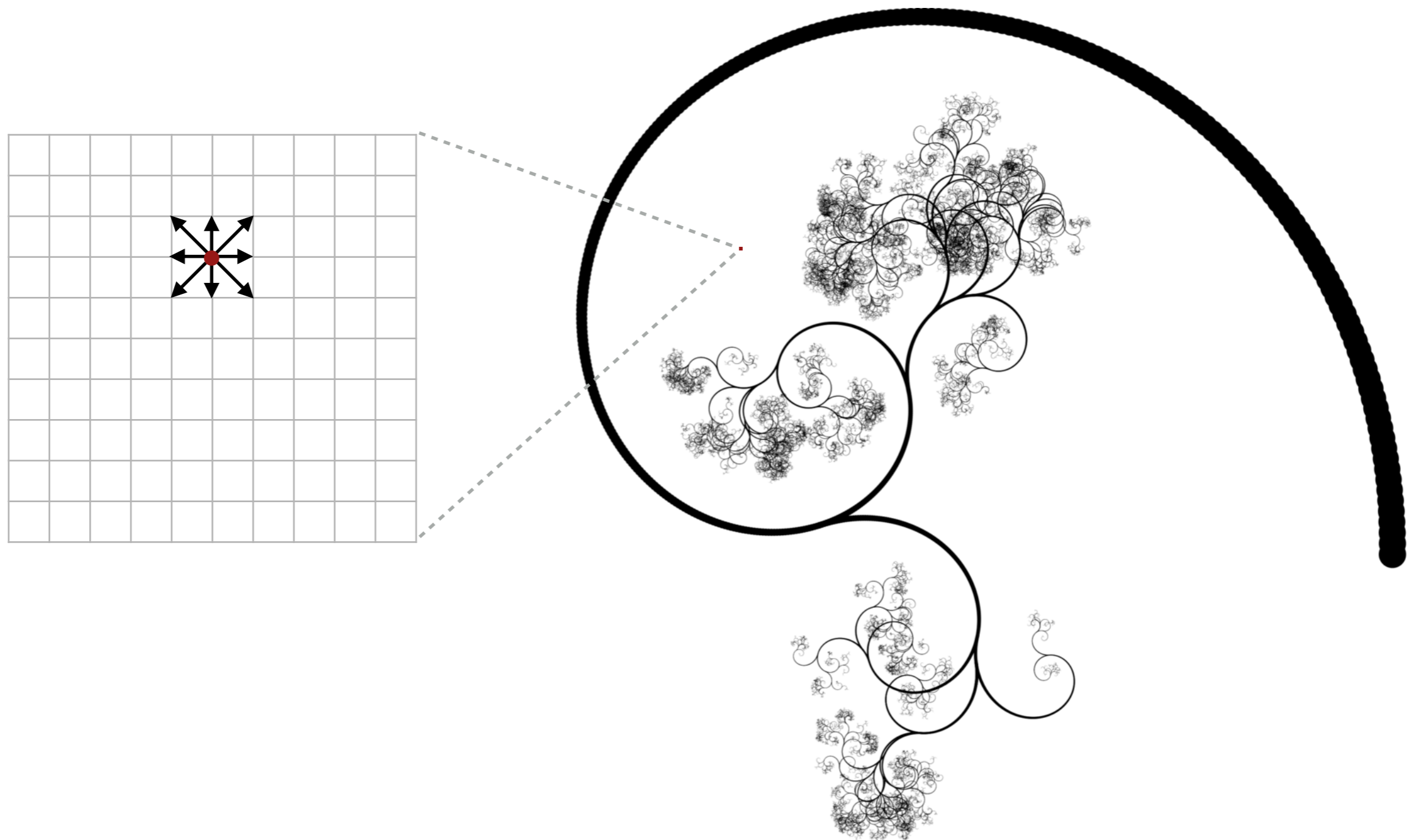


#4. run the lbm code with your png image # (using 400 as a flow volume threshold)

```
./occaLBM yourImageName.png 400
```

OCCA Flow Simulation: background

The image pixels become flow nodes in a lattice: the Lattice Boltzmann Method tracks the density of 9 species of colliding particles constrained to move on the lattice



Details: a D2Q9 lattice Boltzmann method is implemented using an OCCA update kernel that uses a single thread to update the 9 particle densities at each lattice node.

OCCA Flow Simulation: instructions

#5. the lbm code generates bah#####.png image files:

To make a movie:

```
ffmpeg -r 24 -i bah%06d.png -b:v 16384k -vf scale=1024:-1 foo.mp4
```

#6. transfer foo.mp4 to your laptop via globus and open with movie player:



Raise your hand and demo your movie when done.

OCCA Flow: changing thread model

The lbm code is set up to use CUDA by default.

#7. Find out what compute modes are available:

```
$OCCA_DIR/bin/occainfo
```

#8. change OCCA device setup in main to change the thread model:

```
occa::device device;  
// device.setup("mode=OpenCL, deviceID=1, platformID=0");  
device.setup("mode=CUDA, deviceID=0");  
// device.setup("mode=OpenMP");
```

