08/03/2017

# 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

# Overview

Background on instructor.

Part 0: GPU Myths.

Part 1: NVIDIA Graphical Processing Unit

Part 2: The Compute Unified Device Architecture (CUDA)

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

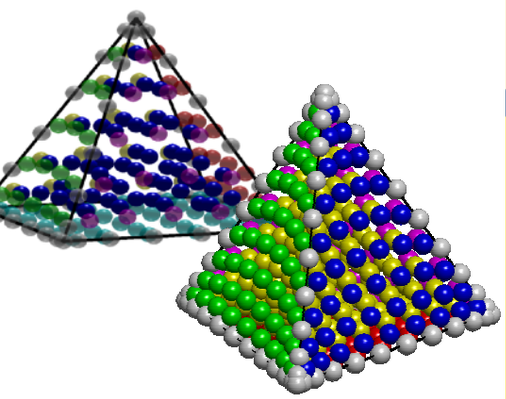
Part 3: Interlude on CUDA optimization.

Part 4: Portable programming models:

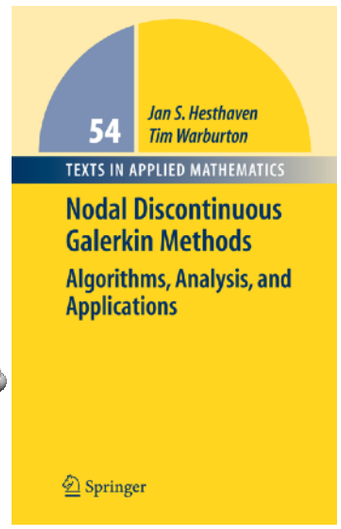
- The Open Computing Language (OpenCL)
- The Open Concurrent Computing Abstraction (OCCA)
- Hands on:
    - Task: flow simulations movie.
    - Prep: find a png image with white background.

# My Research…

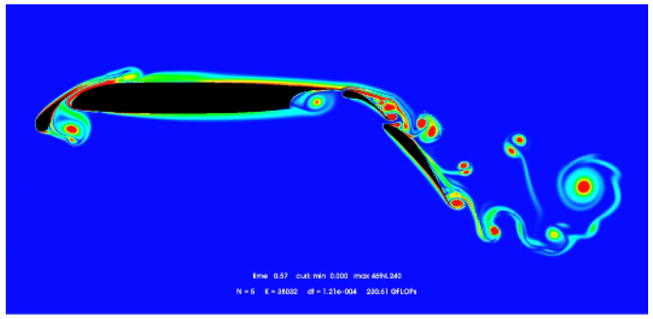
Goal: fast, scalable, flexible & accurate numerical PDE solvers adapted for modern many-core architectures.



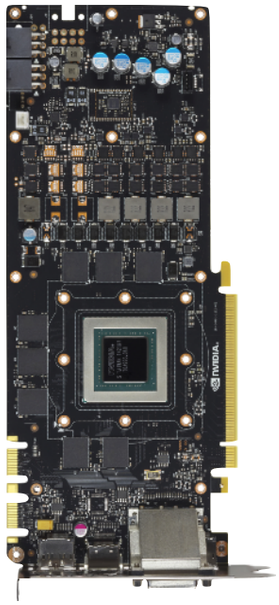
Approximation Theory




Numerical Analysis



Numerical methods & Physical PDE Modeling



Accelerated Computing

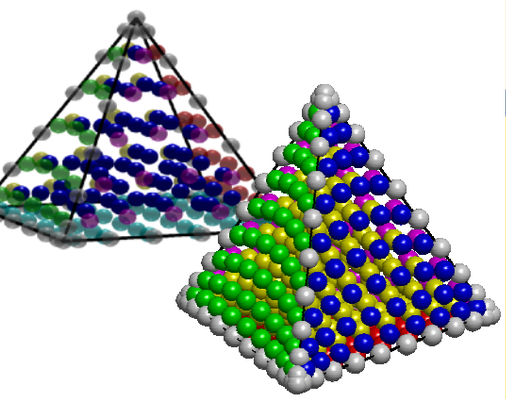
High Performance Scalability

Basic science → Application → Industrial Scale

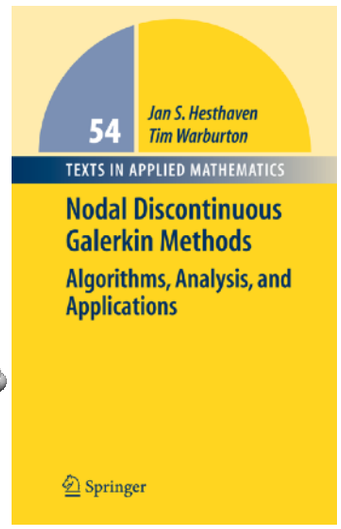
*High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.*
*GPU programming tools & applications. Industrial collaboration.*

# My Research…

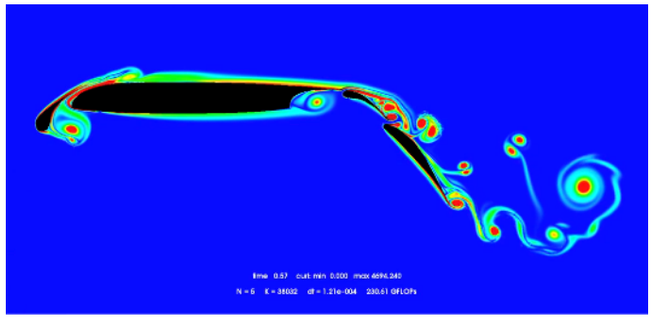
Goal: fast, scalable, flexible & accurate numerical PDE solvers adapted for modern many-core architectures.



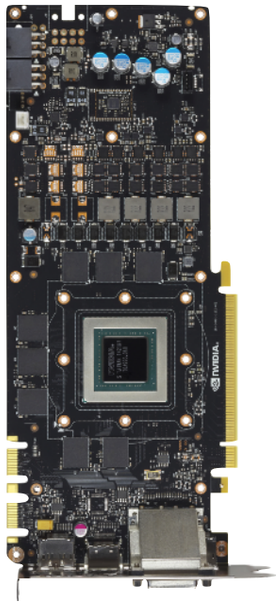
Approximation Theory




Numerical Analysis



Numerical methods & Physical PDE Modeling



Accelerated Computing

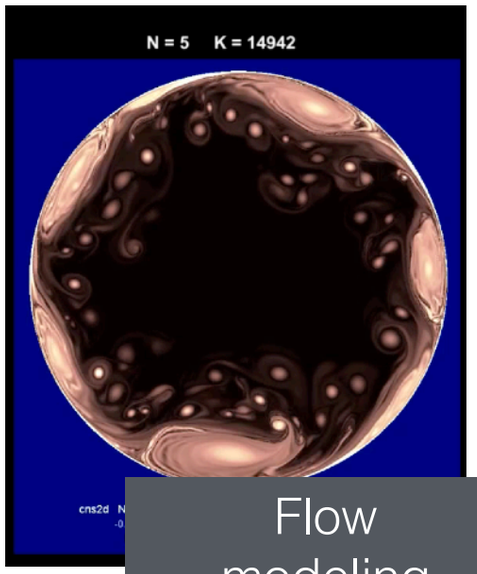
High Performance Scalability

Basic science → Application → Industrial Scale

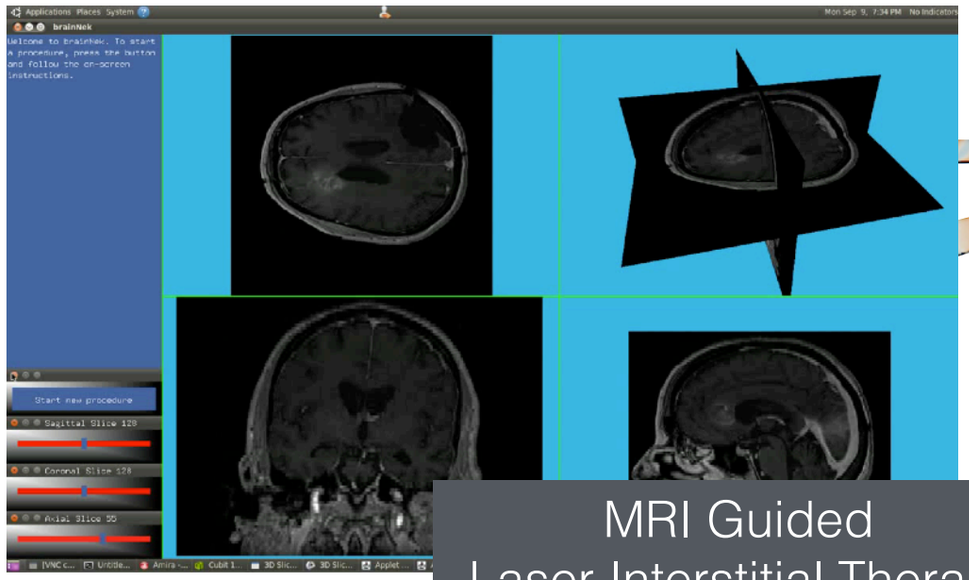
*High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.*
*GPU programming tools & applications. Industrial collaboration.*

# Some GPU Accelerated Apps…

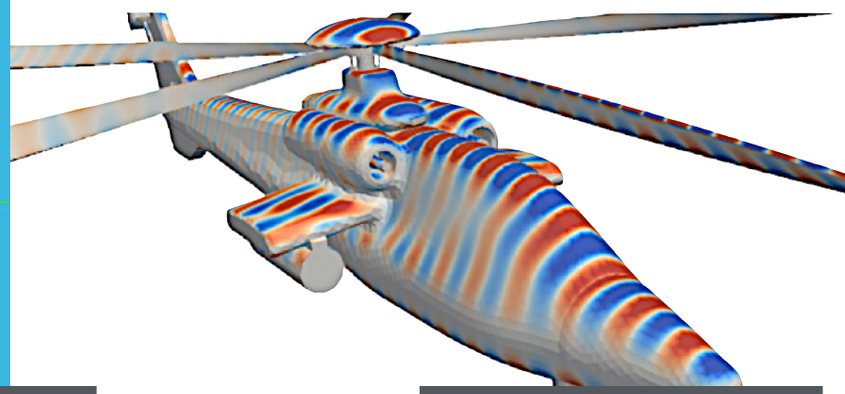
We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy…



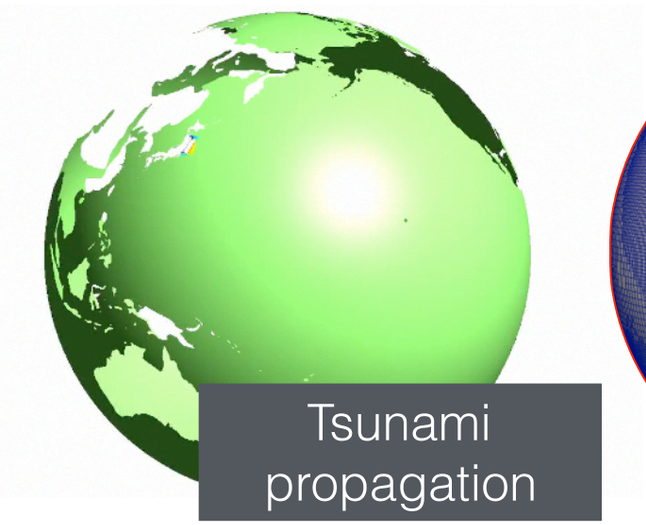

Flow modeling



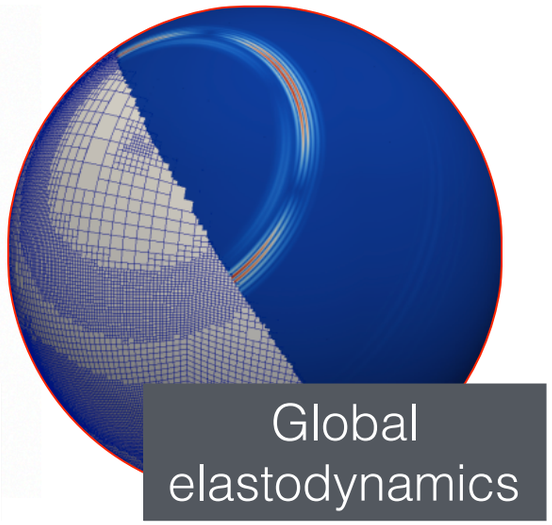
MRI Guided Laser Interstitial Therapy



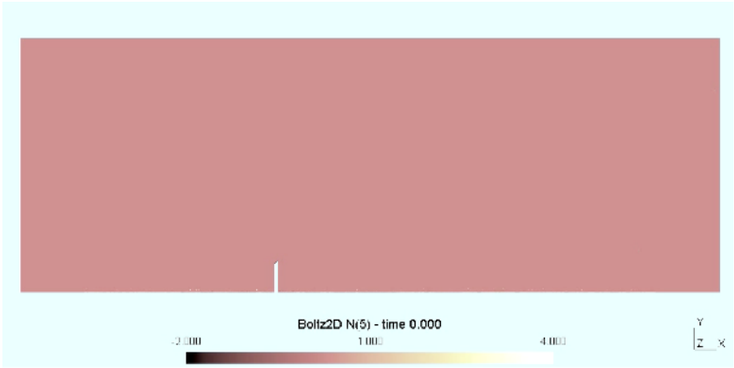
Electromagnetic scattering



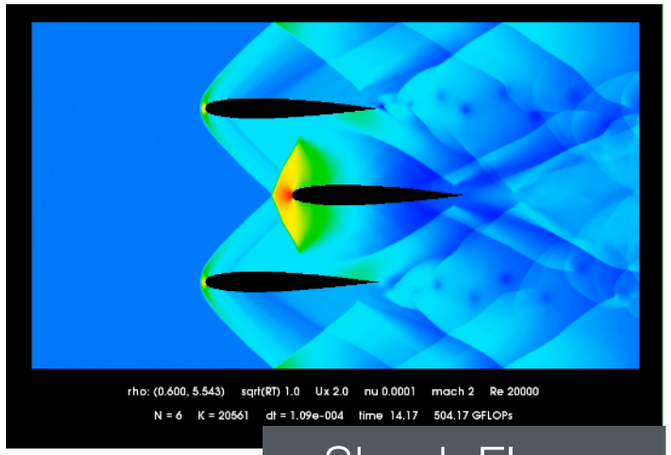
Tsunami propagation



Global elastodynamics




Kinetic Models

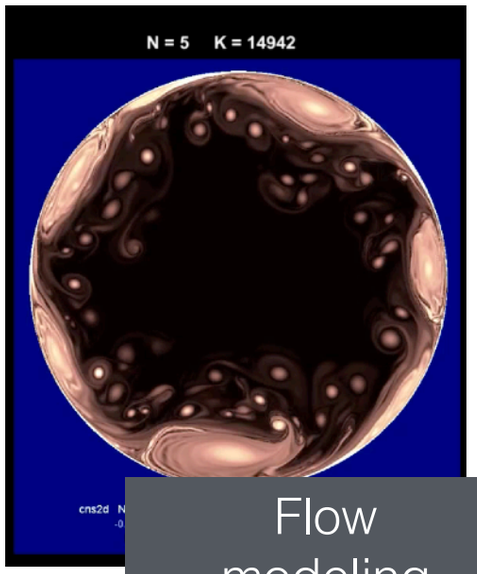



Shock Flows

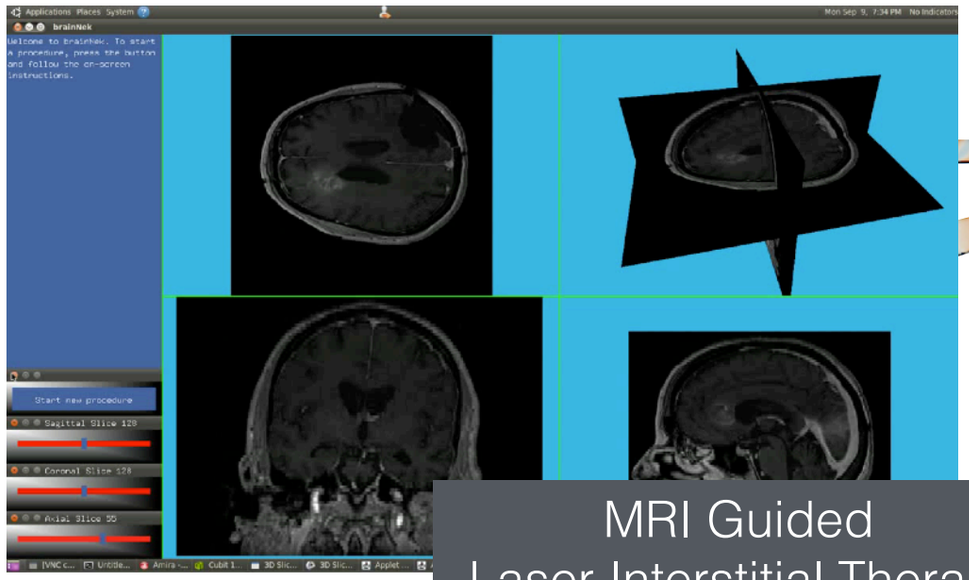
*High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.*
*GPU programming tools & applications. Industrial collaboration.*

# Some GPU Accelerated Apps…

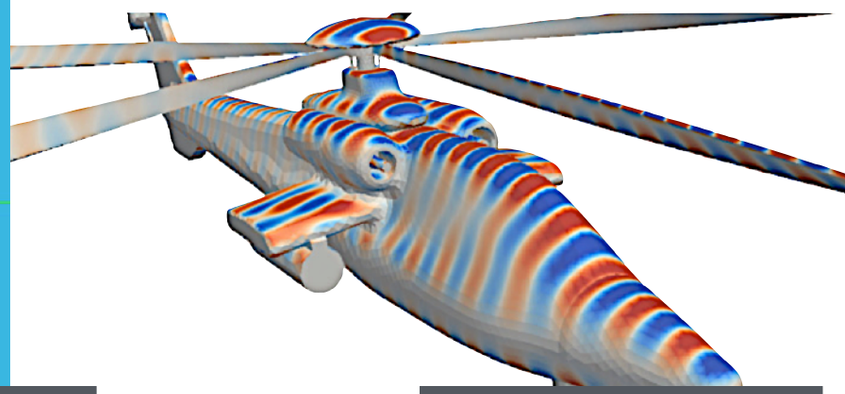
We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy…



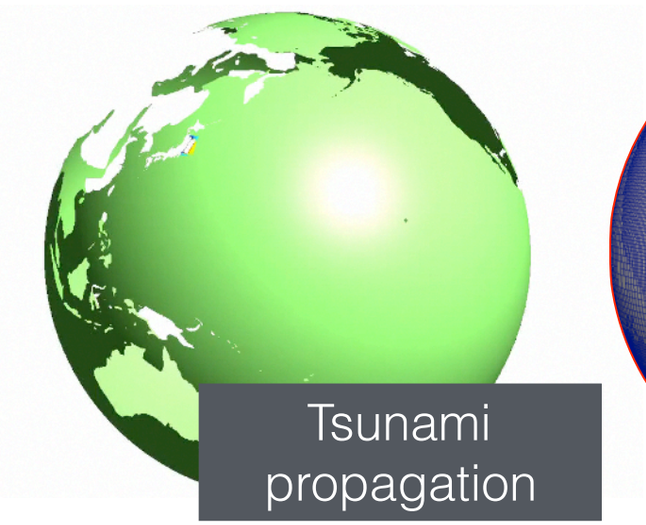

Flow modeling



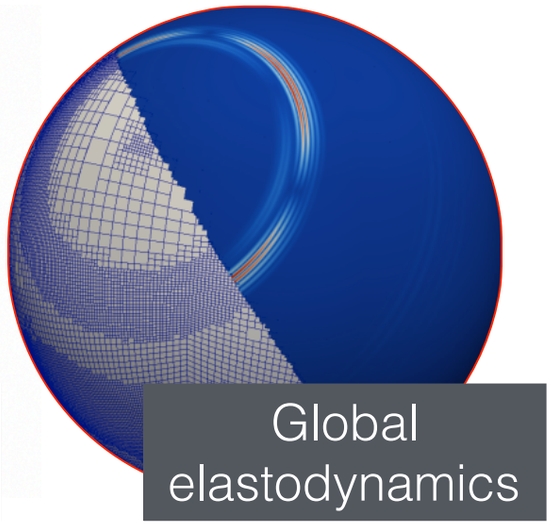
MRI Guided Laser Interstitial Therapy



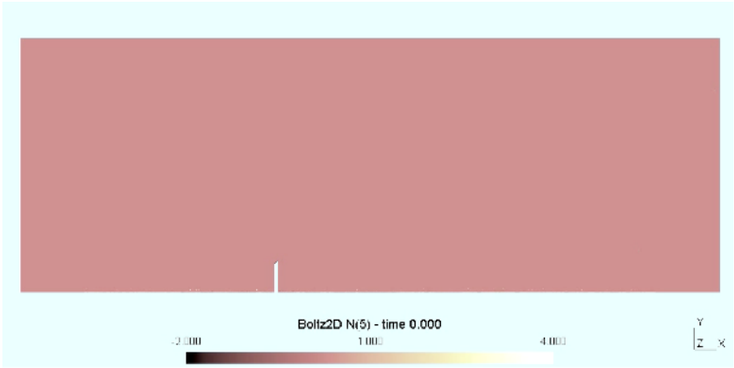
Electromagnetic scattering



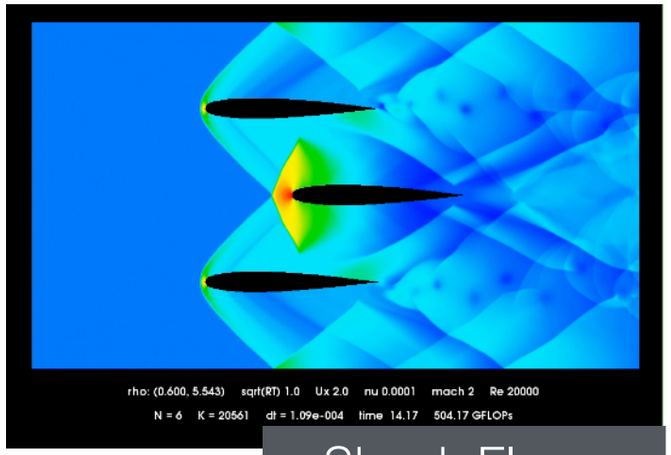
Tsunami propagation



Global elastodynamics




Kinetic Models

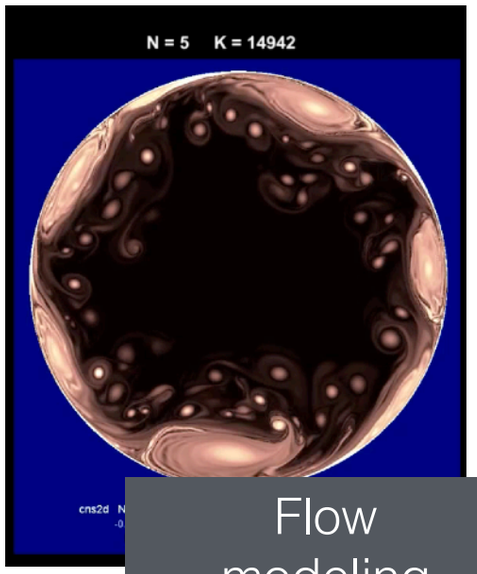



Shock Flows

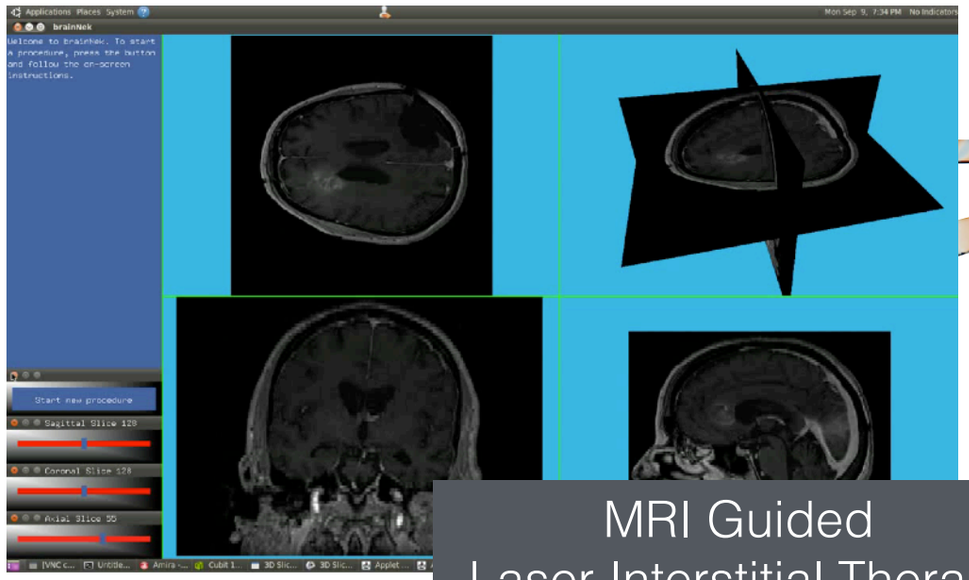
*High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.*
*GPU programming tools & applications. Industrial collaboration.*