#9. re-make the executable:

```
make -f makefile.occa
```

#10. rerun

```
./occaLBM yourImageName.png 400
```

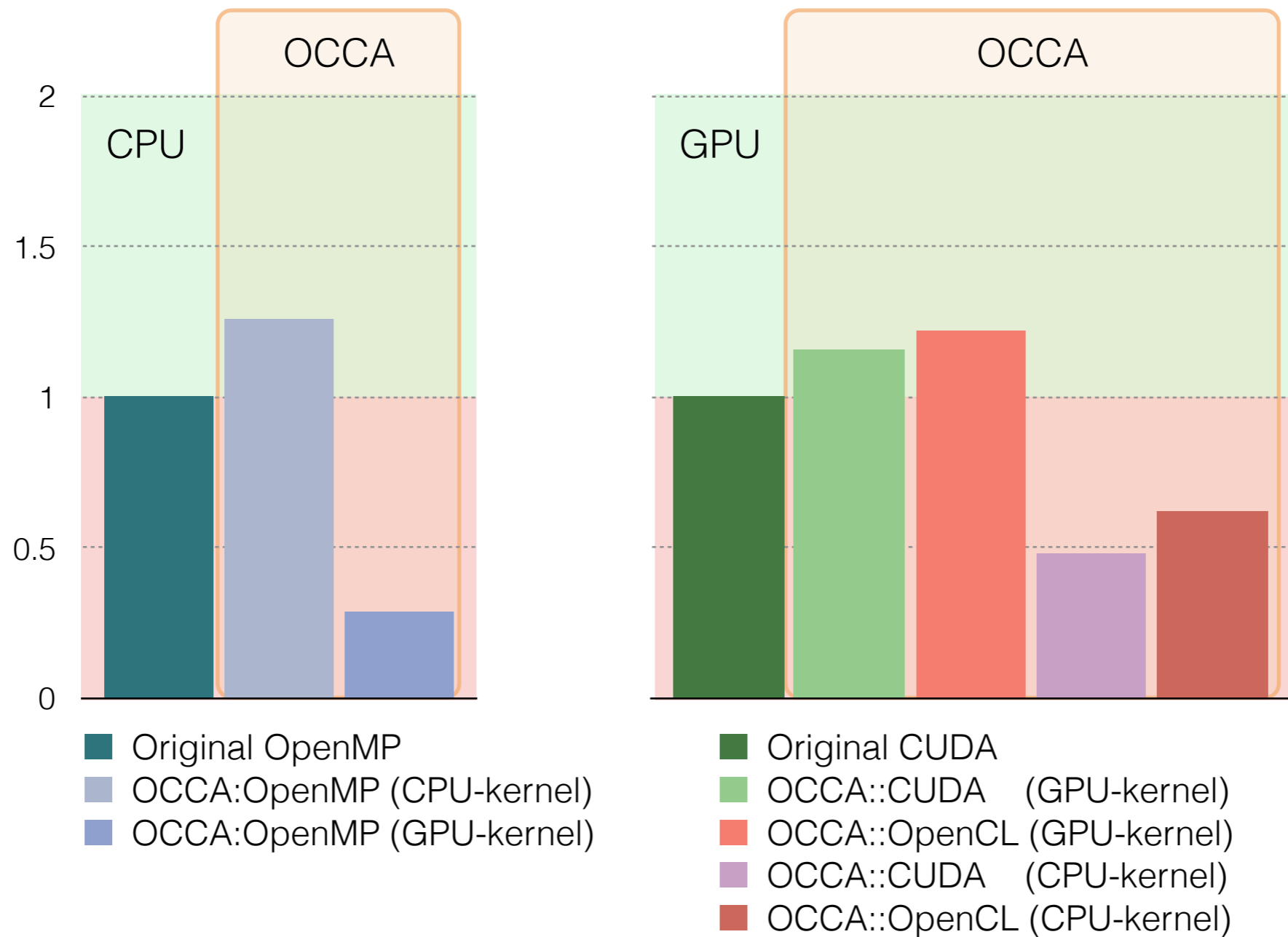
Do you notice a speed change ?

#11. try installing and running on your laptop - this might be tricky.

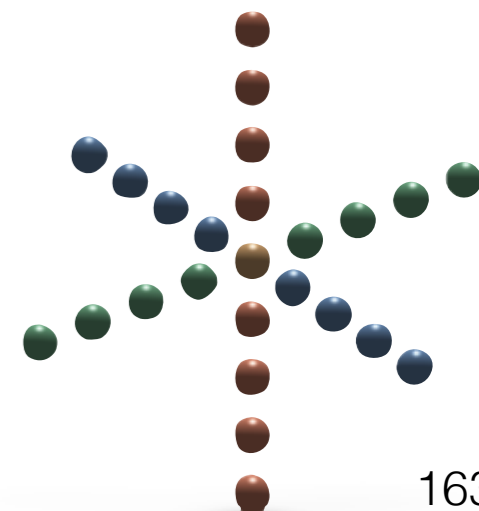
Congratulations: you have found out how easy it can be to switch compute mode with OCCA.

OCCA: apps & benchmarks

High-order finite difference for Reverse Time Migration (imaging algorithm)

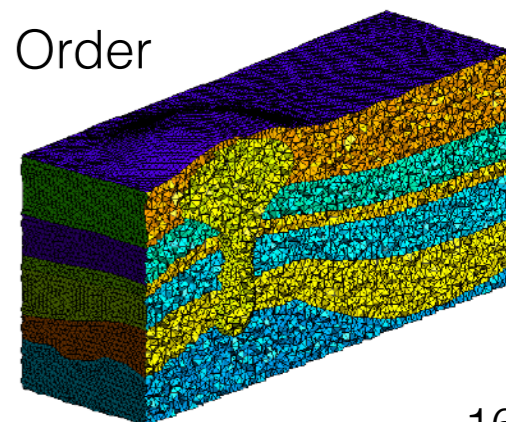
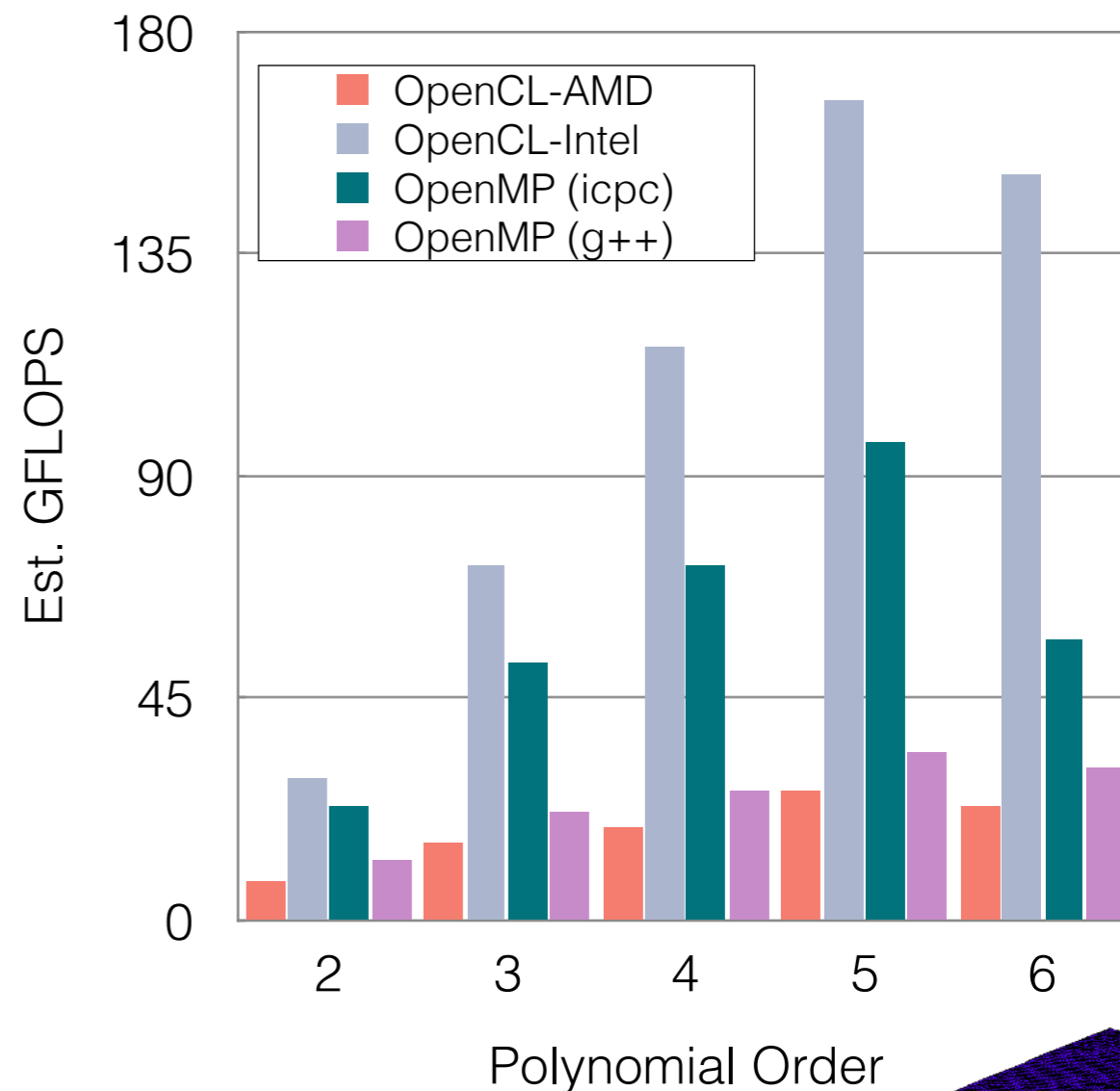
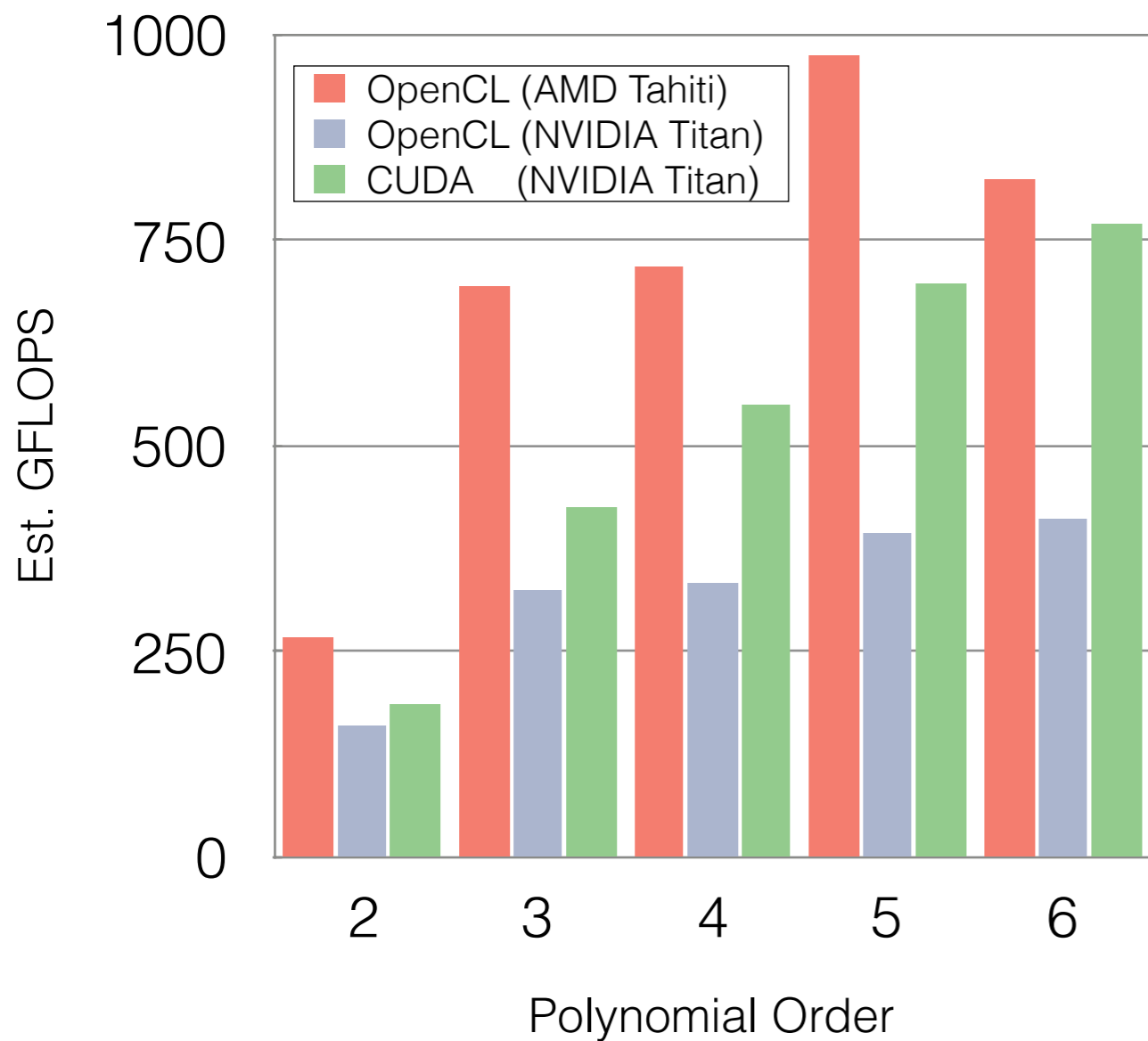