# Some GPU Accelerated Apps…

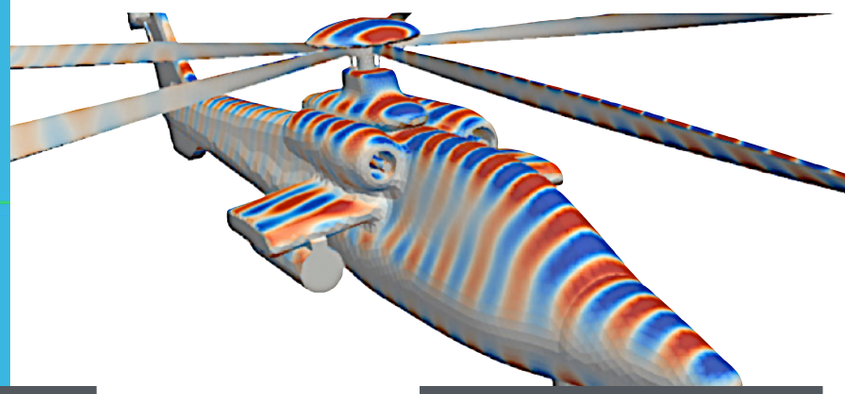
We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy…



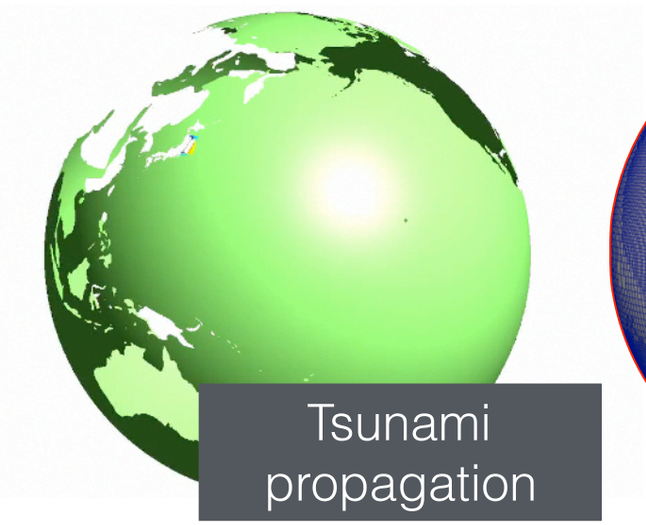

Flow modeling



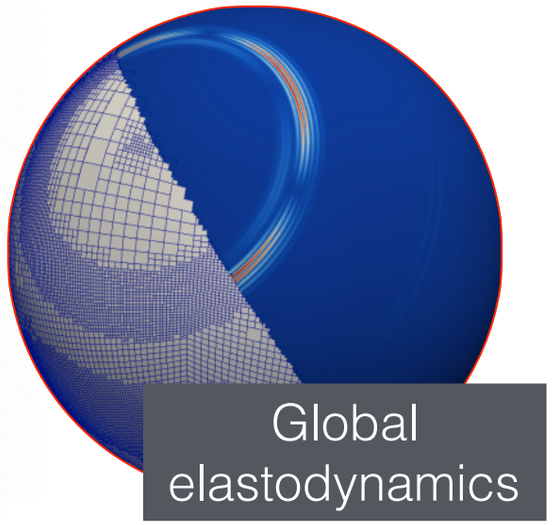
MRI Guided Laser Interstitial Therapy



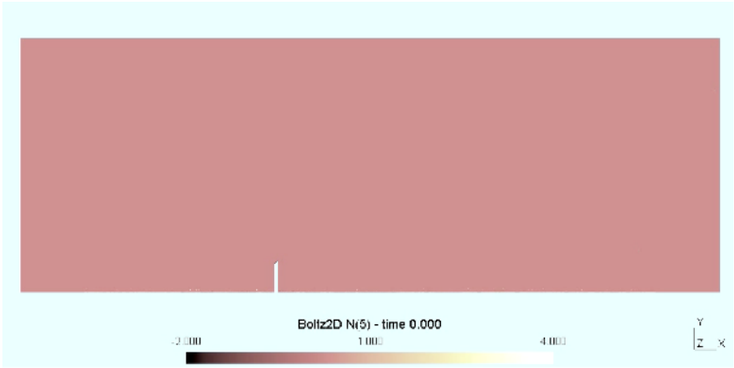
Electromagnetic scattering



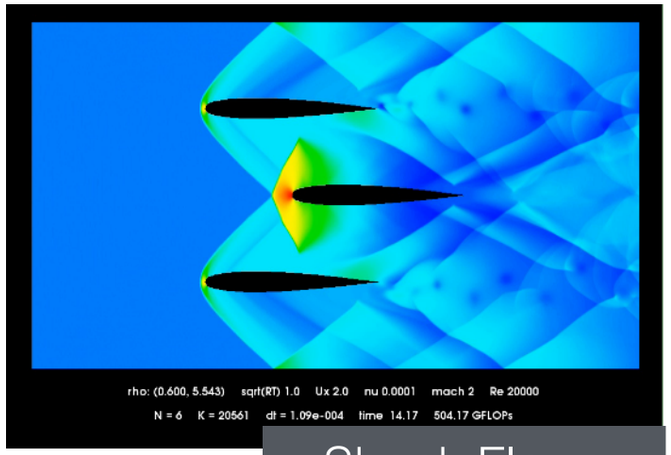
Tsunami propagation



Global elastodynamics




Kinetic Models

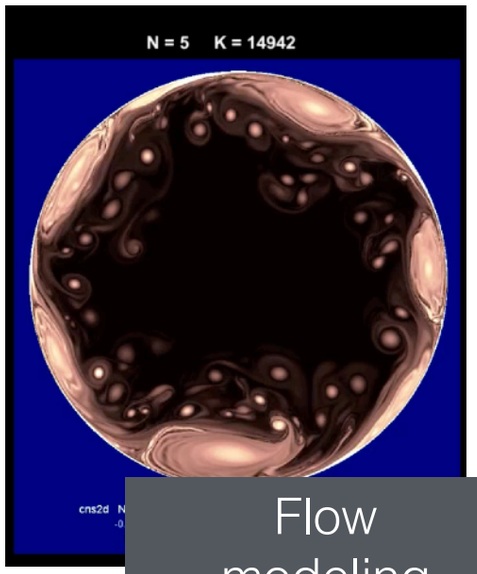



Shock Flows

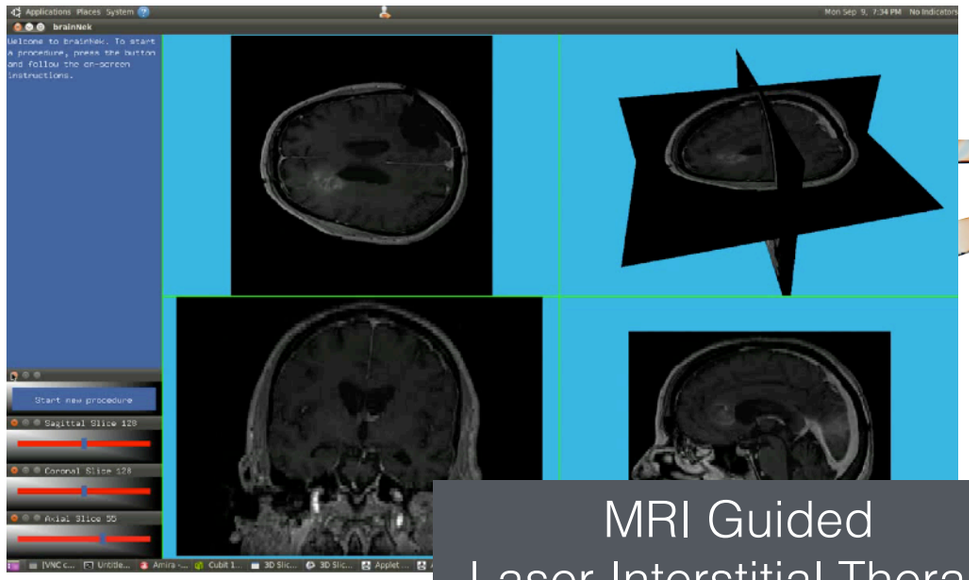
*High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.*
*GPU programming tools & applications. Industrial collaboration.*

# Some GPU Accelerated Apps…

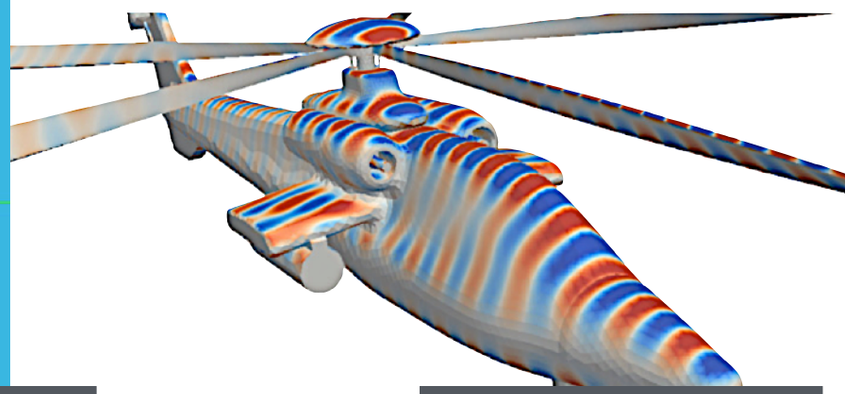
We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy…



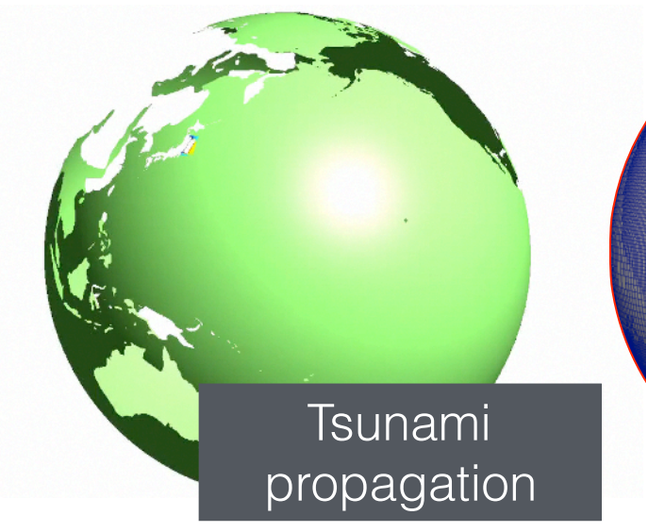

Flow modeling



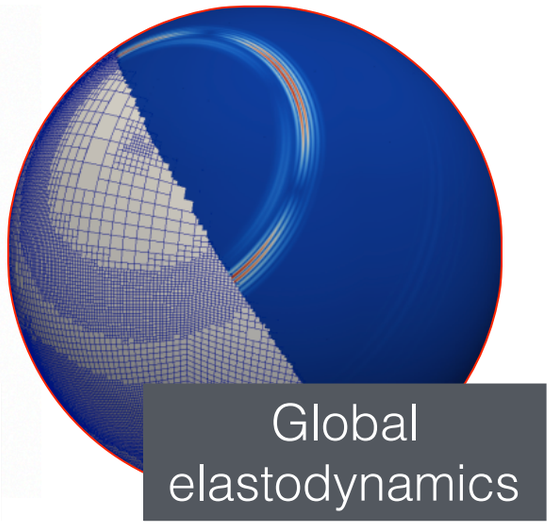
MRI Guided Laser Interstitial Therapy



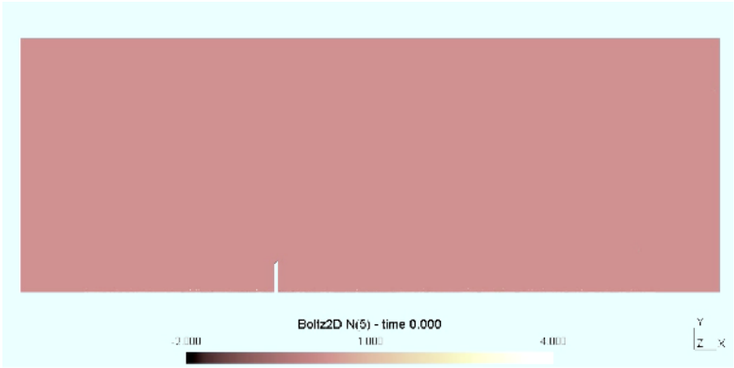
Electromagnetic scattering



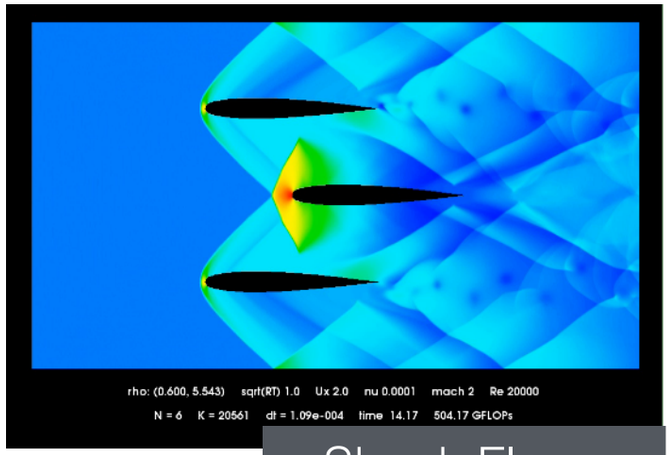
Tsunami propagation



Global elastodynamics




Kinetic Models




Shock Flows

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

# Accelerators: slides & repos

Slides:

www.math.vt.edu/people/tcew/ATPESC17

Examples:

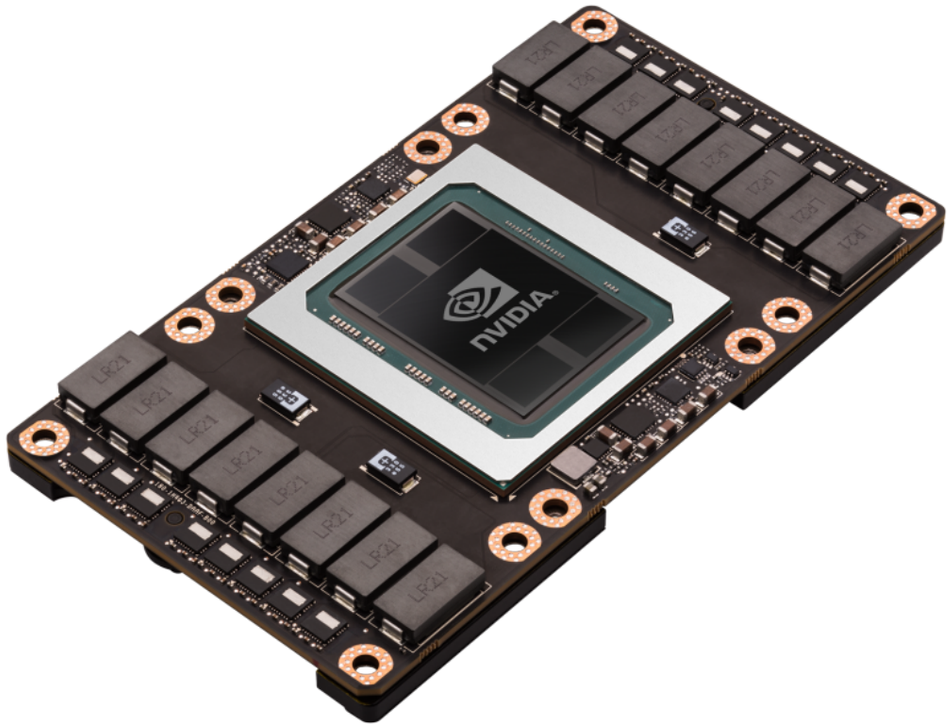
git clone https://github.com/tcew/ATPESC17

OCCA repo:

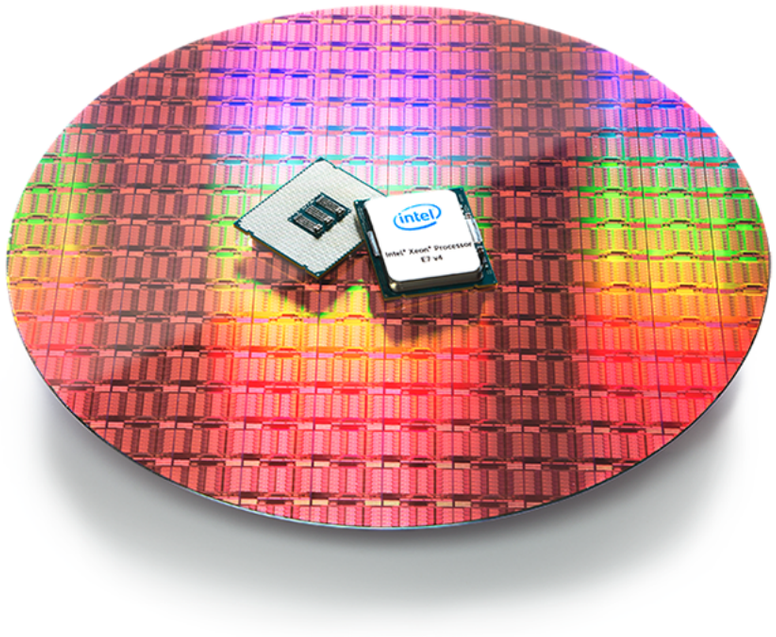
git clone https://github.com/libocca/occa

# Part 0: GPU Reality Check

# Myth #1: GPU 100x faster than CPU




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

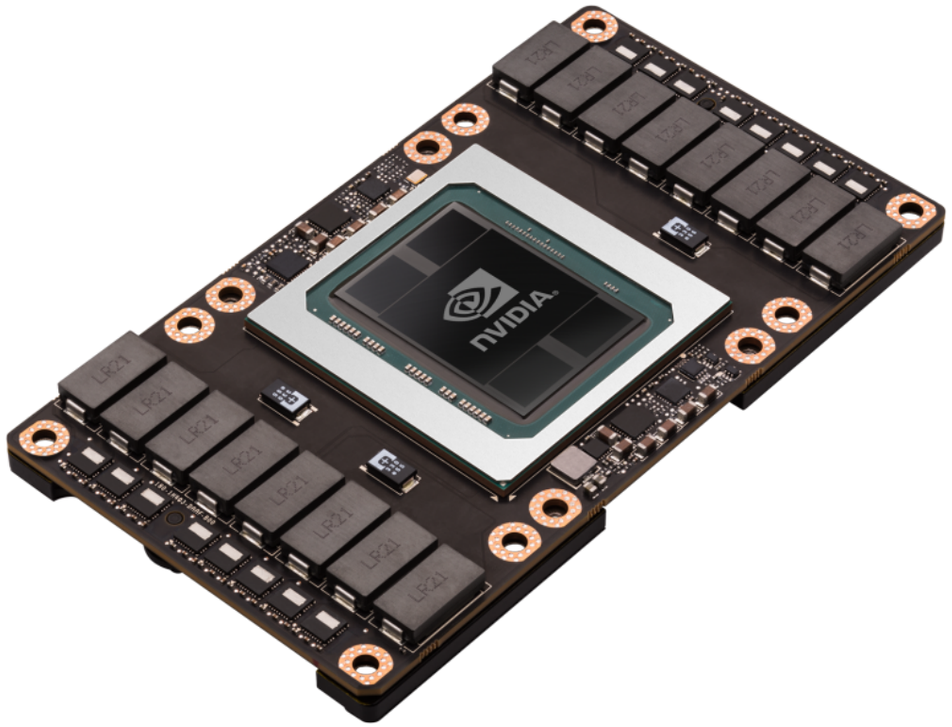



Intel E7-8894 v4: 4 memory channels
up to 85 GB/s per socket

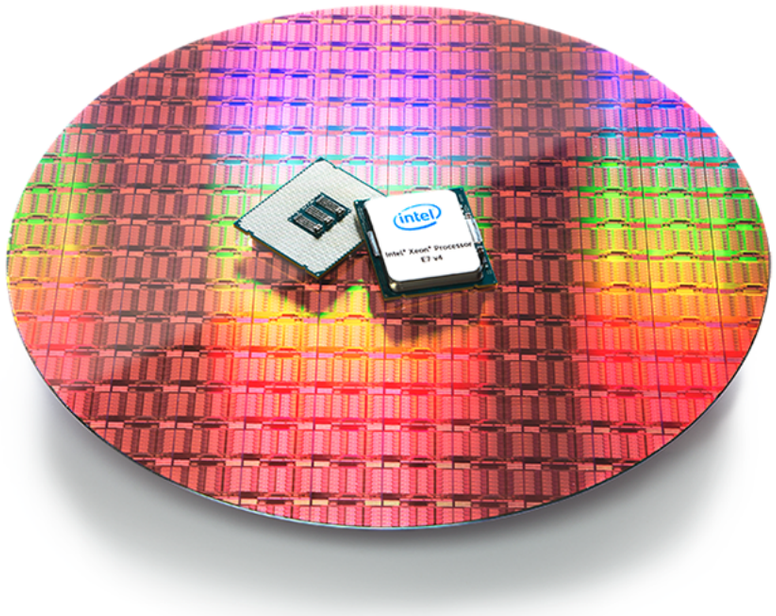
http://www.nvidia.com/object/tesla-p100.html
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz

*Majority of HPC codes are memory and network bound.*

# Myth #2: GPUs are expensive




NVIDIA P100: ~$5K

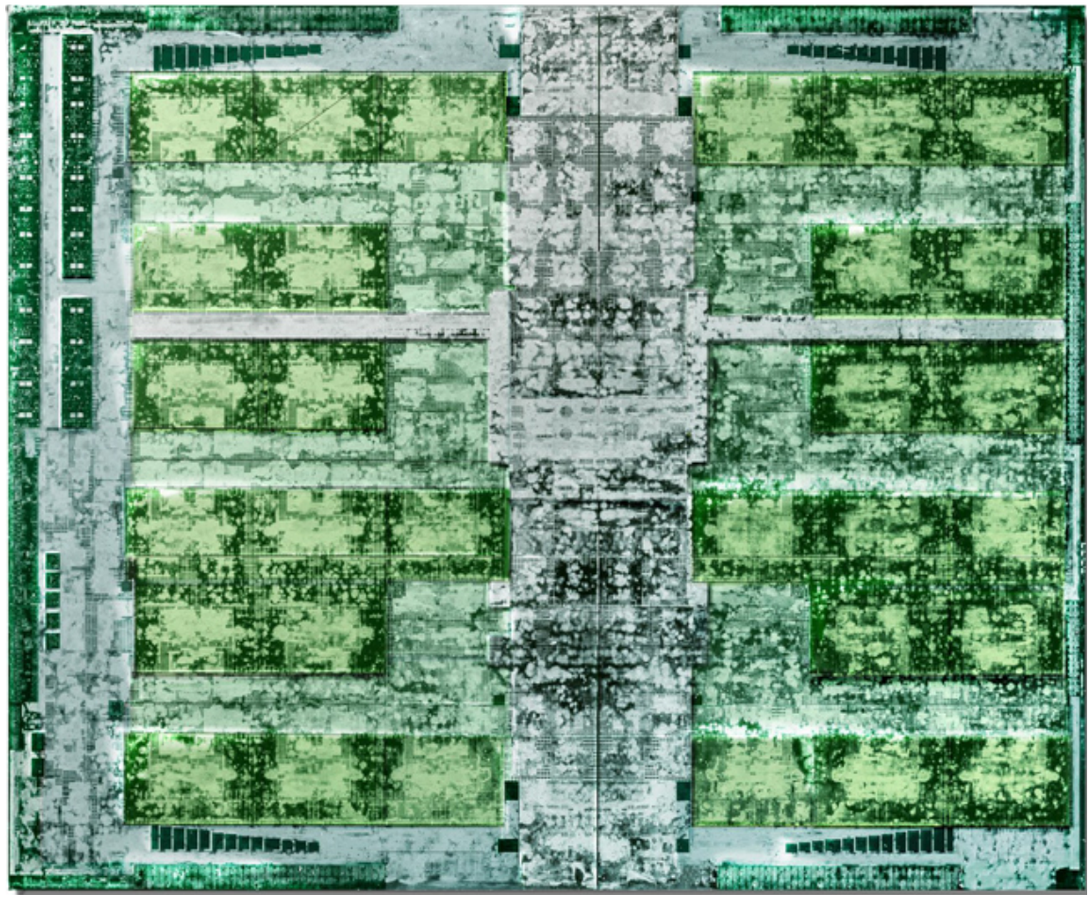



Intel E7-8894 v4: ~$9K

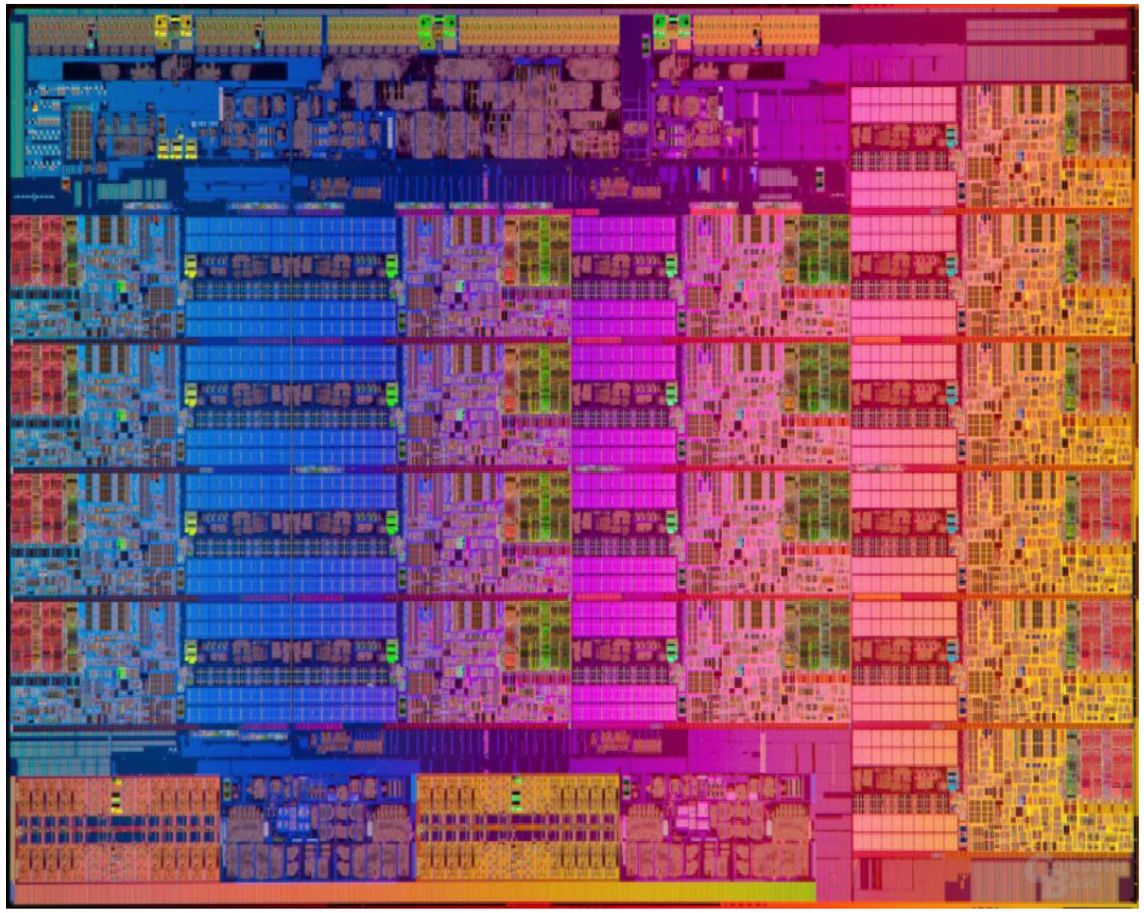
*Price estimates from http://www.thinkmate.com/*
*http://www.anandtech.com/show/11121/intel-xeon-e7-8894-v4-cpu-24c-48t-9000-usd*

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

# Myth 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

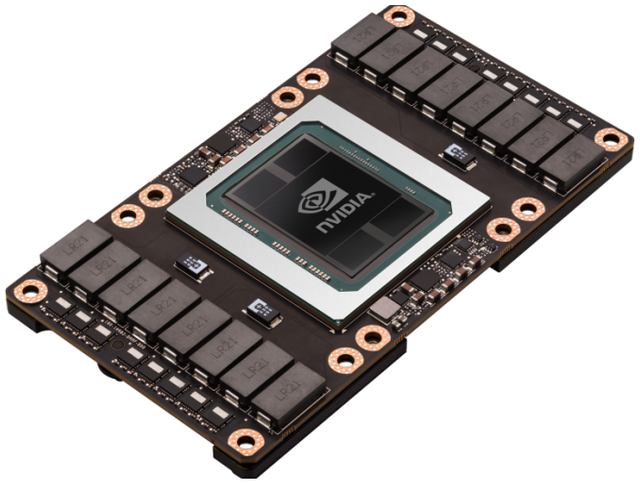
http://www.nvidia.com/object/tesla-p100.html
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz

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

# Myth #4.0: OpenACC is magic

OpenACC
Directives for Accelerators

+



≠

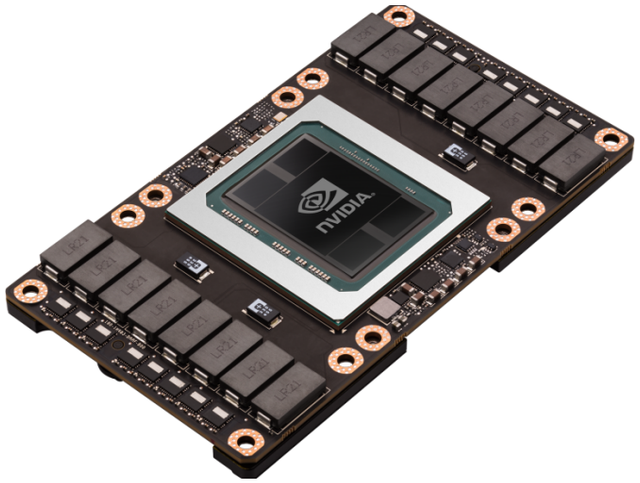
http://icons.iconarchive.com/icons/hadezign/hobbies/256/Magic-icon.png

*"good" OpenACC codes are quite often derived from good CUDA implementations.*

# Myth #4.1: CUDA is magic

NVIDIA®
CUDA®

+



≠

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…

*21 hours of fun: https://developer.nvidia.com/udacity-cs344-intro-parallel-programming*

# Reality Check

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

*21 hours of fun: https://developer.nvidia.com/udacity-cs344-intro-parallel-programming*

# Reality Check

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… there are many web resources …

*21 hours of fun: https://developer.nvidia.com/udacity-cs344-intro-parallel-programming*

# Reality Check

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… there are many web resources …

… and nothing beats hands on.

*21 hours of fun: https://developer.nvidia.com/udacity-cs344-intro-parallel-programming*

# Reality Check

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… add some background to Nikolay Sakharnykh's talk & hands on …

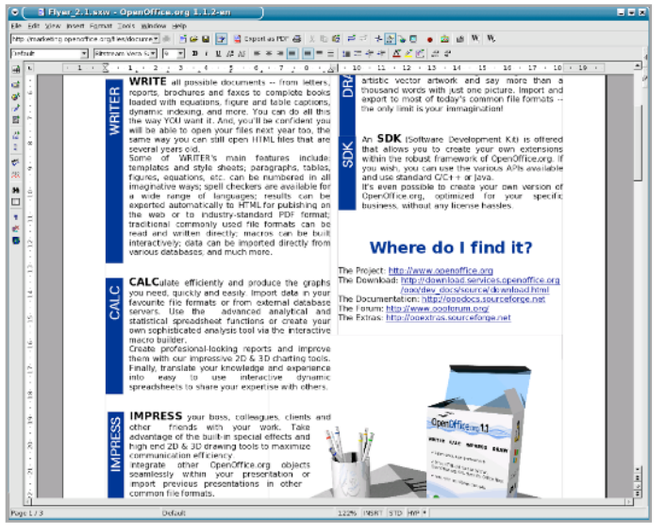
… there are many web resources …

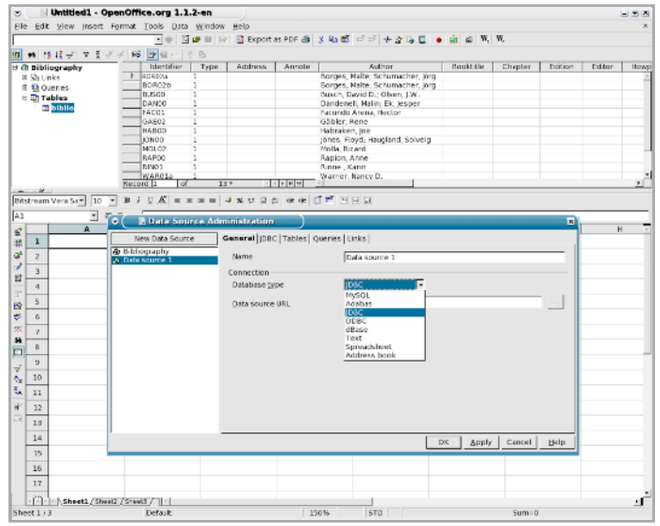
… and nothing beats hands on.

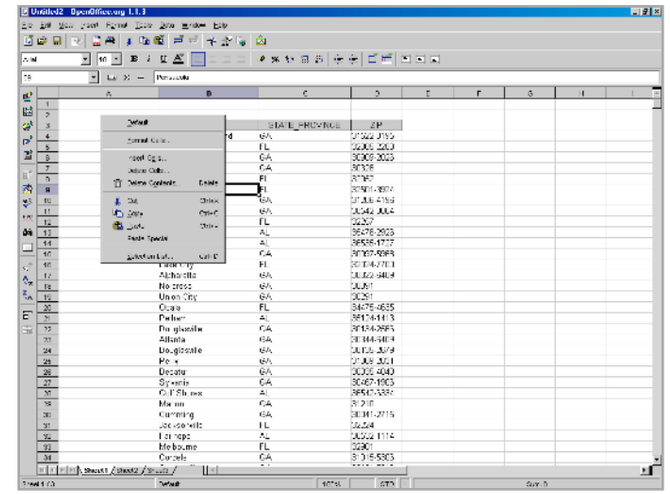
*21 hours of fun: https://developer.nvidia.com/udacity-cs344-intro-parallel-programming*

# Part 1: From CPU to GPU

# CPU: architecture follows purpose



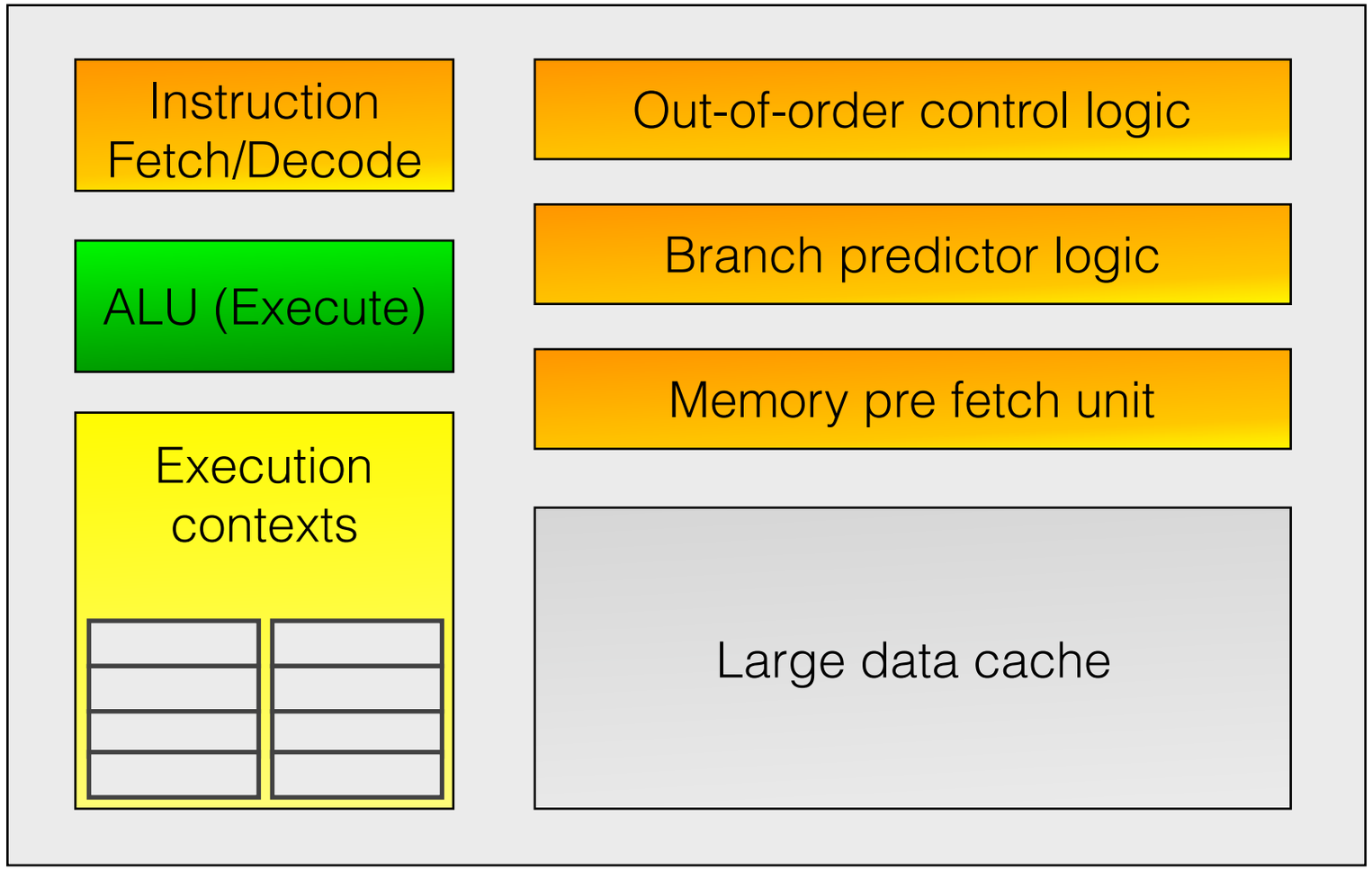



Original design goals for CPUs:

- 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

The main purpose of graphics processing units is to project textured polygons onto the screen in a fiercely competitive consumer-facing industry.

This is an embarrassingly parallel process and specialized MPP chips have been created by ATi (now AMD), Intel, NVIDIA et al to perform floating point intensive operations to render scenes in realtime.

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

# GPU: massively parallel processing

The main purpose of graphics processing units is to project textured polygons onto the screen in a fiercely competitive consumer-facing industry.

This is an embarrassingly parallel process and specialized MPP chips have been created by ATi (now AMD), Intel, NVIDIA et al to perform floating point intensive operations to render scenes in realtime.

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

# GPU: massively parallel processing

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

# GPU: massively parallel compute

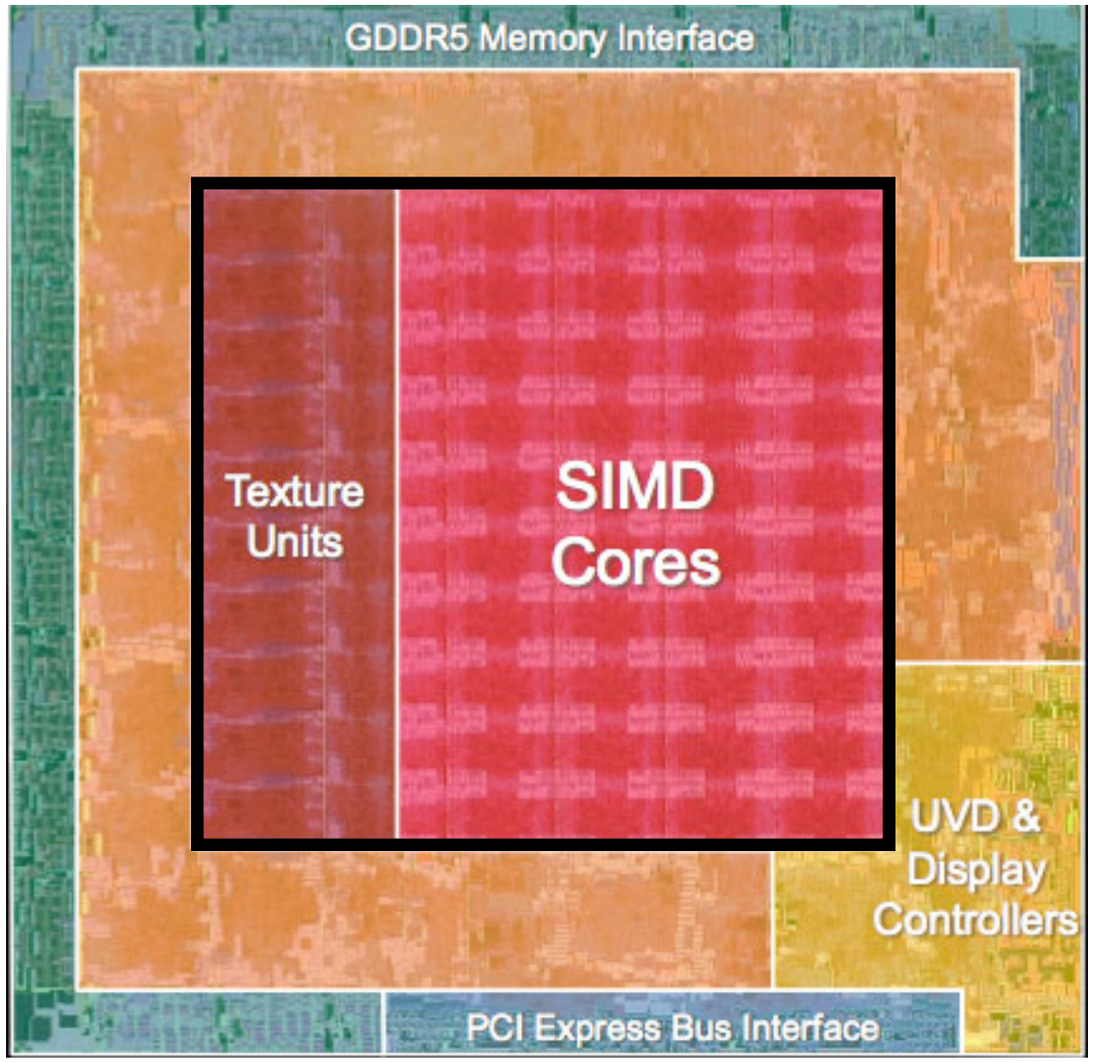
Design goals for GPUs:

- 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

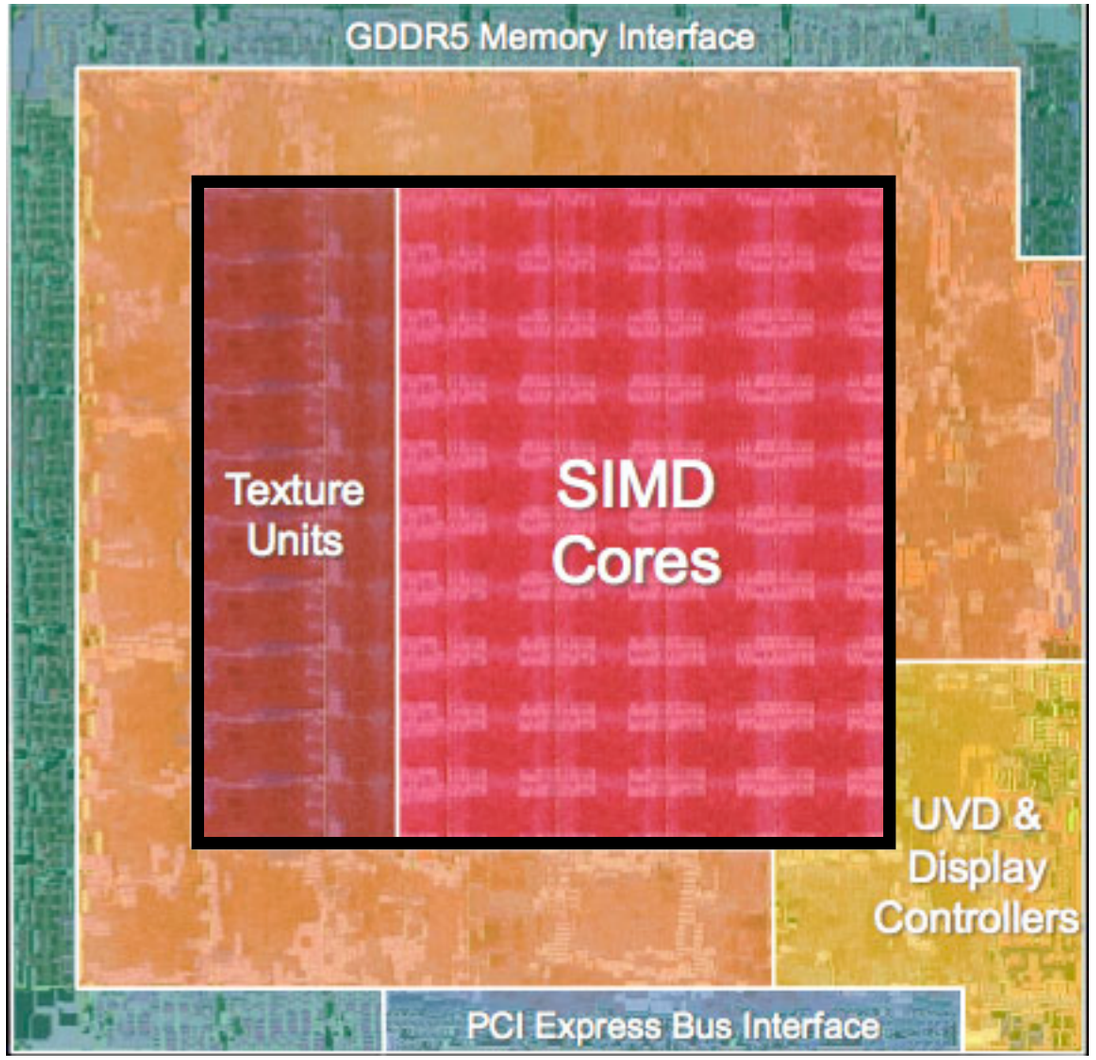
Die floorplan: AMD RV770 (2008) 55 nm, 800 SP simultaneous ops
The majority of the silicon is devoted to computation

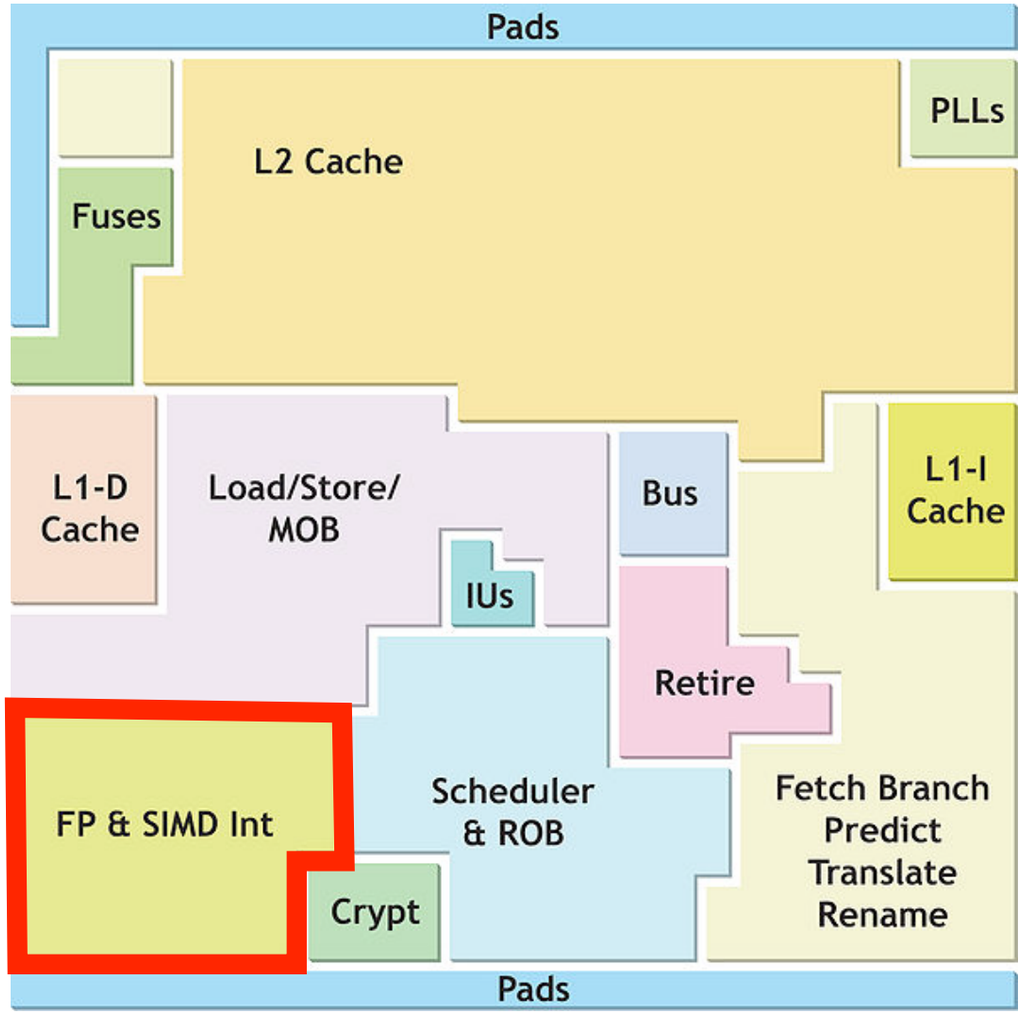



http://www.anandtech.com/show/2556/8

# GPU: early example

Comparison of block diagram of vintage GPU and CPU

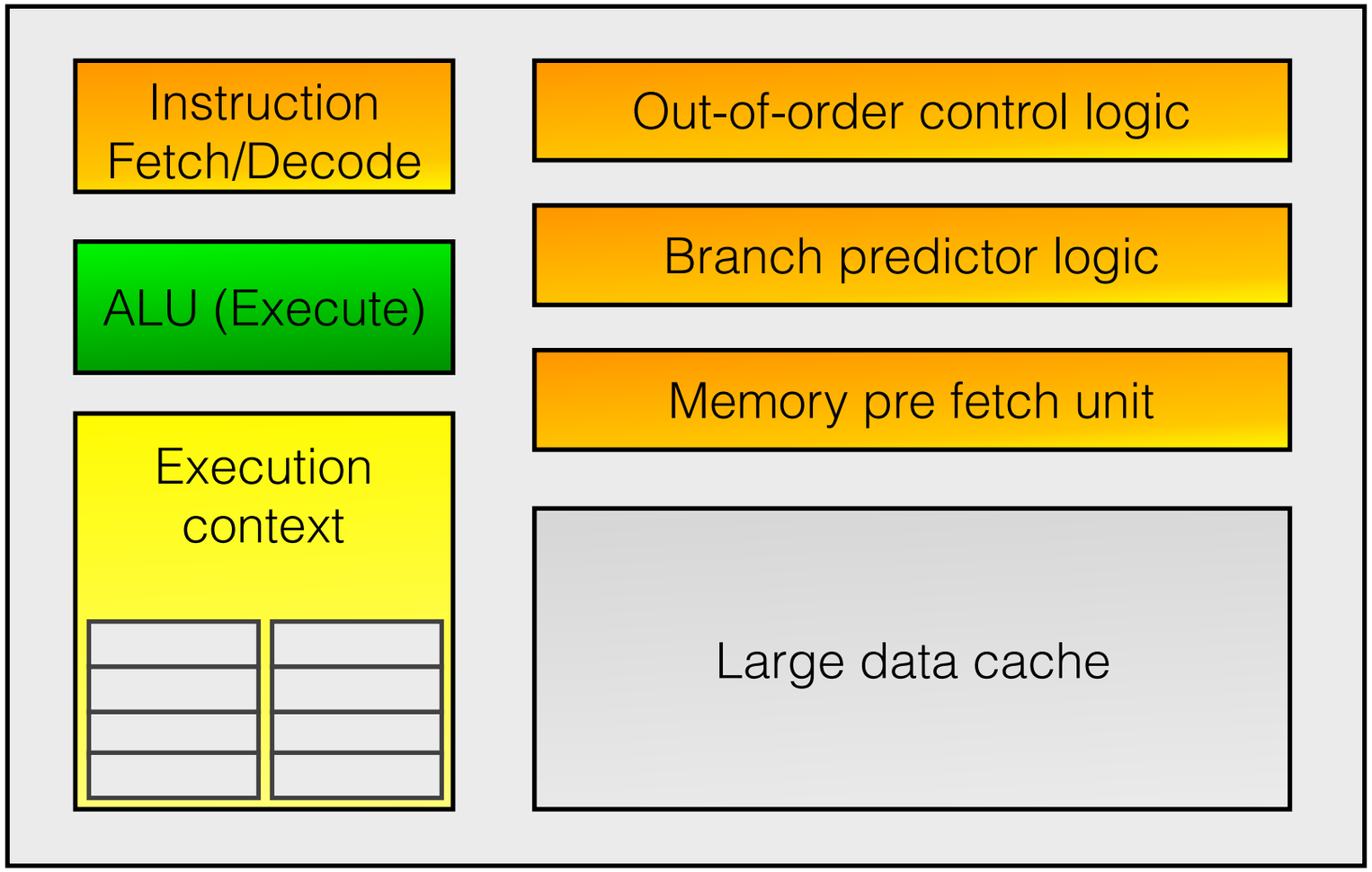



*http://www.anandtech.com/show/2556/8*

# GPU: slim down the core

The first big idea that differentiates GPU and CPU core design:
slim down the footprint of each core.

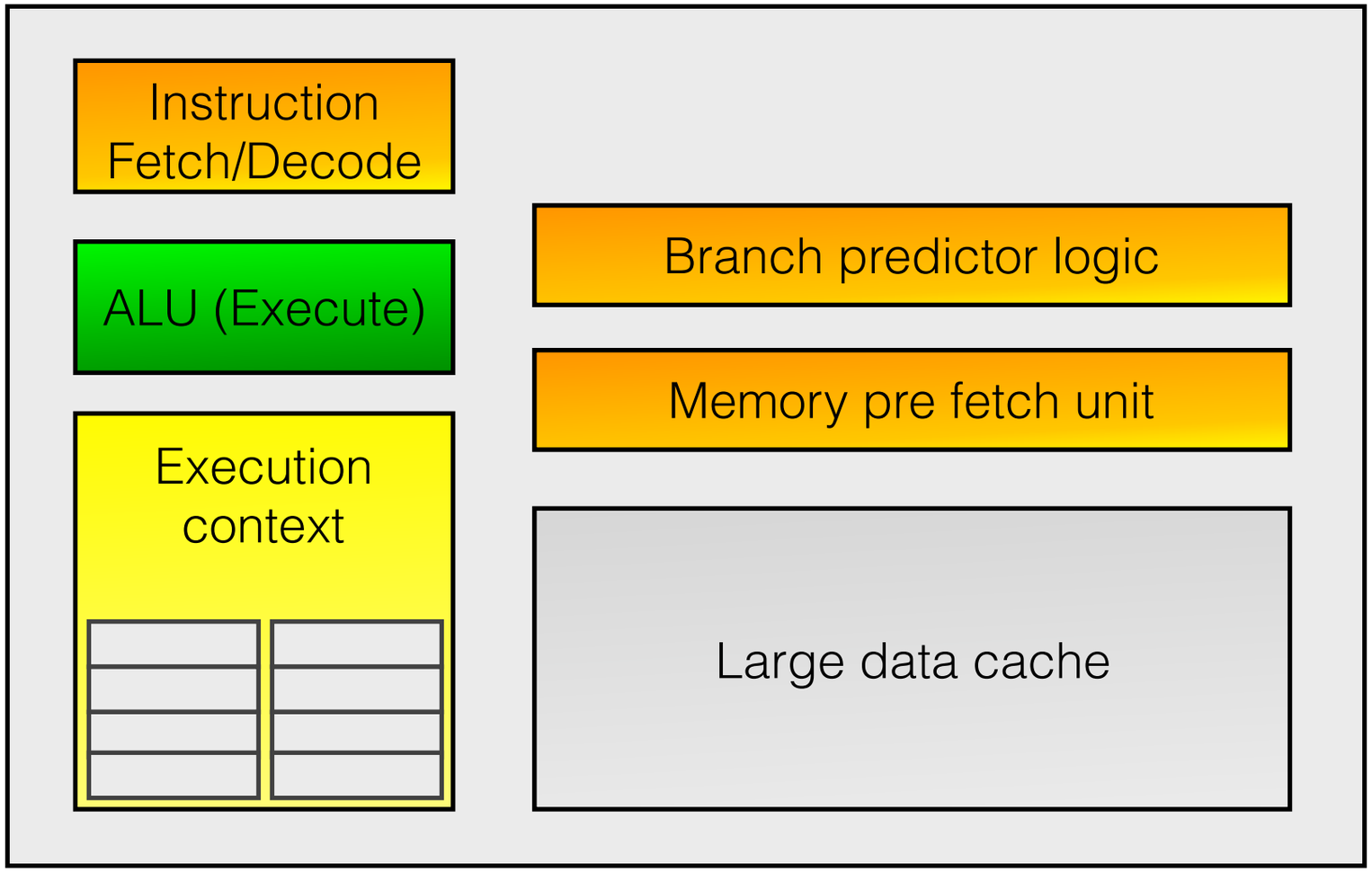



*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

*The result is a core that will stall due to code branching and memory fetches.*

# GPU: slim down the core

The first big idea that differentiates GPU and CPU core design:
slim down the footprint of each core.

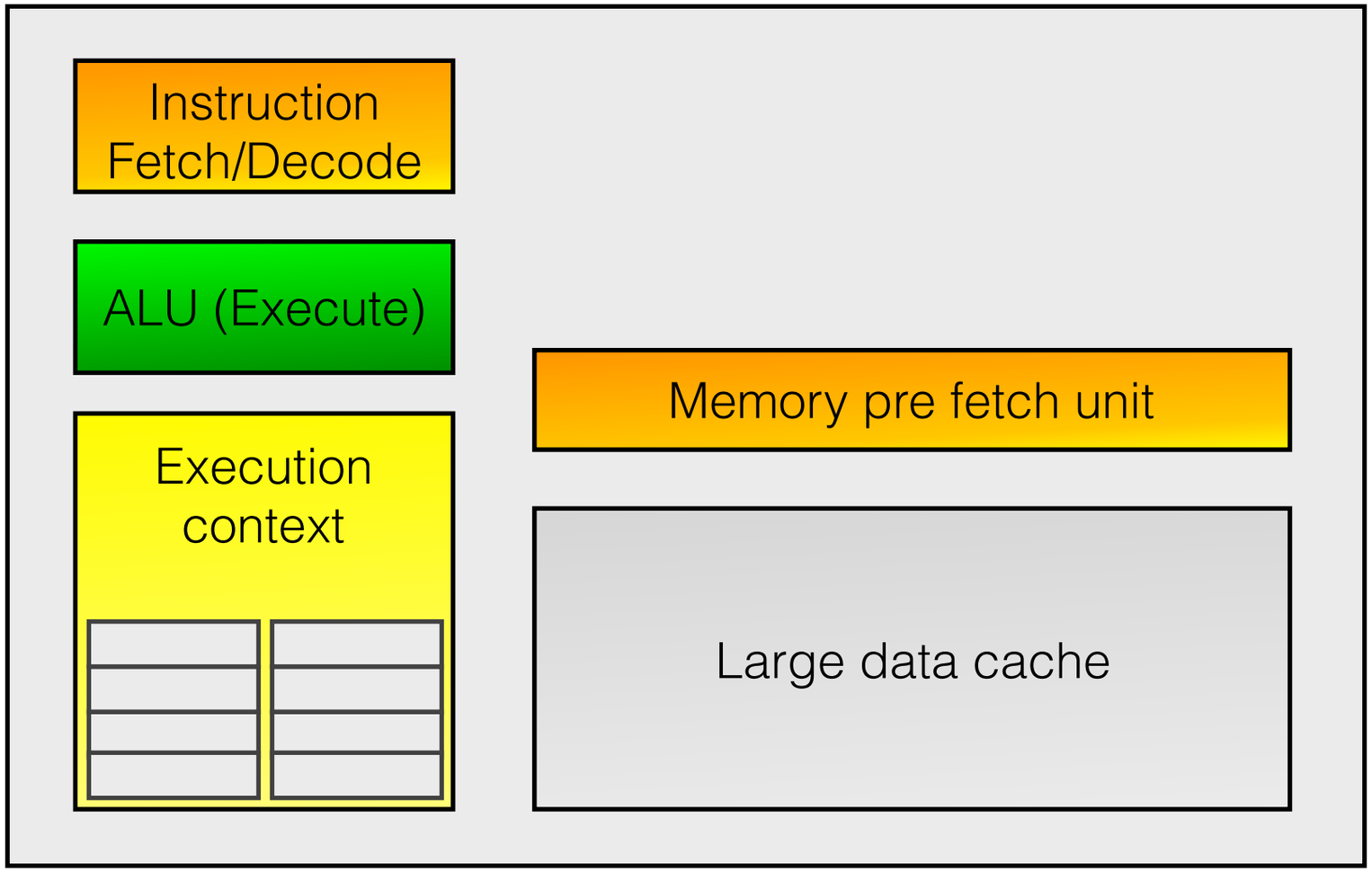



*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

*The result is a core that will stall due to code branching and memory fetches.*

# GPU: slim down the core

The first big idea that differentiates GPU and CPU core design:
slim down the footprint of each core.

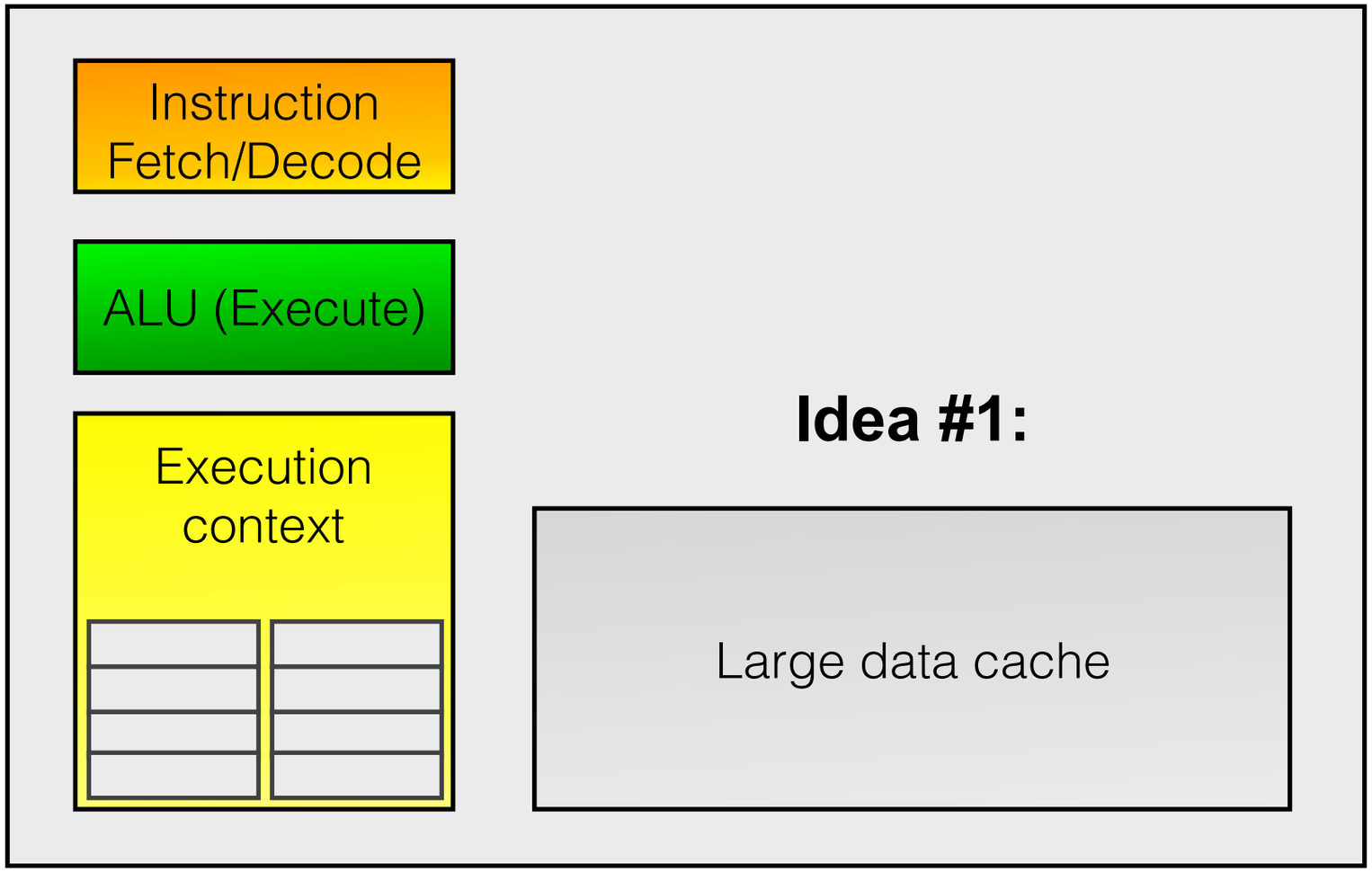



*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

*The result is a core that will stall due to code branching and memory fetches.*

# GPU: slim down the core

The first big idea that differentiates GPU and CPU core design:
slim down the footprint of each core.