OpenMP : Intel Xeon CPU E5-2640
OpenCL/CUDA : NVIDIA Tesla K10



OCCA: apps & benchmarks

Discontinuous Galerkin for RTM



OCCA: apps & benchmarks

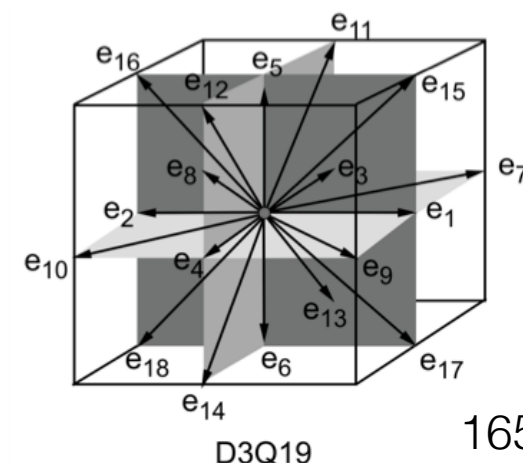
Lattice Boltzmann Method in Core Sample Analysis

Comparison across platforms (Normalized with original code)

	API Mode	Device	Model	Wall Clock	BW (GB/s)	Speedup
	Ref dense code [-O3 in gcc 4.8]	CPU 1-core	Intel i7-5960X	1290	—	x 1
OCCA	OpenMP	CPU	Intel i7-5960X	11.12	22	x 116
	OpenCL: Intel	CPU	Intel i7-5960X	11.18	22	x 115
	OpenCL: AMD	GPU	AMD 7990	1.39	176	x 928
	OpenCL: NVIDIA	GPU	GTX 980	1.25	196	x 1032
	CUDA: NVIDIA	GPU	GTX 980	1.20	205	x 1075

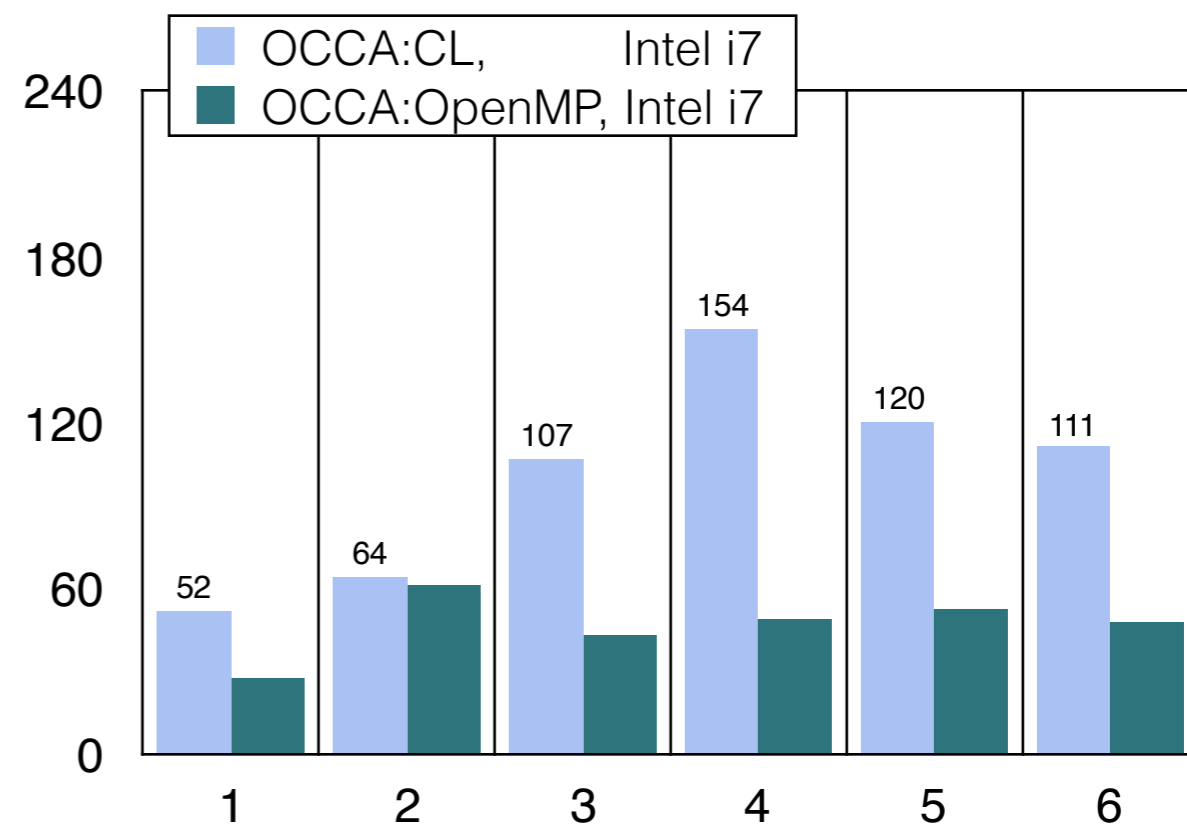
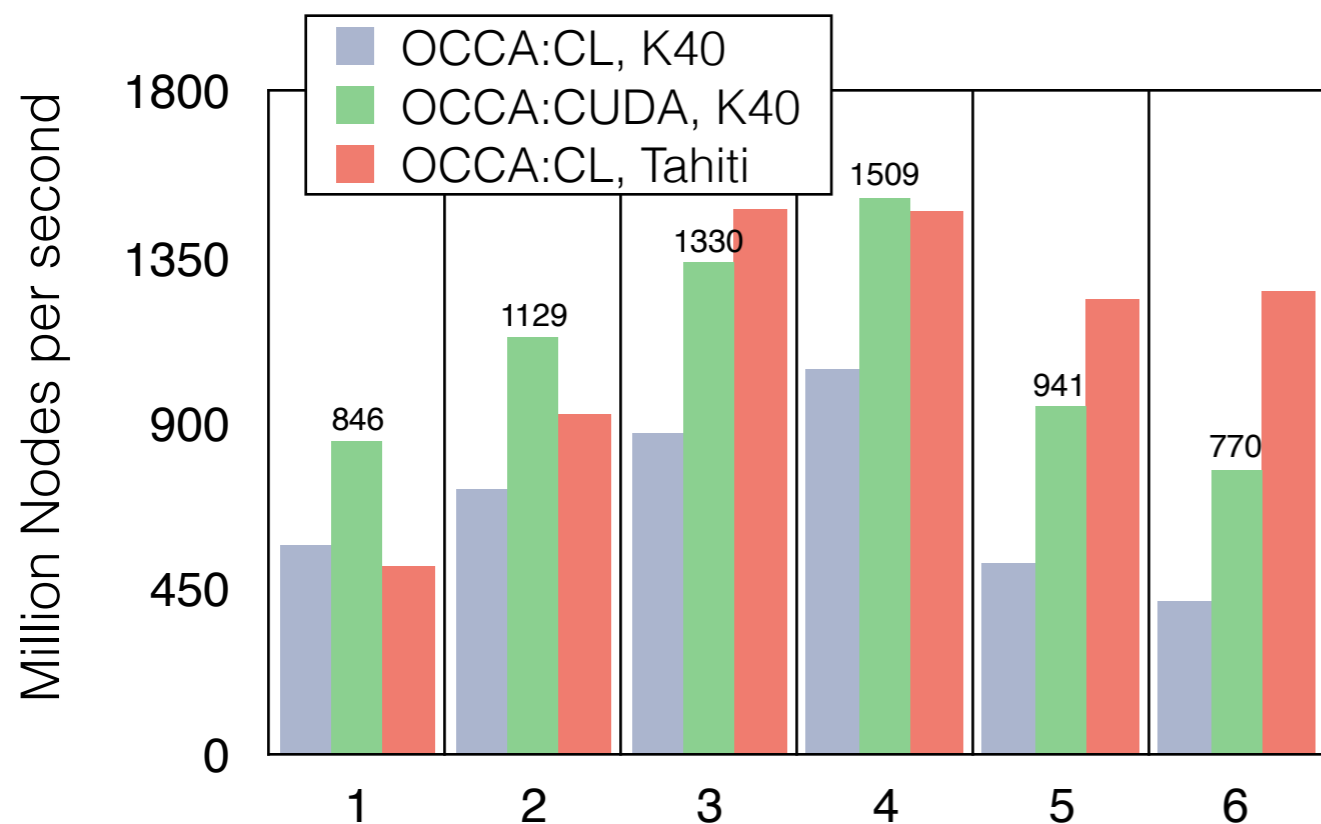
Comparison across platforms (Normalized with OCCA::OpenMP)

	API Mode	Device	Model	Wall Clock	BW (GB/s)	Speedup
OCCA	OpenMP	CPU	Intel i7-5960X	11.12	22	x 1.0
	OpenCL: Intel	CPU	Intel i7-5960X	11.18	22	x 1.0
	OpenCL: AMD	GPU	AMD 7990	1.39	176	x 8.0
	OpenCL: NVIDIA	GPU	GTX 980	1.25	196	x 8.9
	CUDA: NVIDIA	GPU	GTX 980	1.20	205	x 9.3

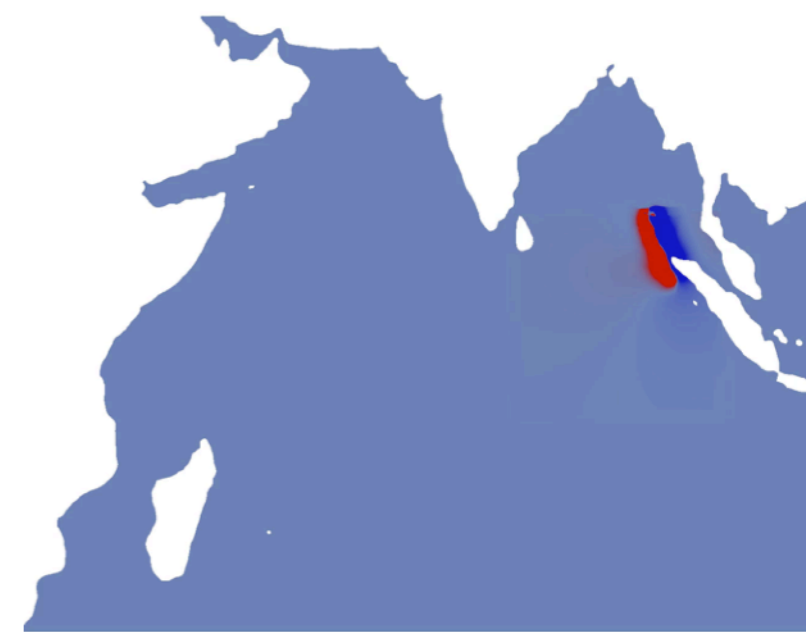


Applications

Discontinuous Galerkin for shallow water equations



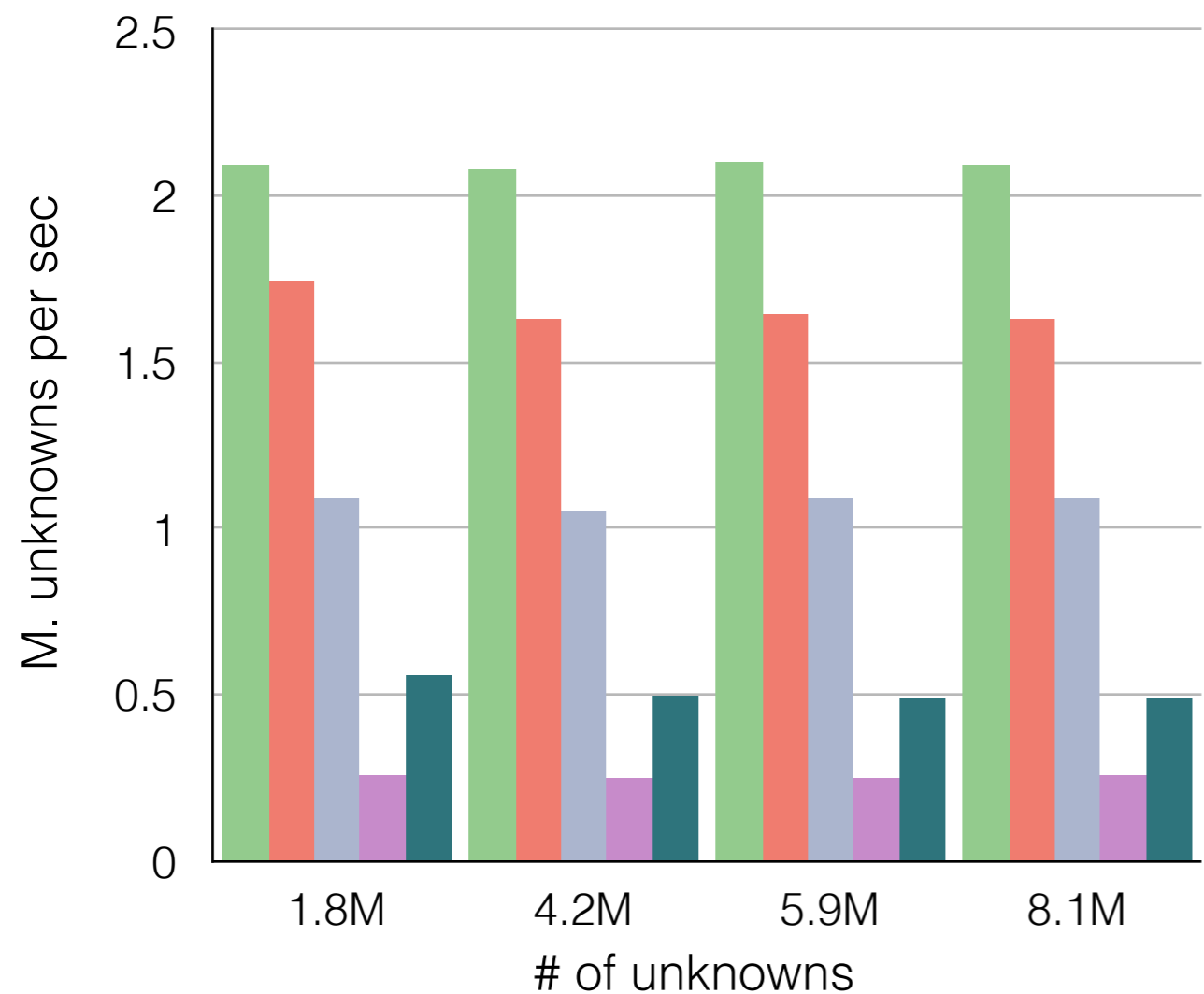
Polynomial Order	Compute-Time vs Real-Time
1	x650
2	x208
3	x95
4	x47