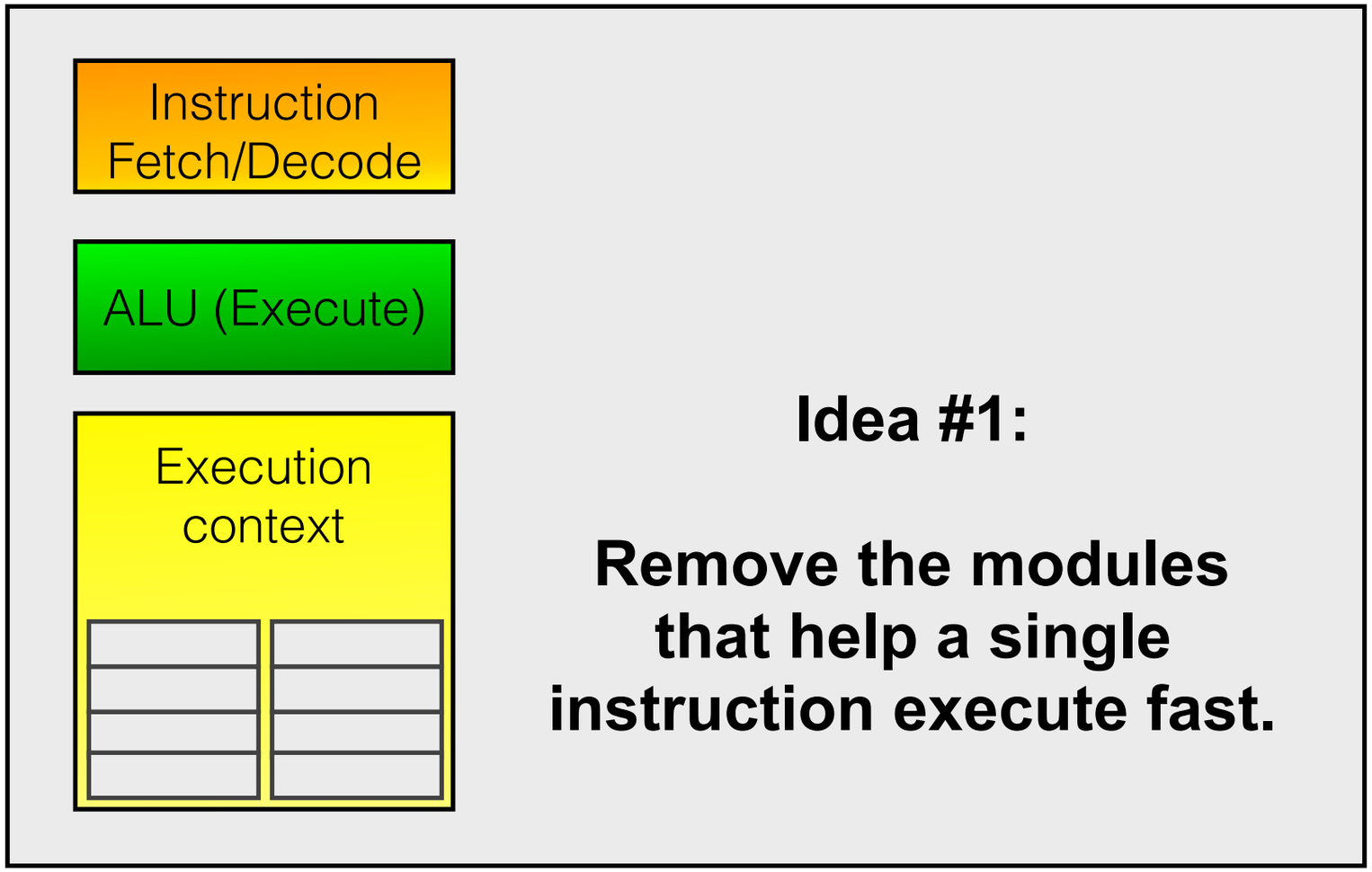



*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

*The result is a core that will stall due to code branching and memory fetches.*

# GPU: slim down the core

The first big idea that differentiates GPU and CPU core design:
slim down the footprint of each core.

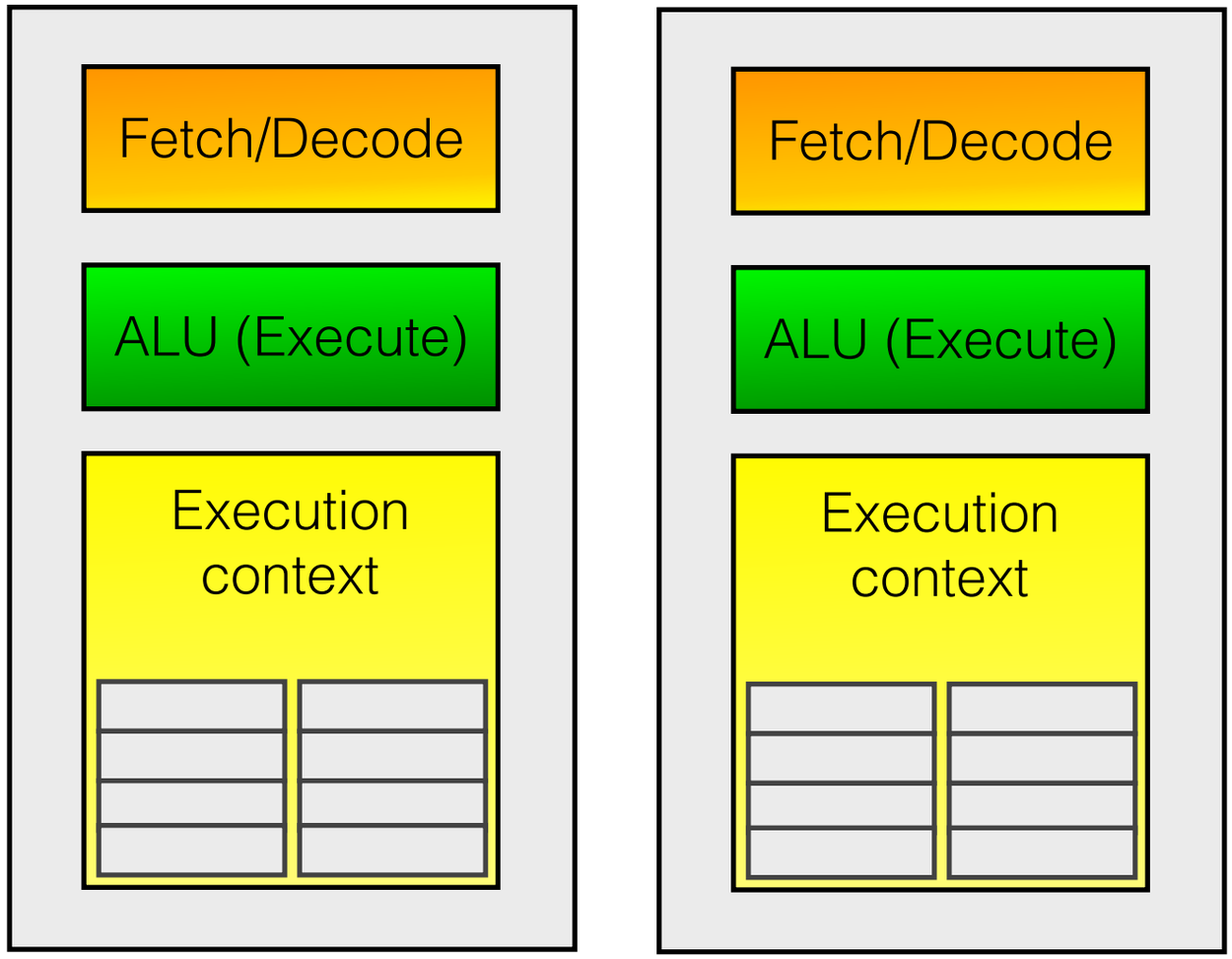



*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

*The result is a core that will stall due to code branching and memory fetches.*

# GPU: replicate cores

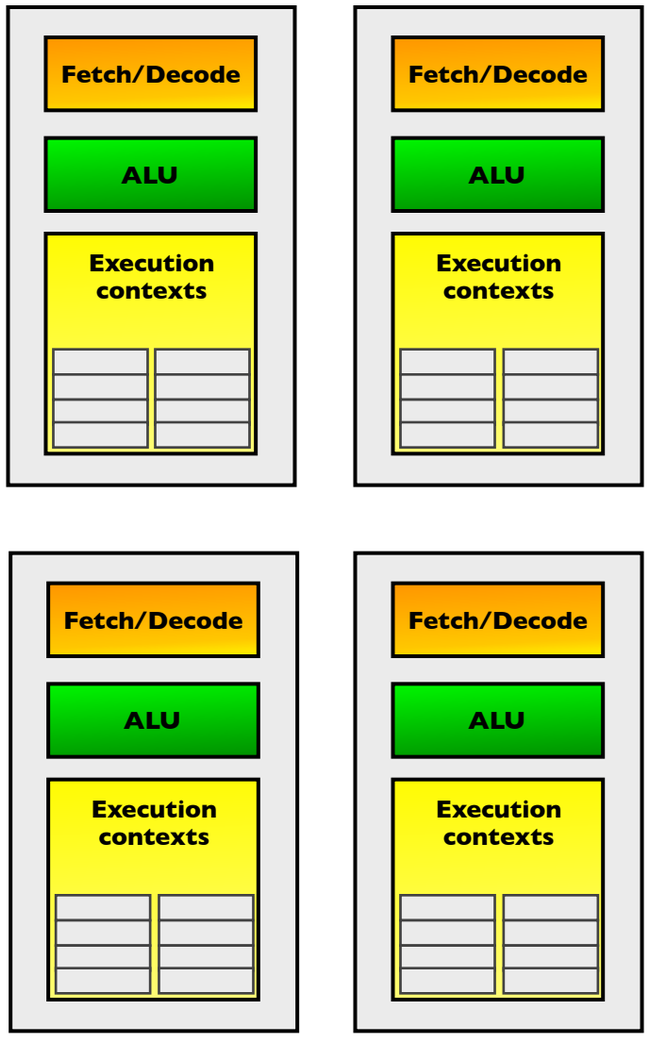
Each simplified core takes less space so double up




*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

# GPU: replicate cores

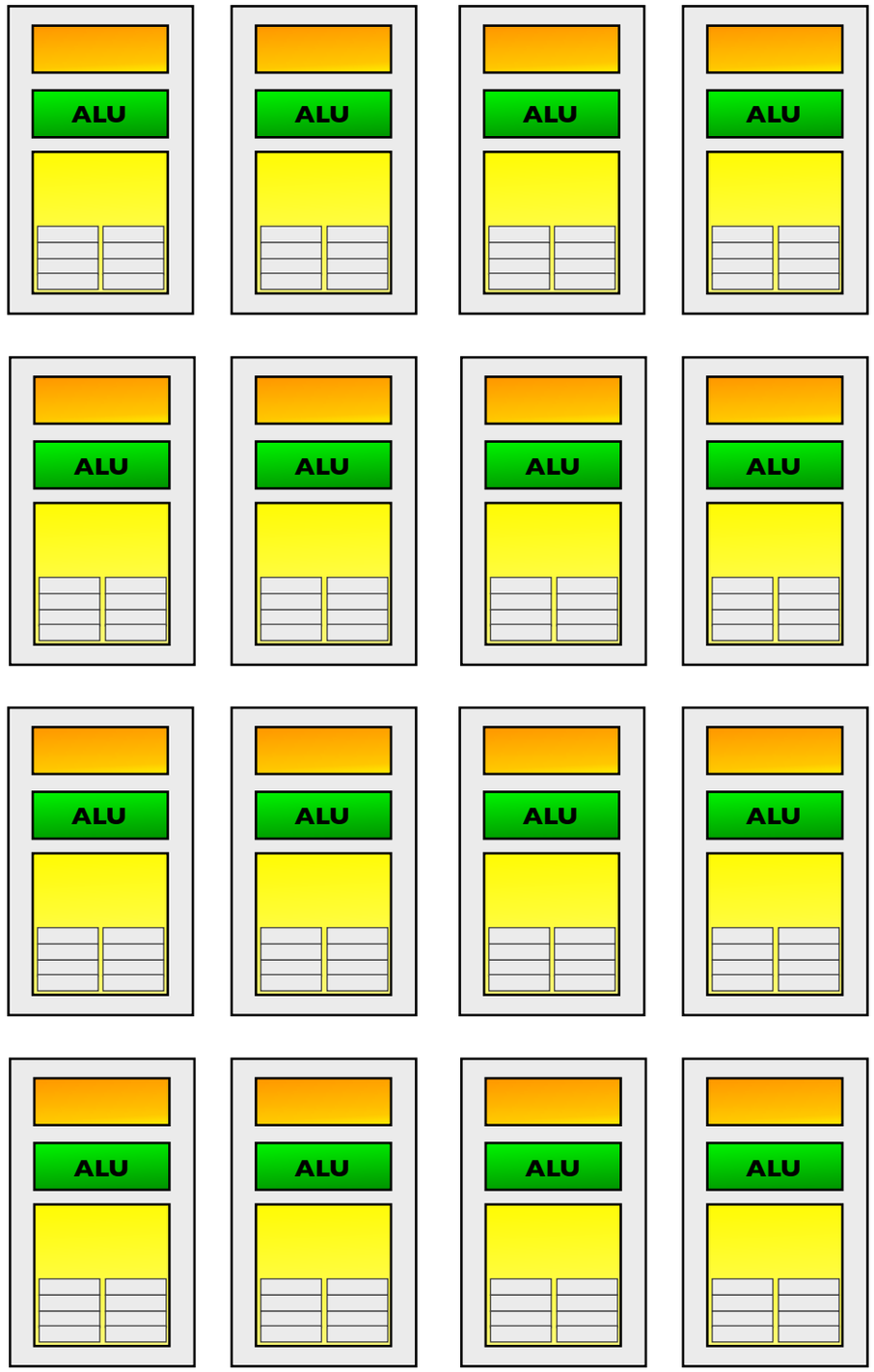
… and again …




*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

# GPU: the clone cores (ahem)

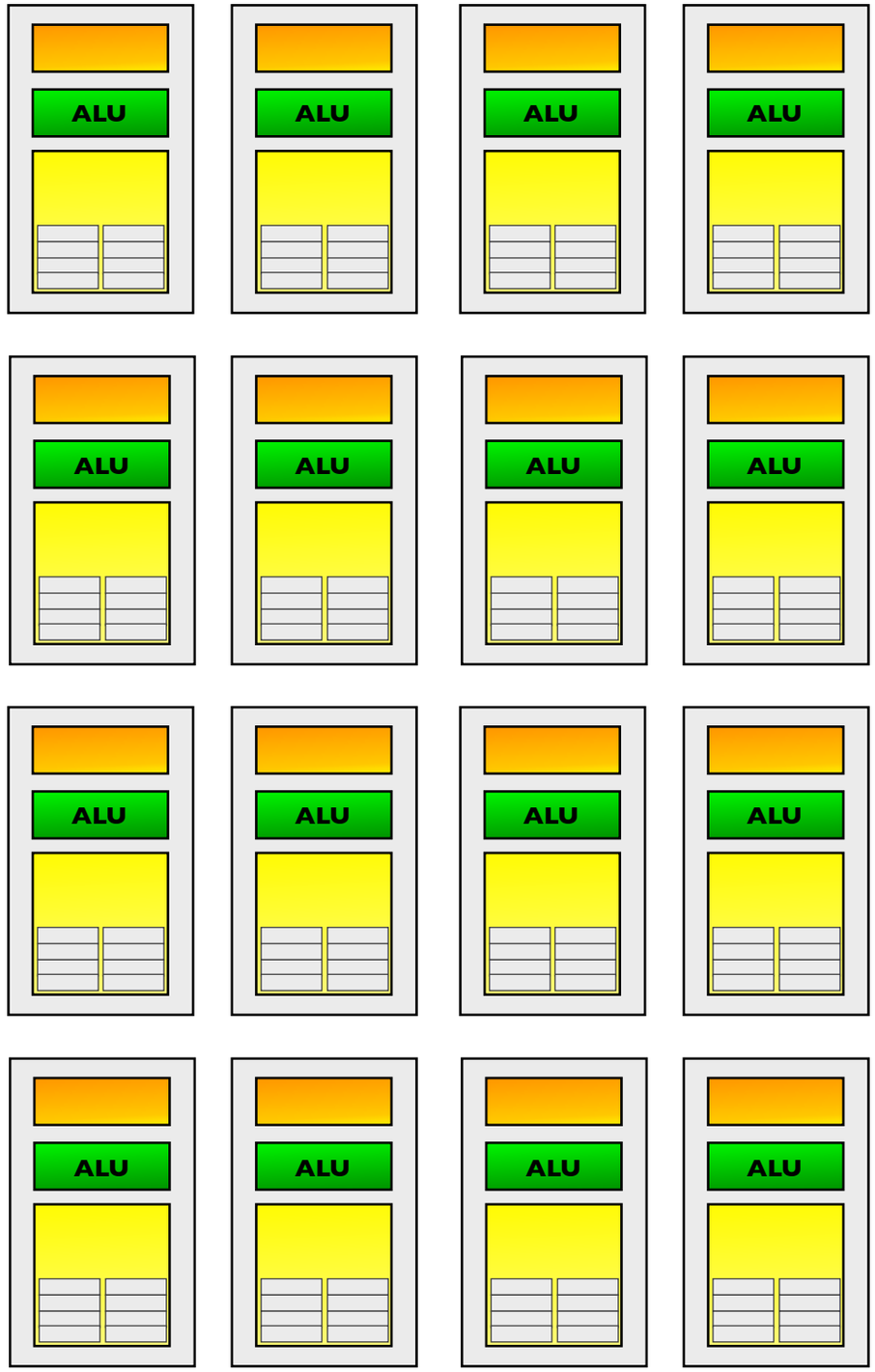
... and again to yield 16 independent instruction streams ...




*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

# GPU: the clone cores (ahem)

... and again to yield 16 independent instruction streams ...

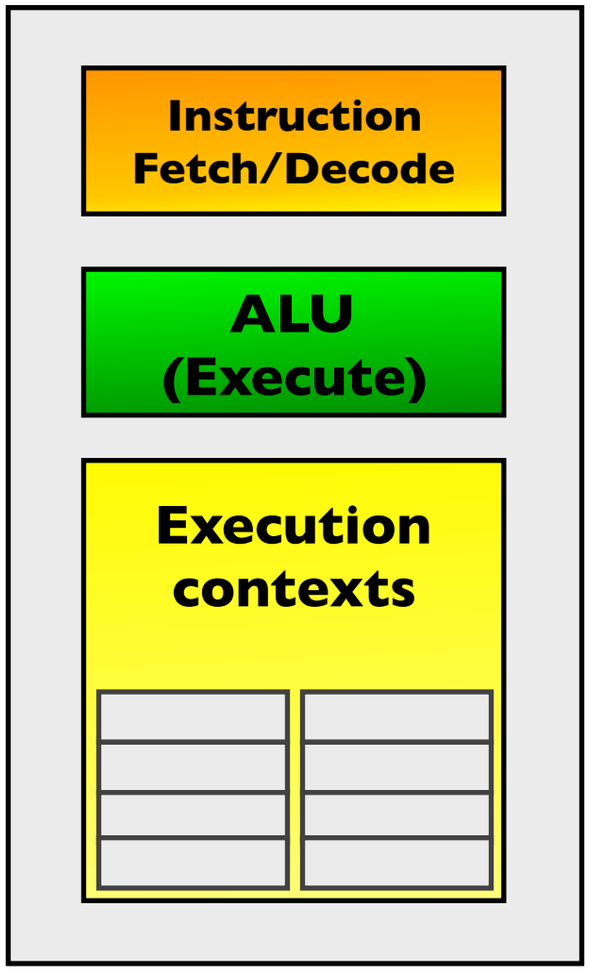



But: GPU rendering instruction streams are typically very similar.

*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

# GPU: reduce # of instruction streams

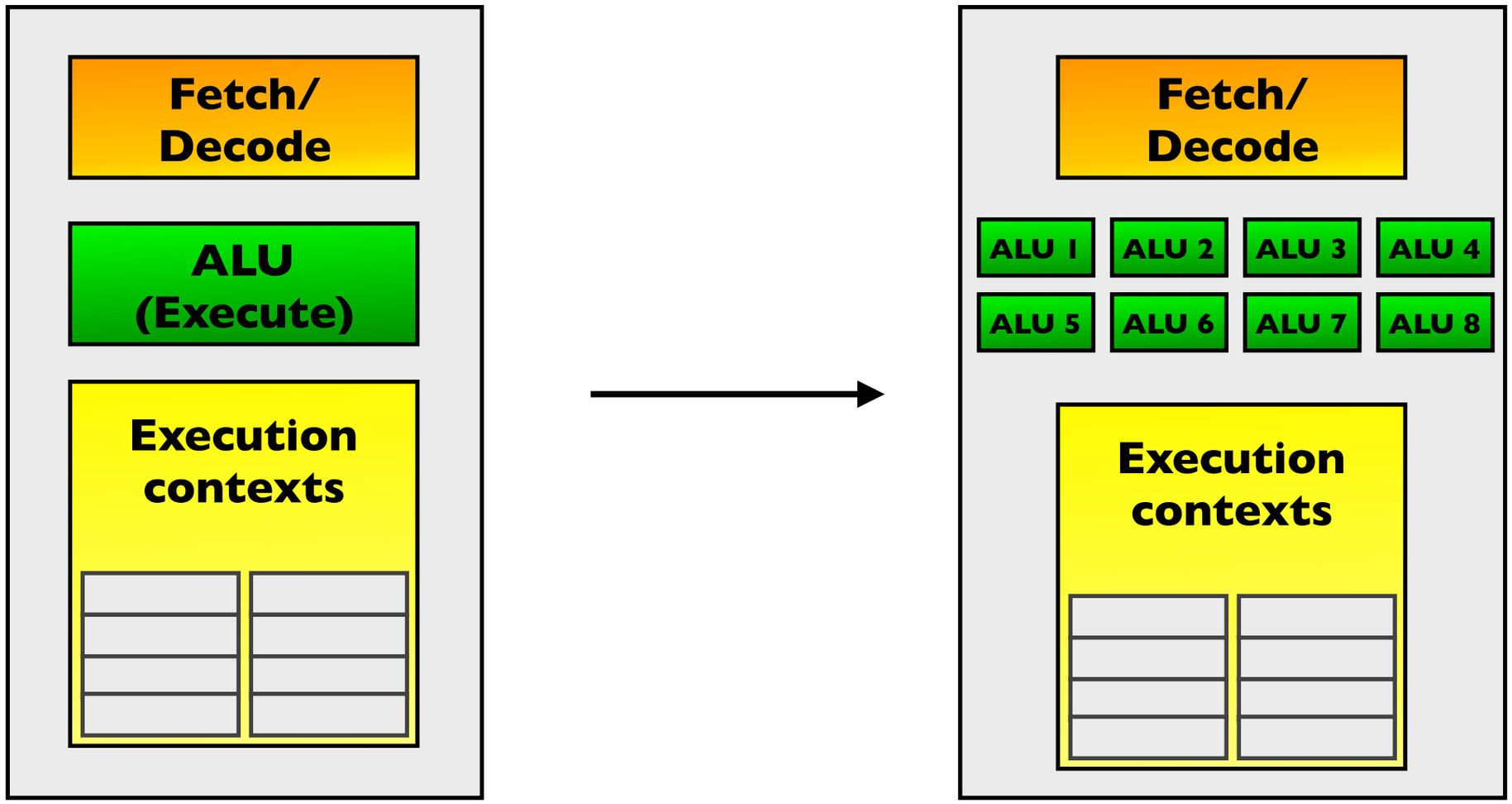
The Fetch/Decode units in those 16 clone cores are likely fetching and decoding the same instruction streams for rendering graphics...