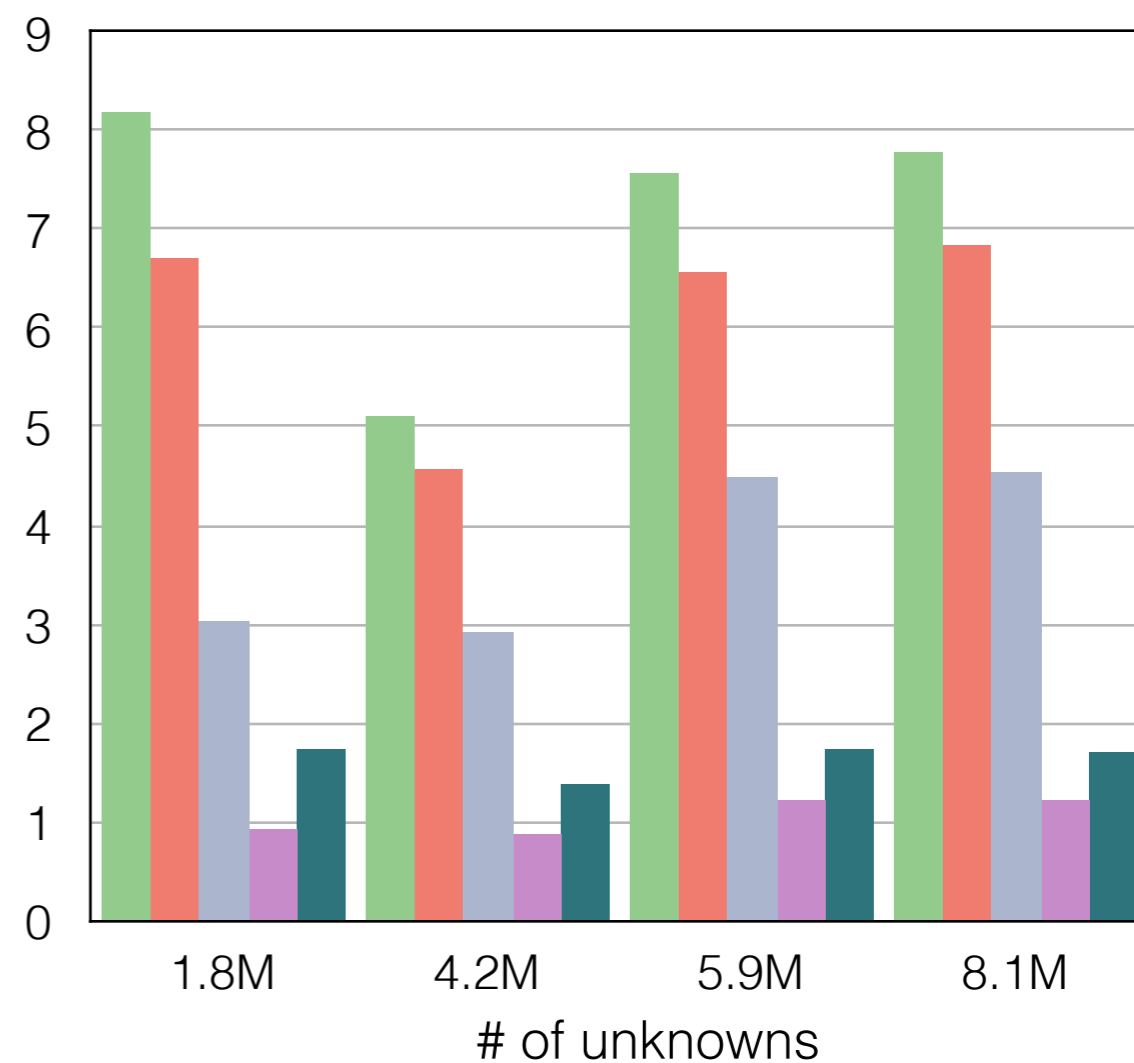
OCCA: apps & benchmarks

Algebraic multigrid for elliptic problems

Setup Time



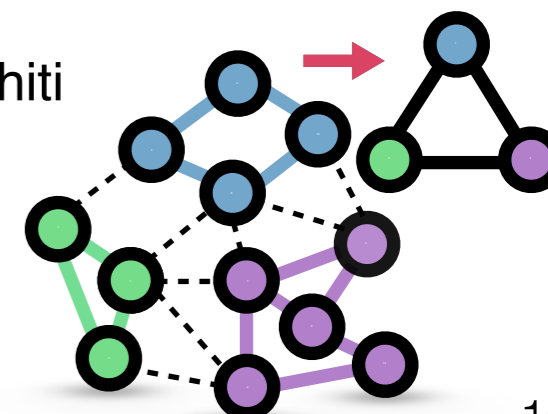
Solve Time



■ CUDA on Titan
■ OpenCL on Intel i7

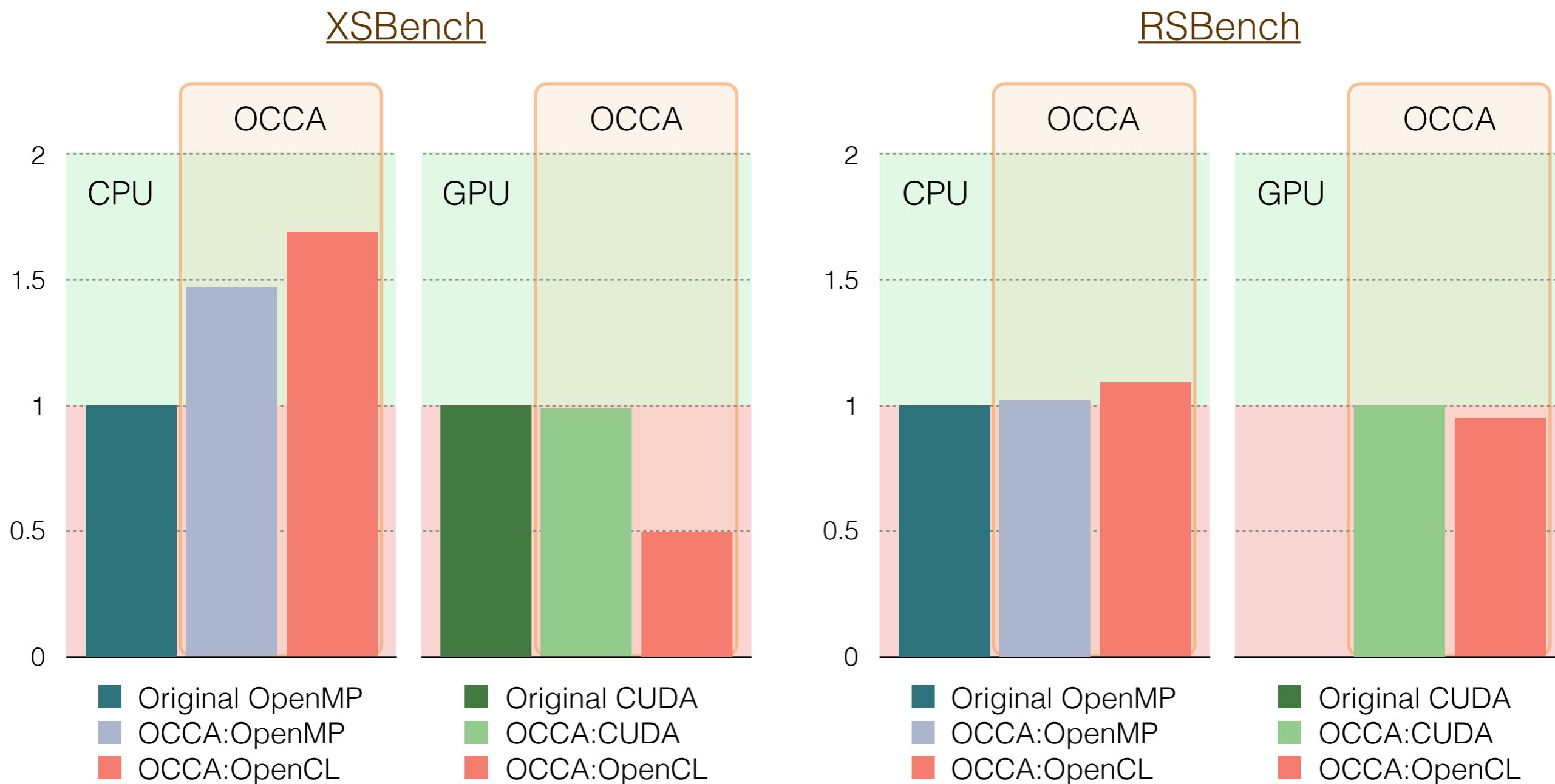
■ OpenCL on Titan
■ OpenMP on Intel i7

■ OpenCL on Tahiti



OCCA: apps & benchmarks

Monte Carlo for neutronics
Collaborations with Argonne National Lab



OpenMP : Intel Xeon CPU E5-2650
OpenCL/CUDA : NVIDIA Tesla K20c

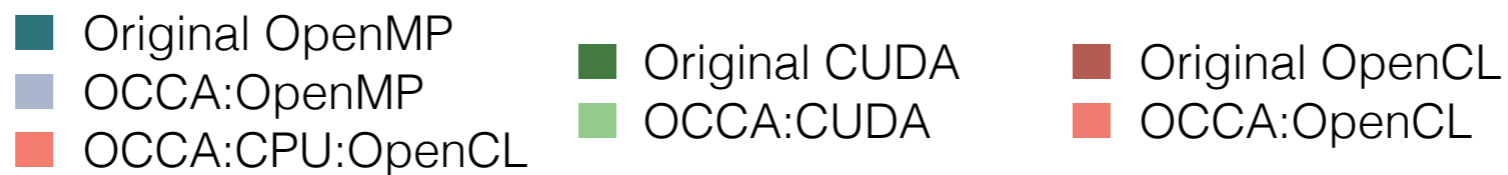
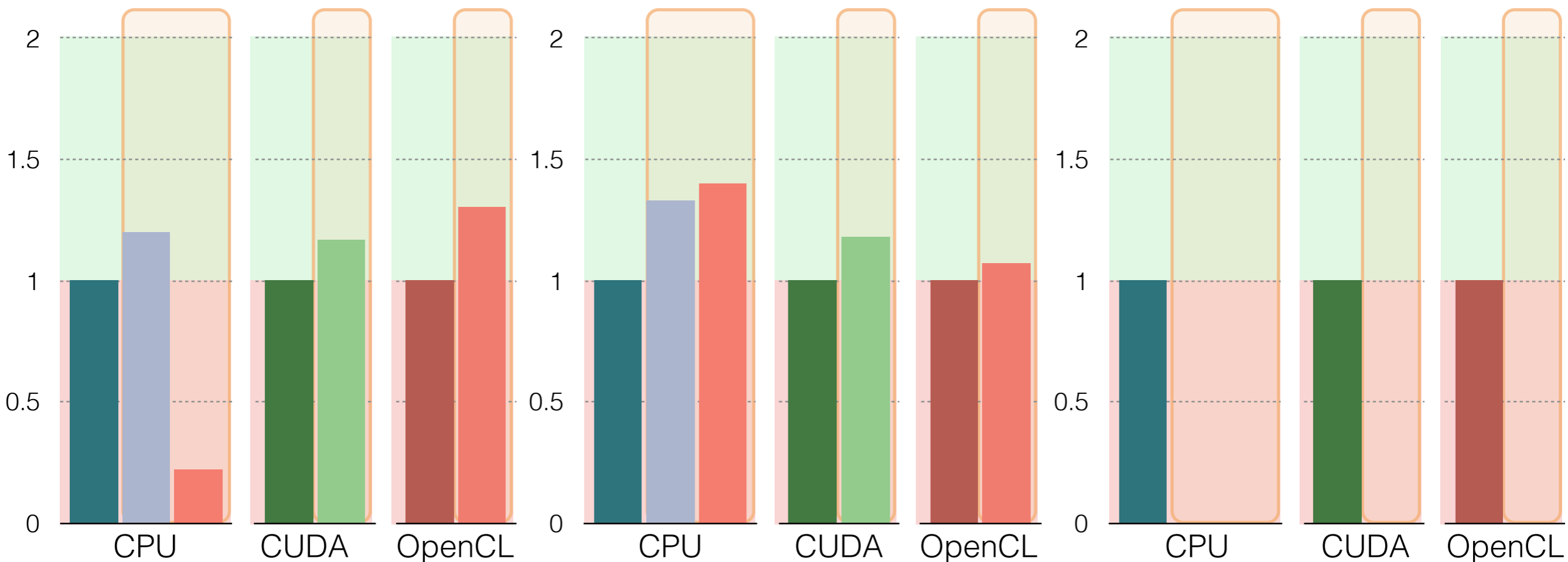
OCCA: apps & benchmarks

Three of our ported Rodinia benchmarks, based on the “11 Dwarves”

Backprop

BFS

Needleman



OpenMP : Intel Xeon CPU E5-2650
OpenCL/CUDA : NVIDIA Tesla K20c

Cloud GPUs: proceed with caution...