*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

# GPU: SIMD groups

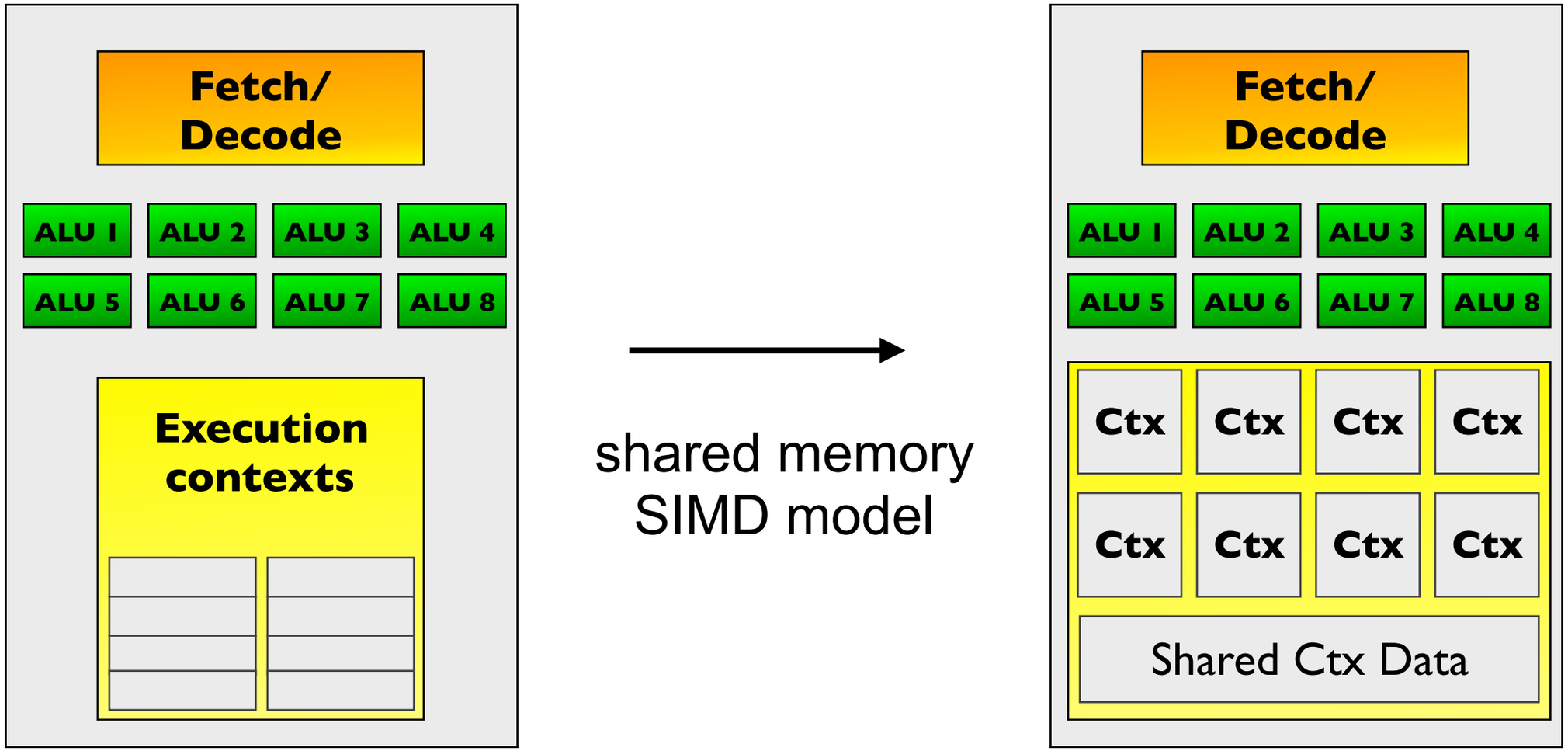
"single instruction multiple data" SIMD model:
share the cost of the instruction stream across many ALUs




*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

# GPU: shared memory for SIMD group

Provision each SIMD group with shared memory



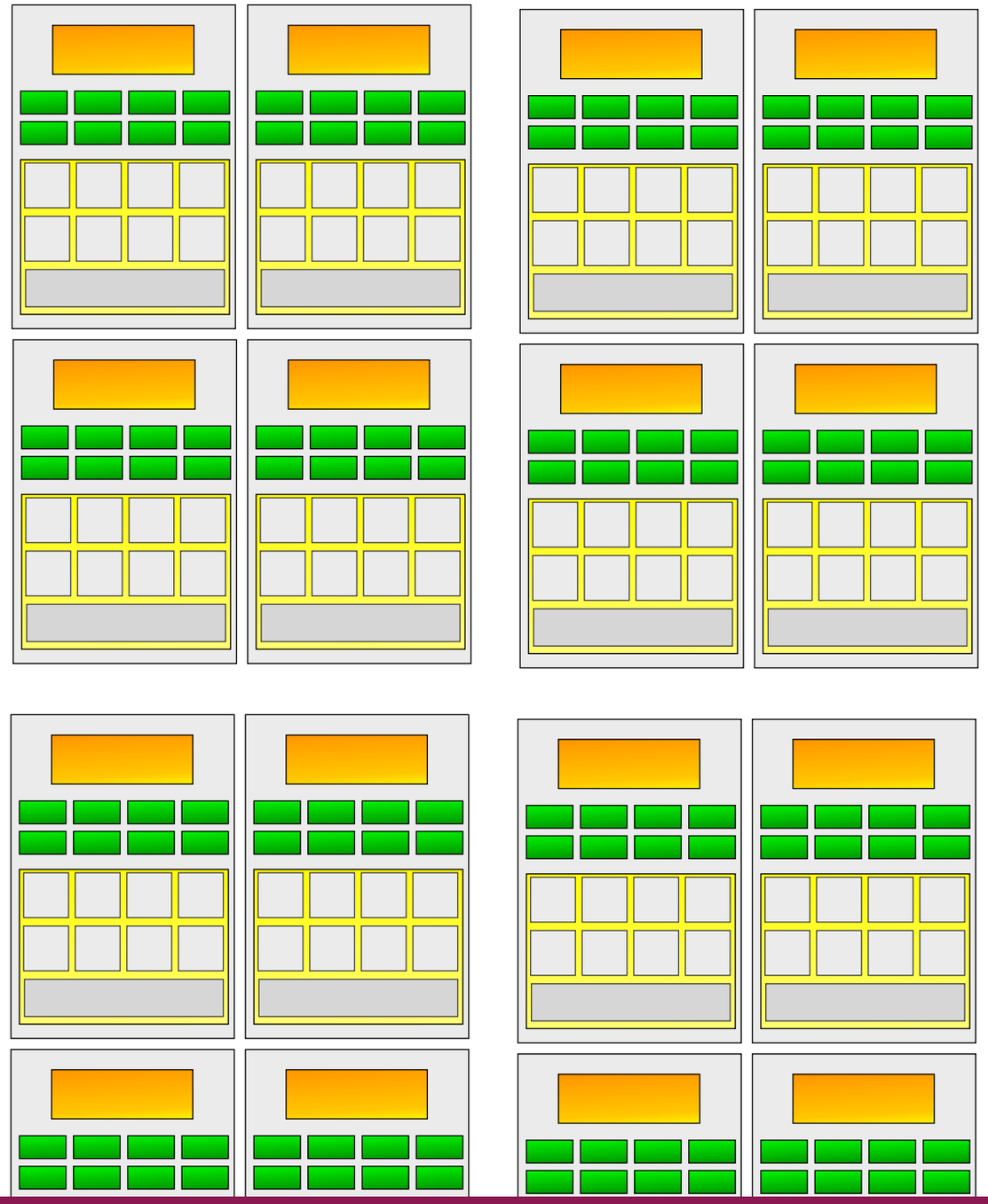

*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

*In reality the GPU cores are equipped with "substantial" register files and shared memory.*

# GPU: multiple independent cores

Summary: 128 parallel instruction streams.
Organized into 16 independent cores, each with 8 instruction streams

The cores are cloned into a massively parallel processor.

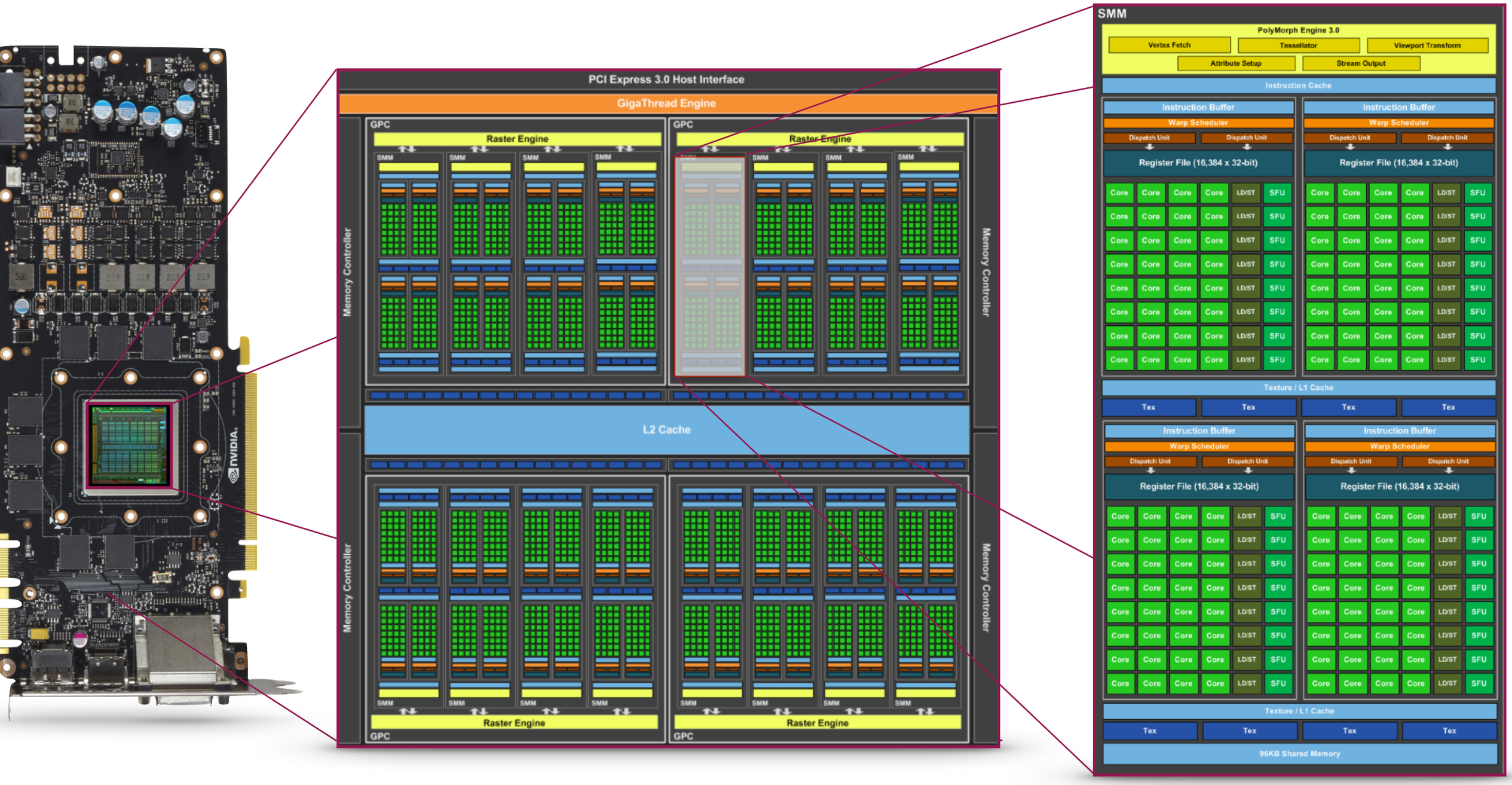


*Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian*

*This is an abstract GPU [ memory system not shown ]*

# Real GPU: Maxwell architecture

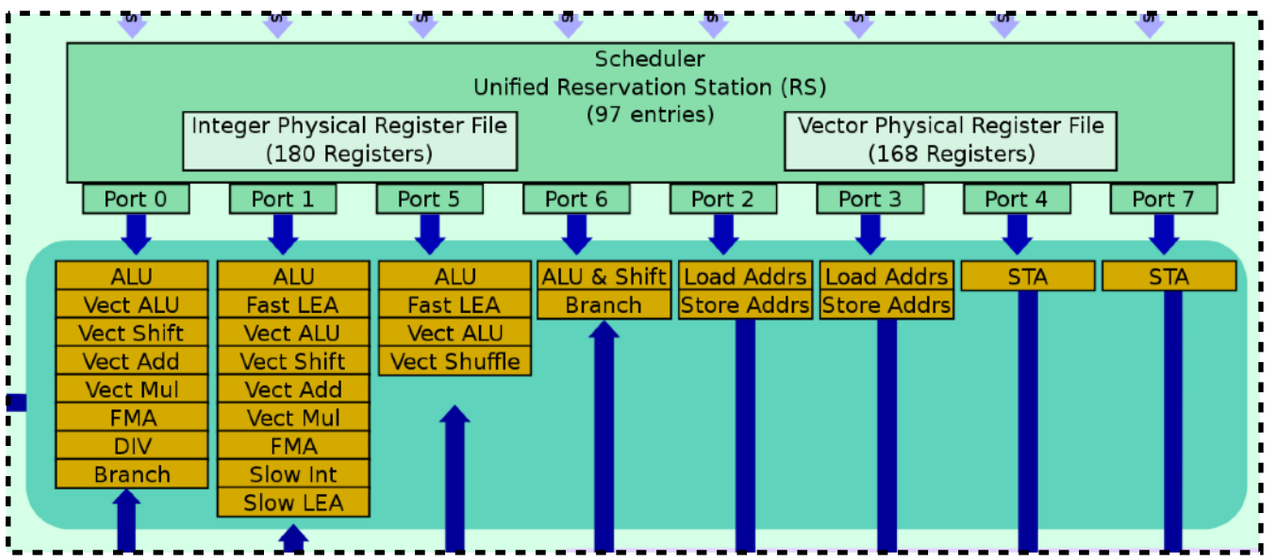
NVIDIA Maxwell GM204 GPU

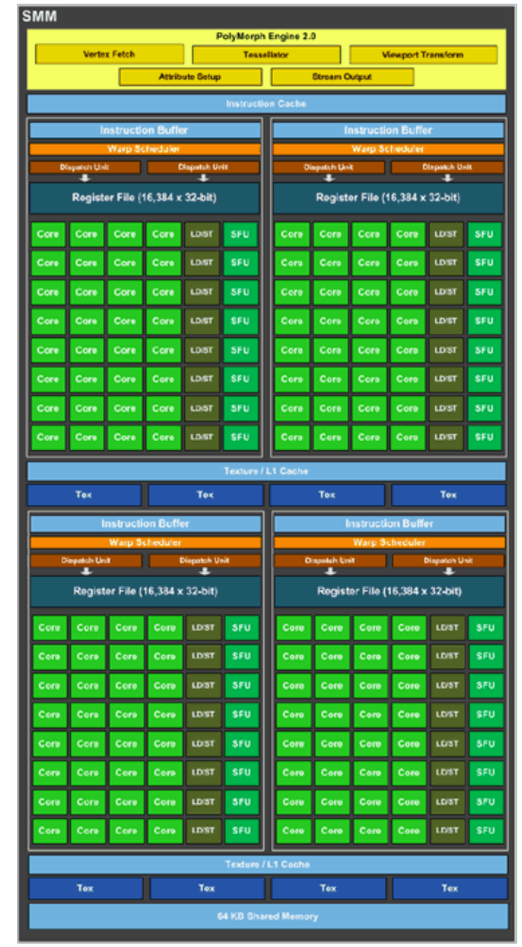



*16 Maxwell cores each have four SIMD clusters with 32 ALUs.*
*Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)*

# CPU v GPU: fundamental difference #1

Each CPU core executes scalar <u>or</u> vector operations.
Each GPU core only executes vector instructions.




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

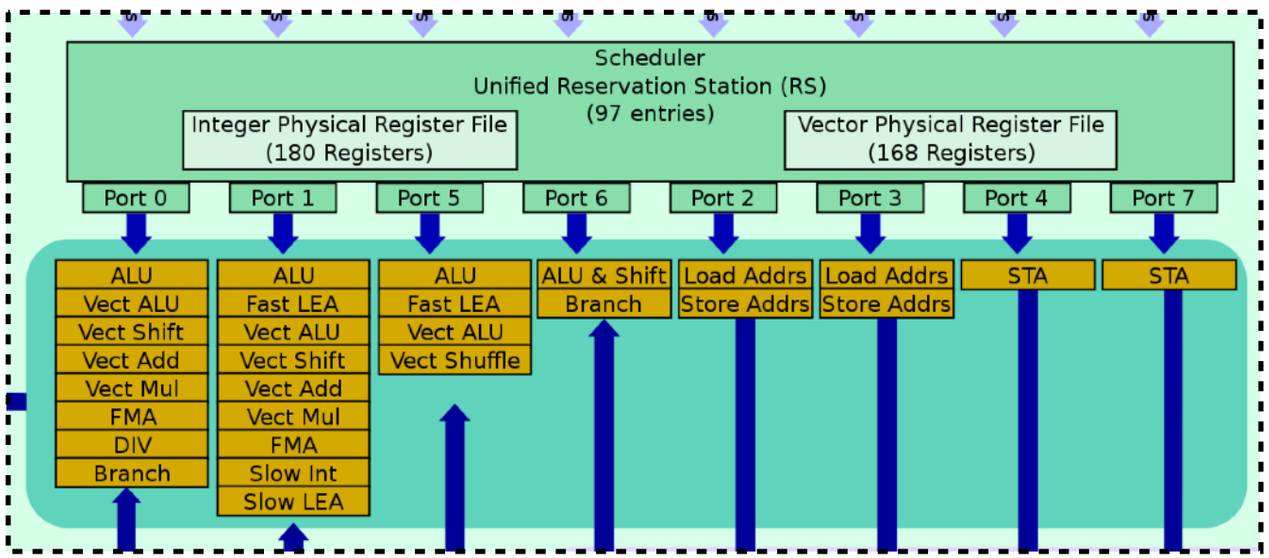
GPU: SIMD parallel execution of **all** operations

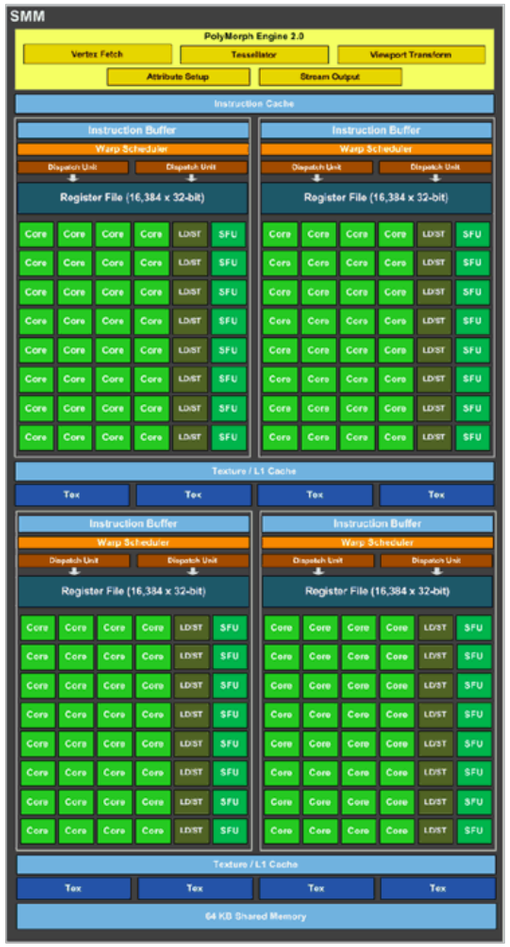
http://en.wikichip.org/wiki/intel/microarchitectures/skylake

*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

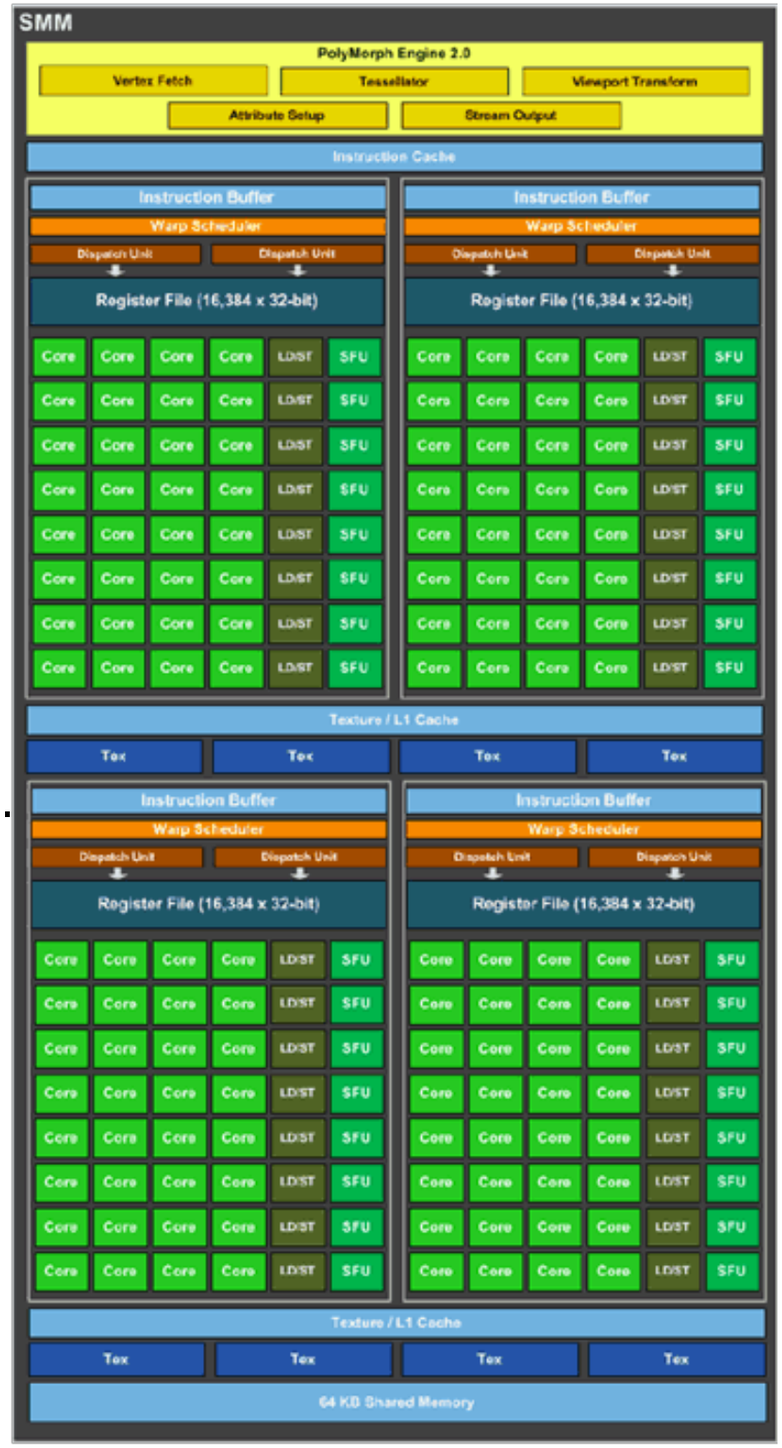
http://en.wikichip.org/wiki/intel/microarchitectures/skylake

*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



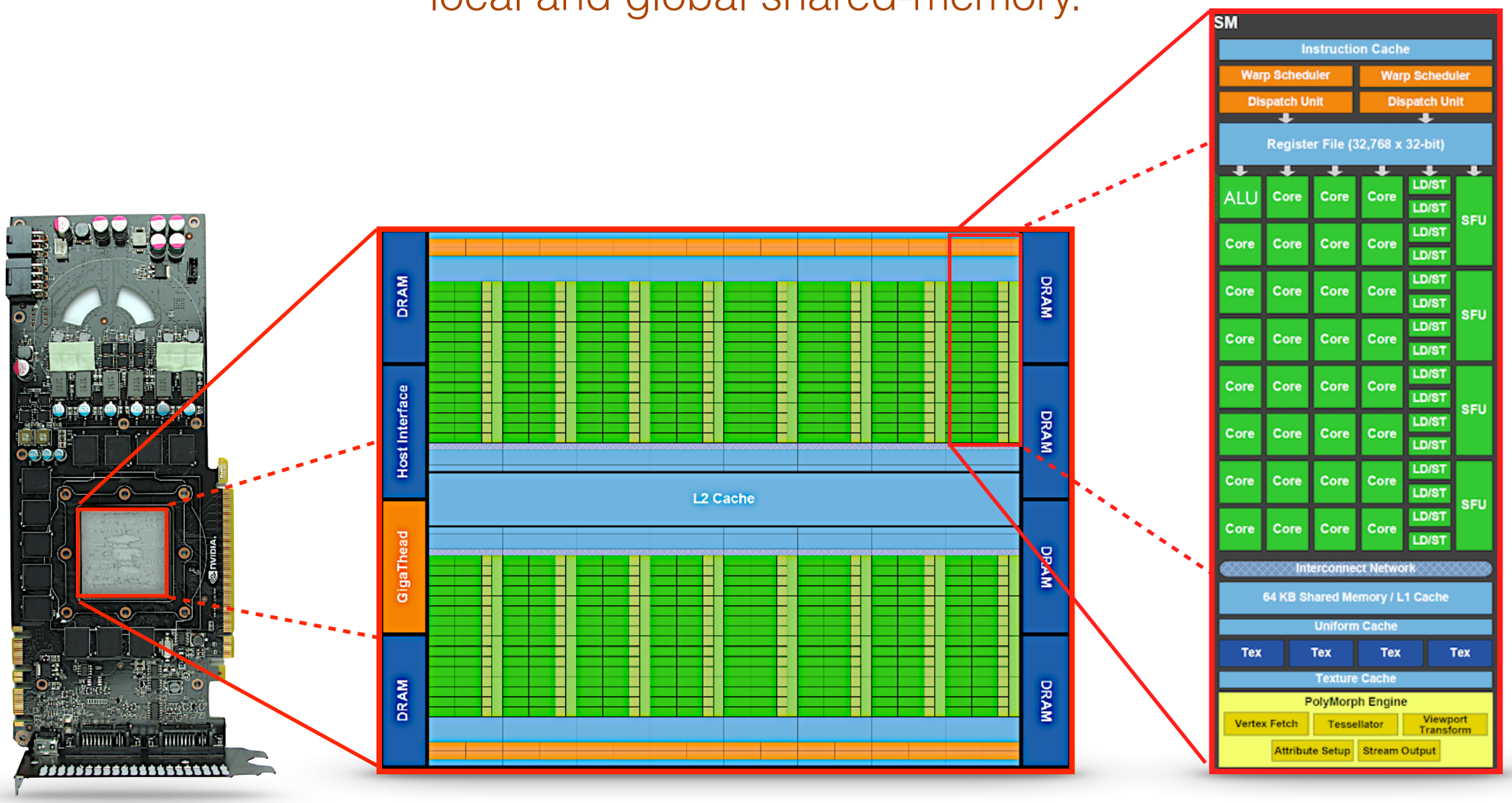

** SIMD width here is the number of ALUs in one of the core's vector unit.*
*The actual specifics vary but this is a good abstract viewpoint.*

# 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.

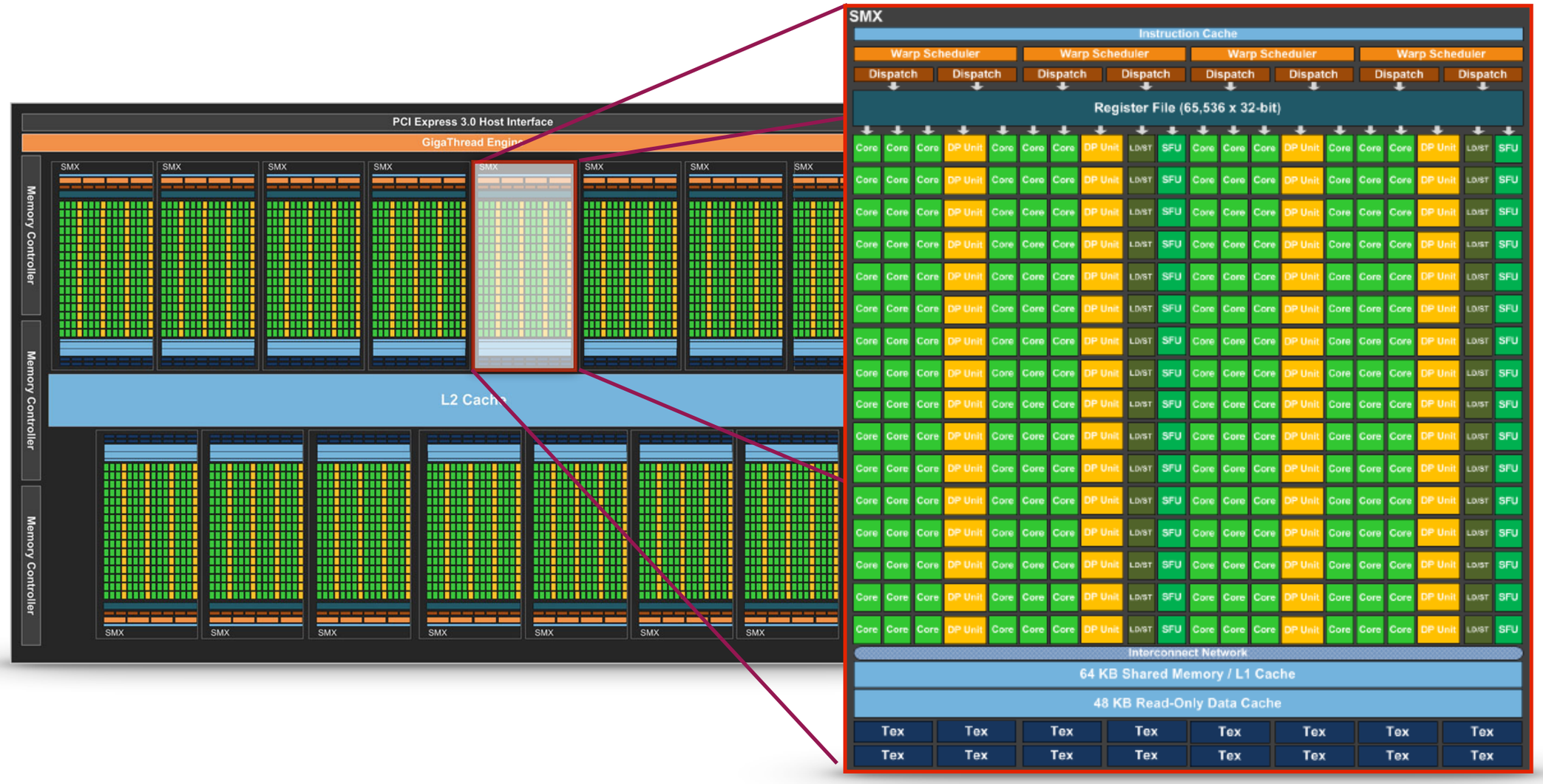



Each Fermi core (SM) has a SIMD clusters of 32 FPUs
Data streams at ~50 GFLOAT/s and computes up to 1.4 TFLOP/s (SP)

*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: NIVIDA Titan

GK110: 15 cores that cluster 192 FPU each.

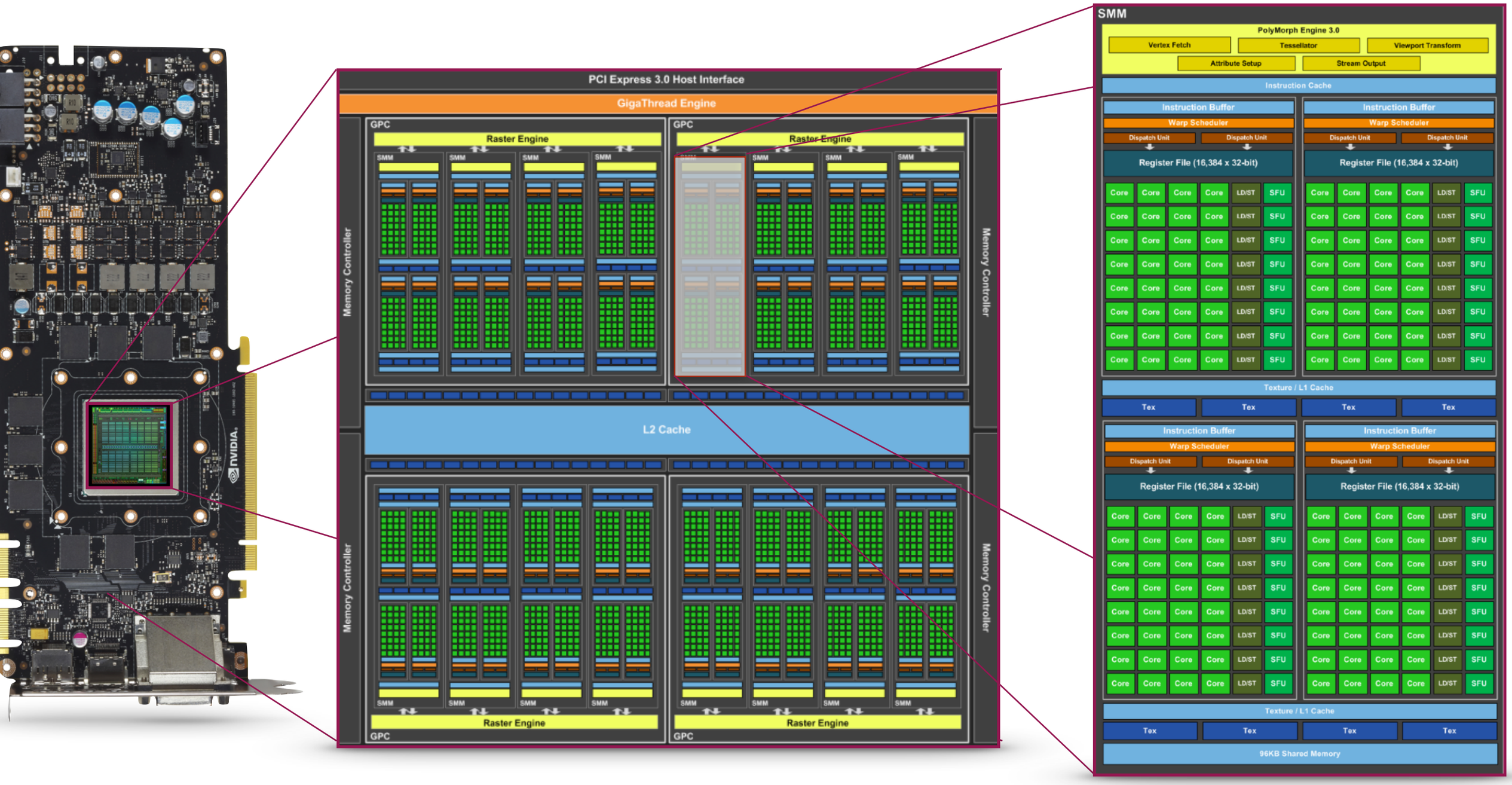



Each Kepler core (SMX) has six SIMD clusters of 32 ALUs
Data streams at ~70 GFLOAT/s and peak 4+ TFLOP/s (SP)

*Image credits: http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last/3*
*http://www.tomshardware.com/reviews/geforce-gtx-titan-gk110-review,3438.html*

# GPU: Maxwell GPU

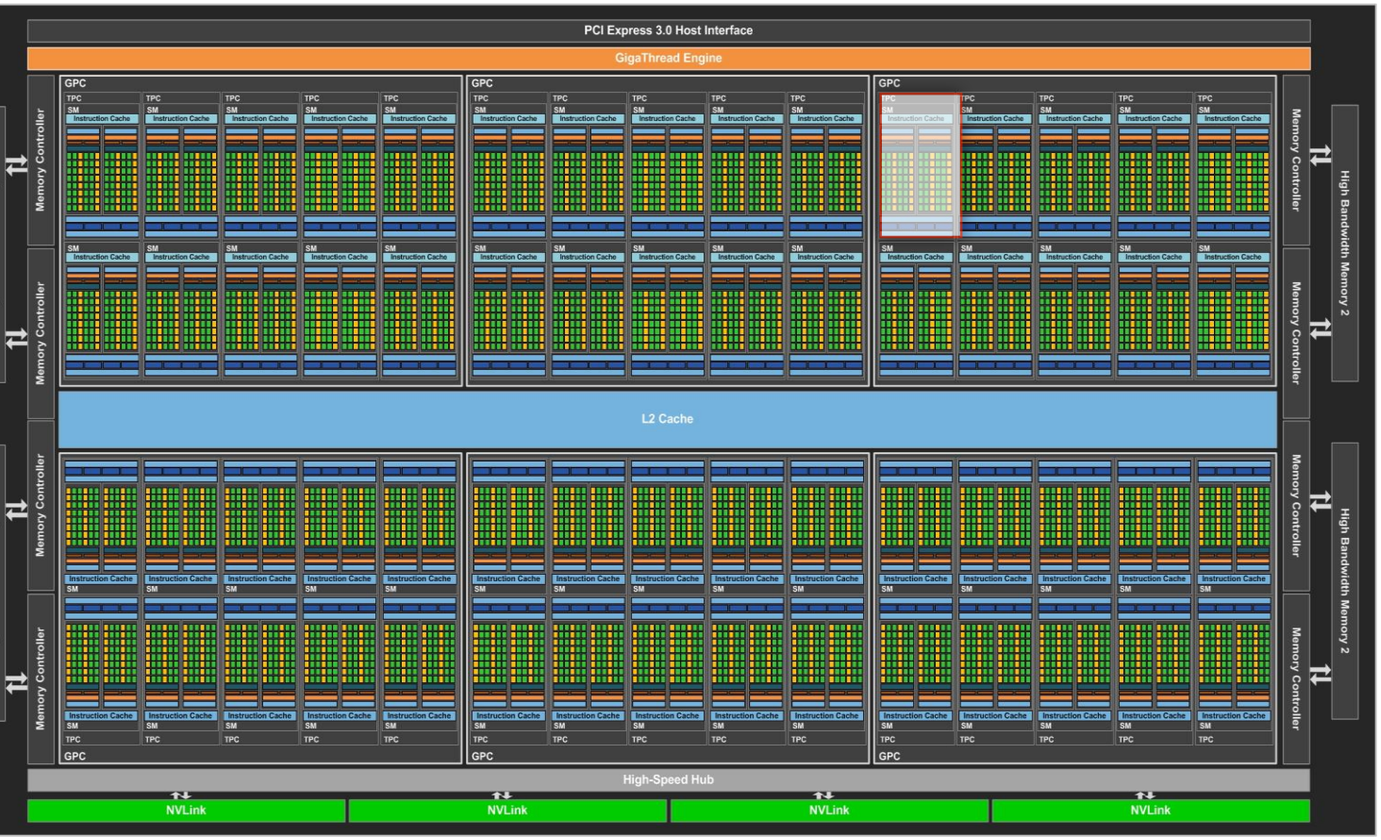
NVIDIA Maxwell GM204 GPU




*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

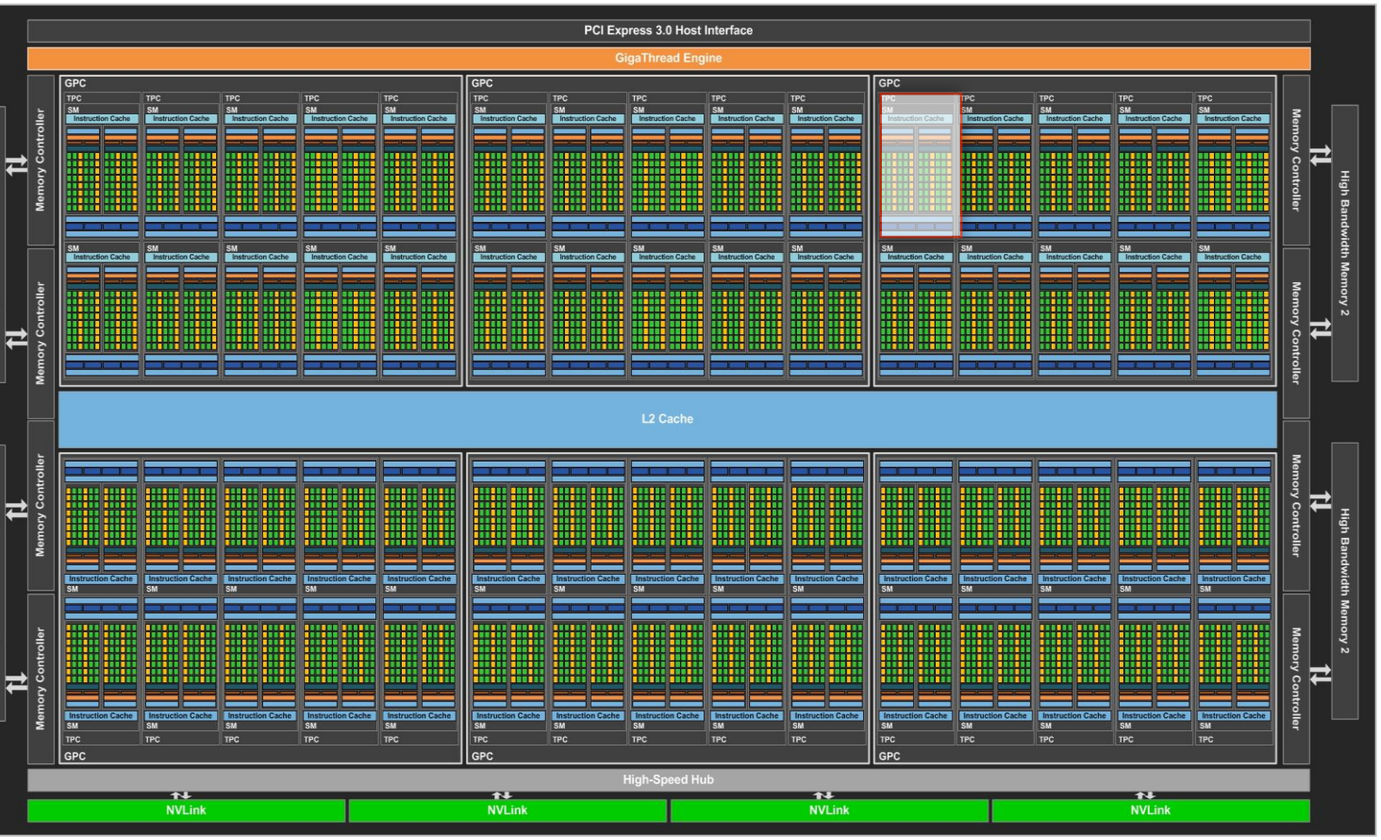
Professional NVIDIA Pascal GP100 architecture with 60 cores using 16nm fab size



*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: Pascal GPU

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



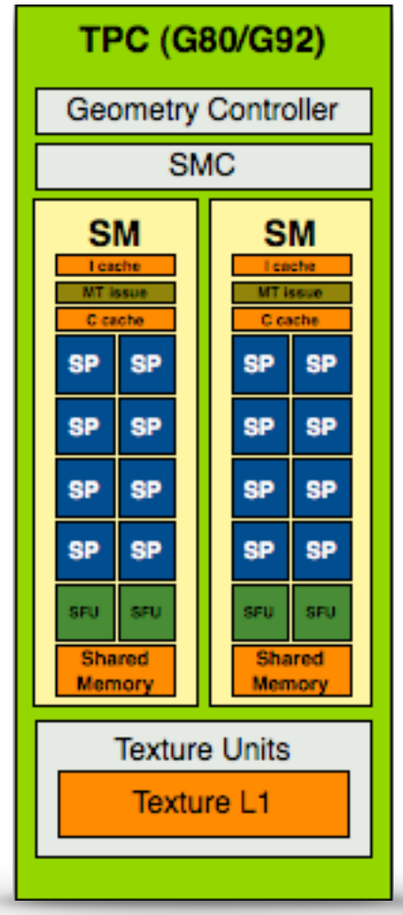
Consumer variants use GDDR5(x) memory with up to 480 GB/s bandwidth and up to 3584 ALUs with peak 10.1 TFLOP/s (SP) 0.3 TFLOP/s (DP)