www.paranumal.com/single-post/2018



PARALLEL NUMERICAL ALGORITHMS RESEARCH TEAM @VT

Home Software Blog Publications Team Alums Gallery paranumal@vt.edu

Concurrent Cloud Computing: installing occaBench for V100

February 6, 2018 | Tim Warburton

Overview: This week we have been experimenting with instances on [Amazon AWS](#) and [Paperspace](#) that come equipped with [NVIDIA V100](#) GPUs. These GPUs are hot properties and not widely available, so we had to request special access to V100 equipped instances on both systems. Both AWS and Paperspace responded quickly to our requests. The Paperspace support team was also incredibly responsive, patient, and helpful getting through some minor technical issues.

Note: this article is not an endorsement of these companies or their products, we are just providing an insight into our experience getting started on their systems. Your mileage may vary. In our experience both systems were very similar once the instances were provisioned.

Configuration: On AWS we set up a p3.2xlarge instance and on Paperspace we set up a V100 machine. In both cases we chose Ubuntu 16.04, for no other reason than familiarity with Ubuntu/Linux.

Our Recent Posts

-  libParanumal: Galerkin-Boltzmann 3D flow simulation
July 5, 2018
-  libParanumal: Galerkin-Boltzmann flow simulation
June 28, 2018
-  Undergraduate Summer Researchers Join the Paranumal Team

For my own development needs I have switched to use GPU cloud servers including: gpueater (AMD GPUs), Amazon AWS (NVIDIA GPUs incl. V100), paperspace (NVIDIA GPUs incl. V100).



tim.warburton@vt.edu