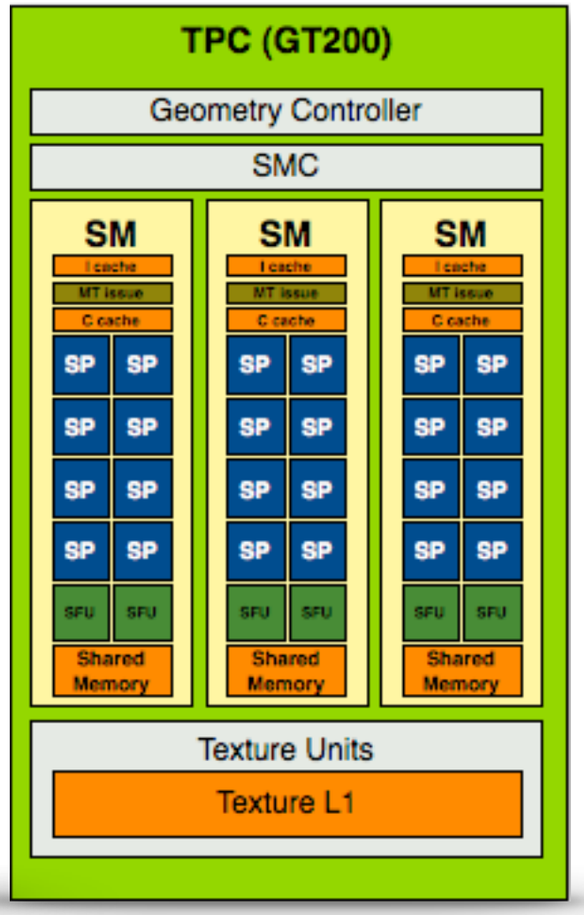
*60 Pascal GP100 cores each with two SIMD clusters of 32 ALUs (3840 Total).*
*HBM2 memory streams data at ~1 TB/s and peak 10.6 TFLOP/s (SP), 5.3TFLOP/s (DP)*

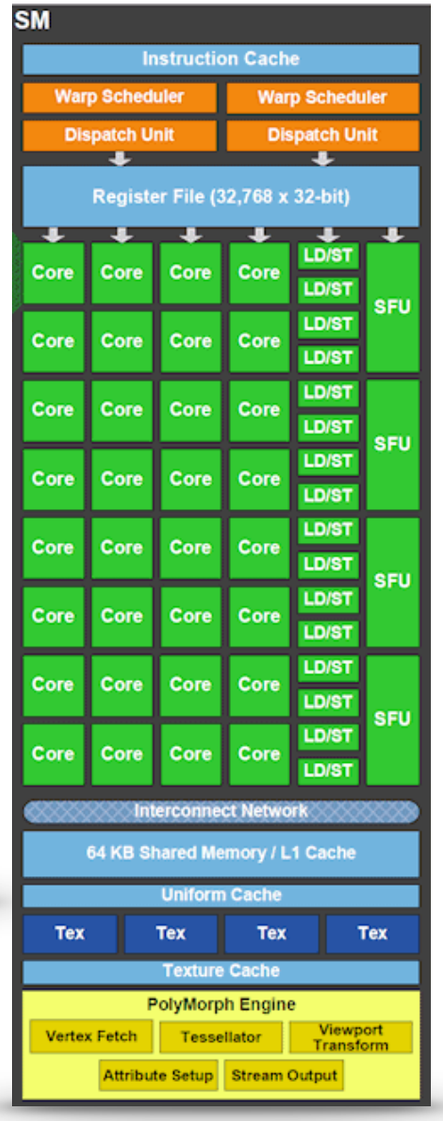
# GPU: trends in FPU Clusters

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

| 2007: G80 | 2008: Tesla | 2010: Fermi | Q4 2012: Kepler GK110 |
|---|---|---|---|



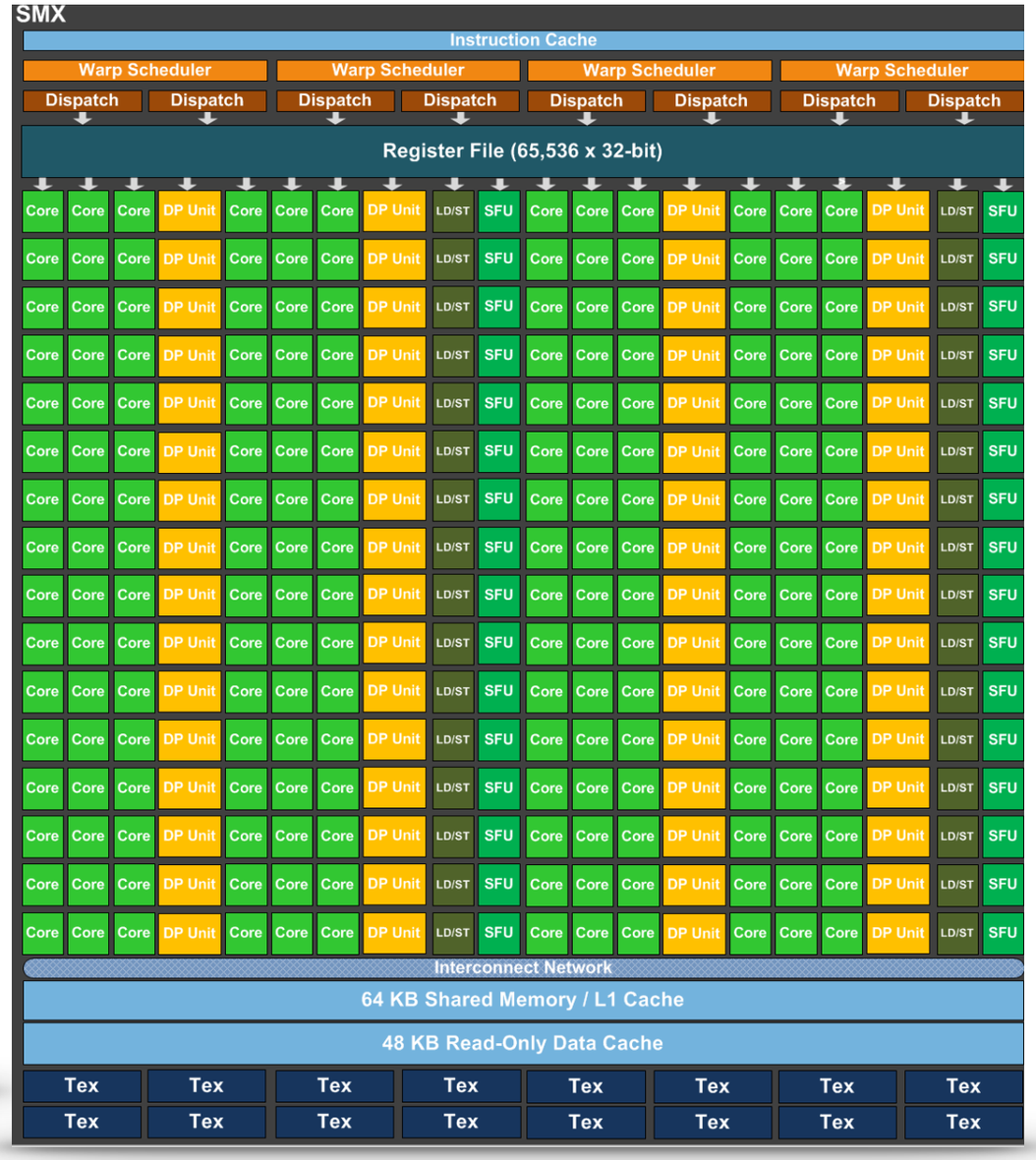



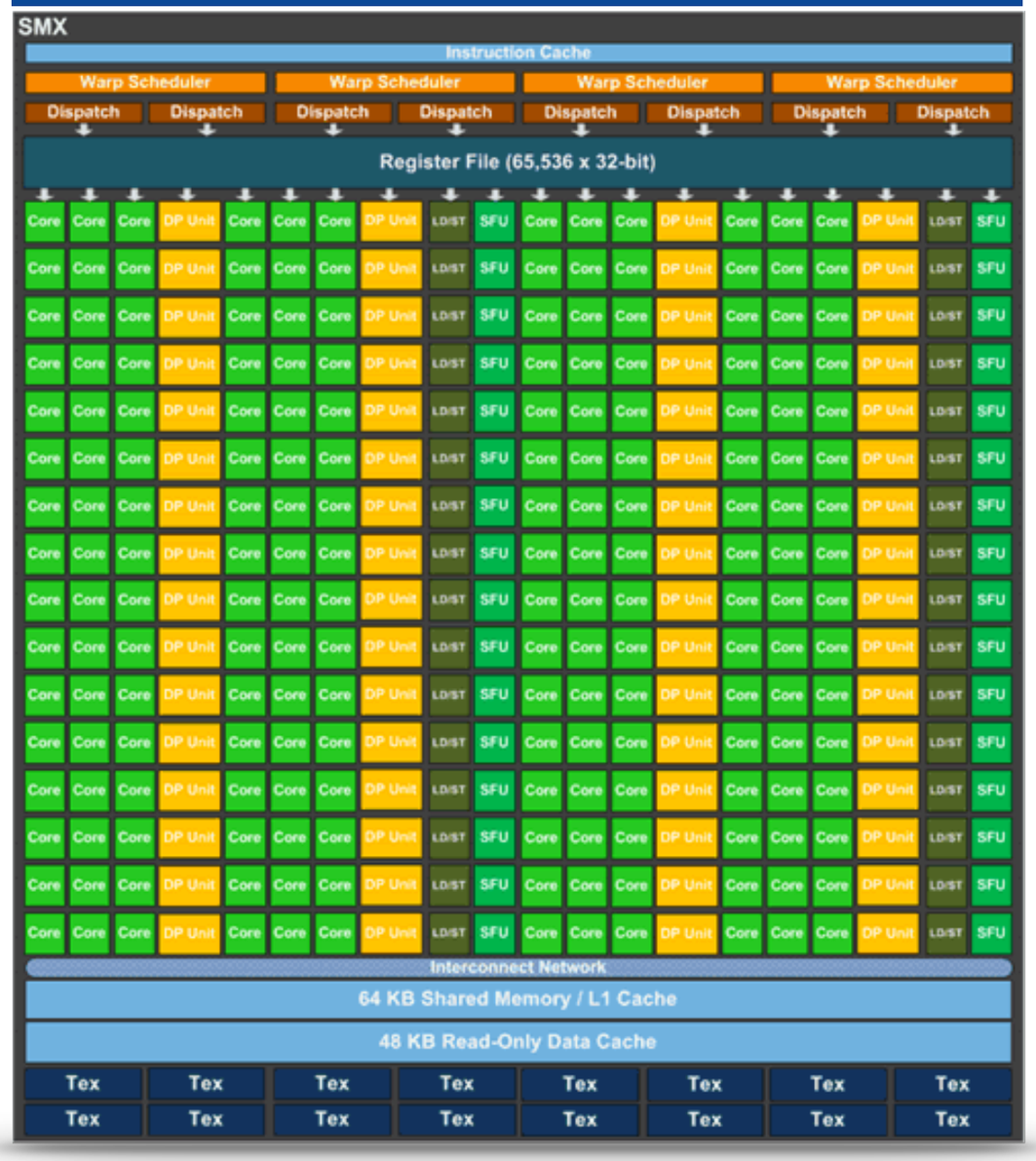

Image sources: http://forum.beyond3d.com/showthread.php?t=58668&page=140, http://www.anandtech.com/show/2549/2 , http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

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

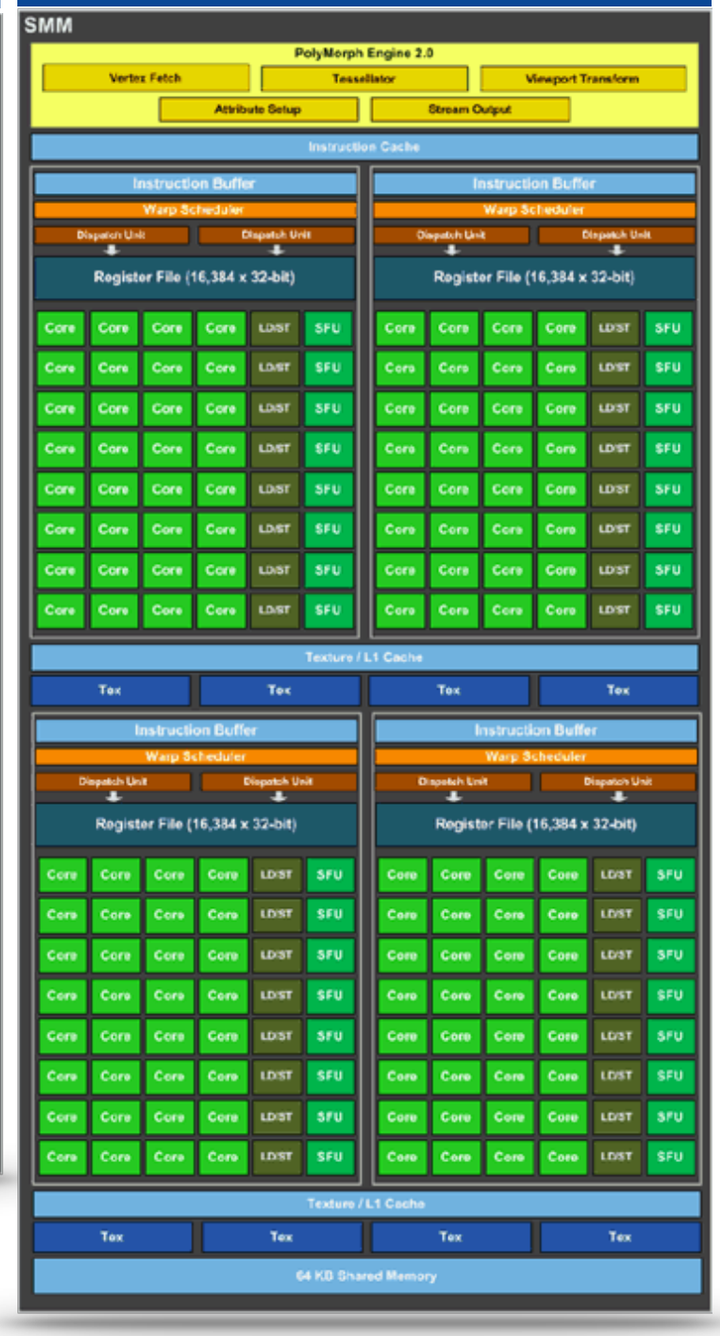
# GPU: Kepler to Maxwell to Pascal

The FPU clusters ("core") in 3 latest 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.*

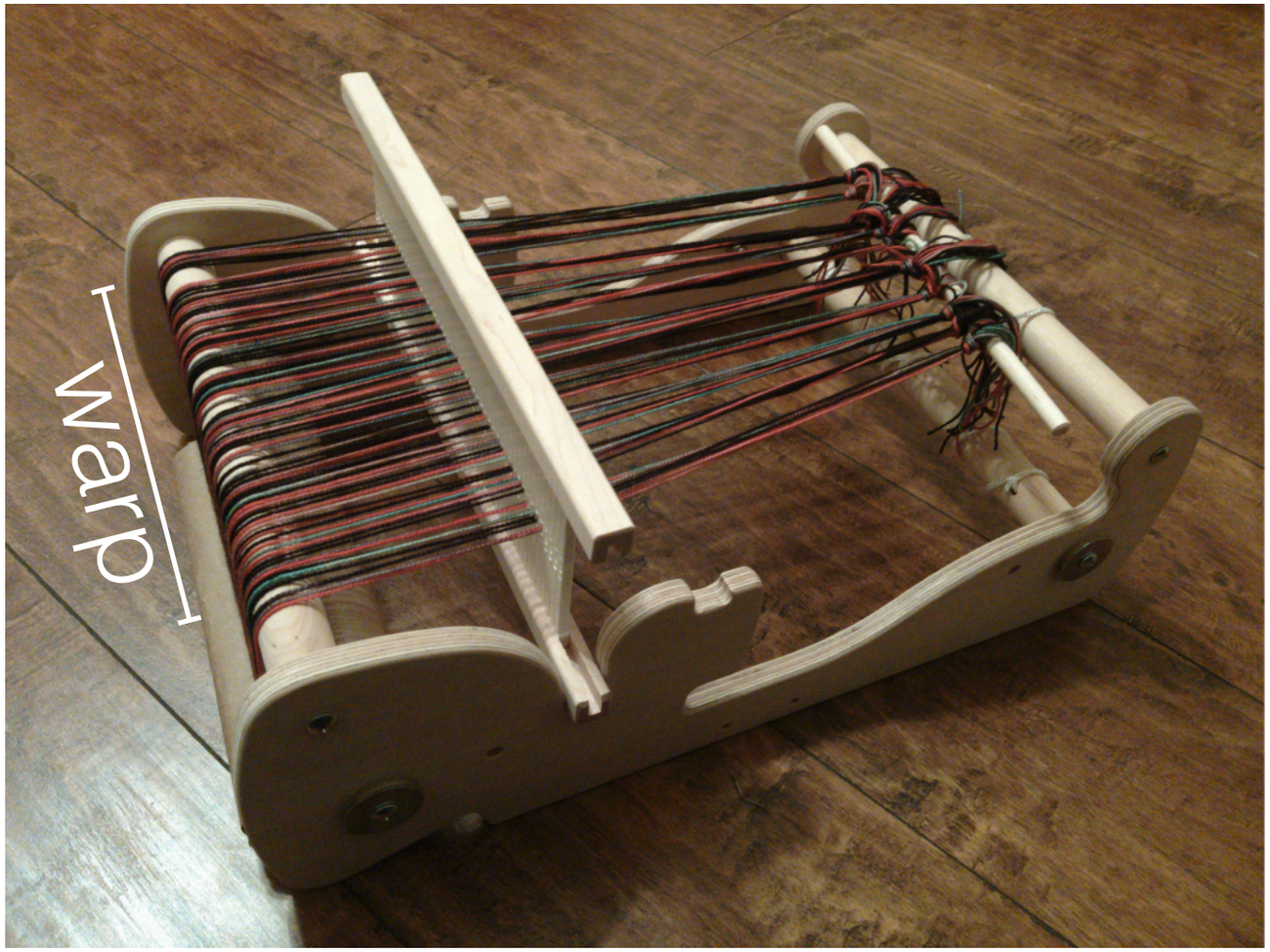
# Part 2b: GPU programming with CUDA Threading

NVIDIA's Compute Unified Device Architecture
GPU programming model

# Warped Terminology

CUDA

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

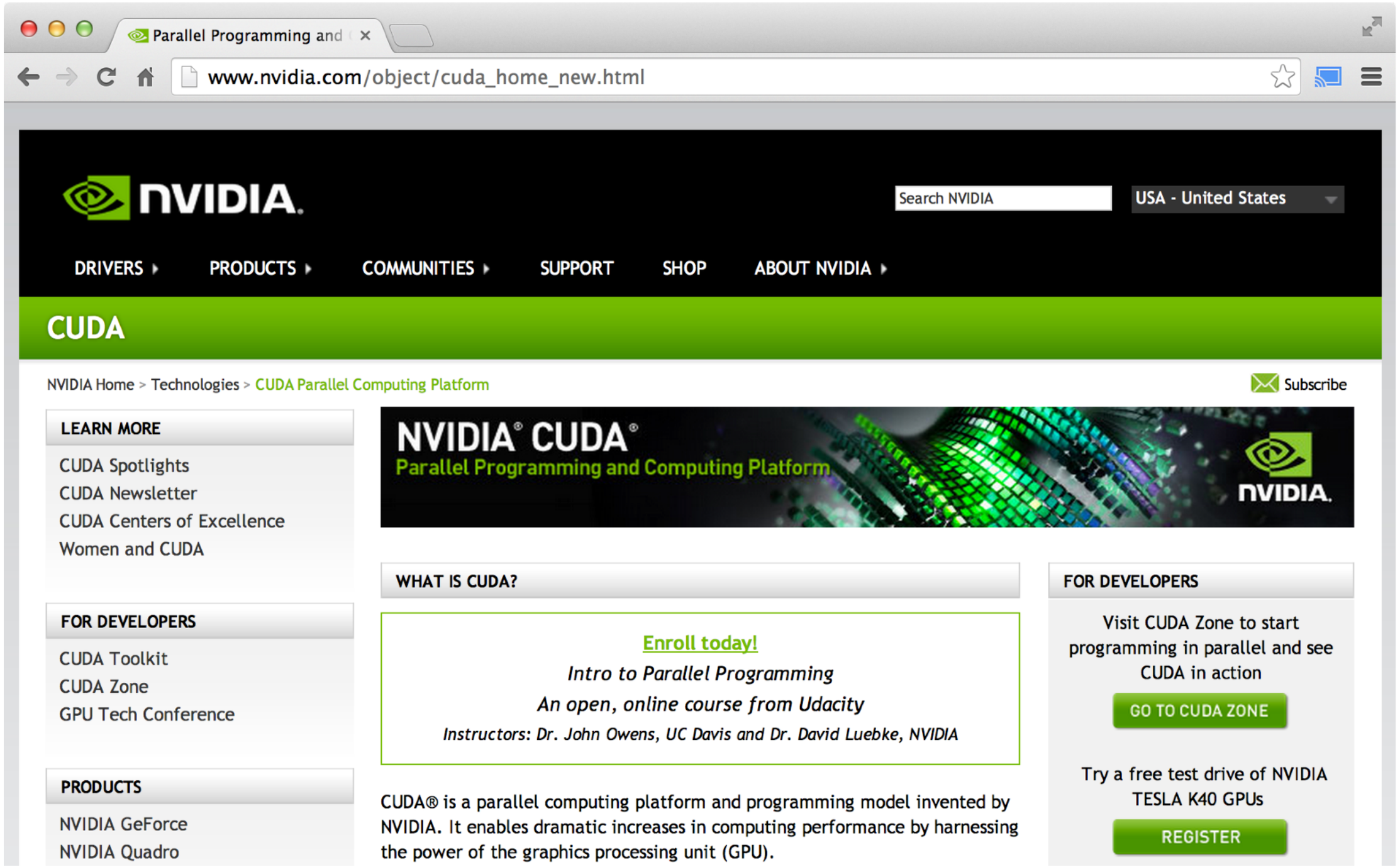



- We refer instead to a thread array and SIMD groups.

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

# CUDA: compute unified device architecture

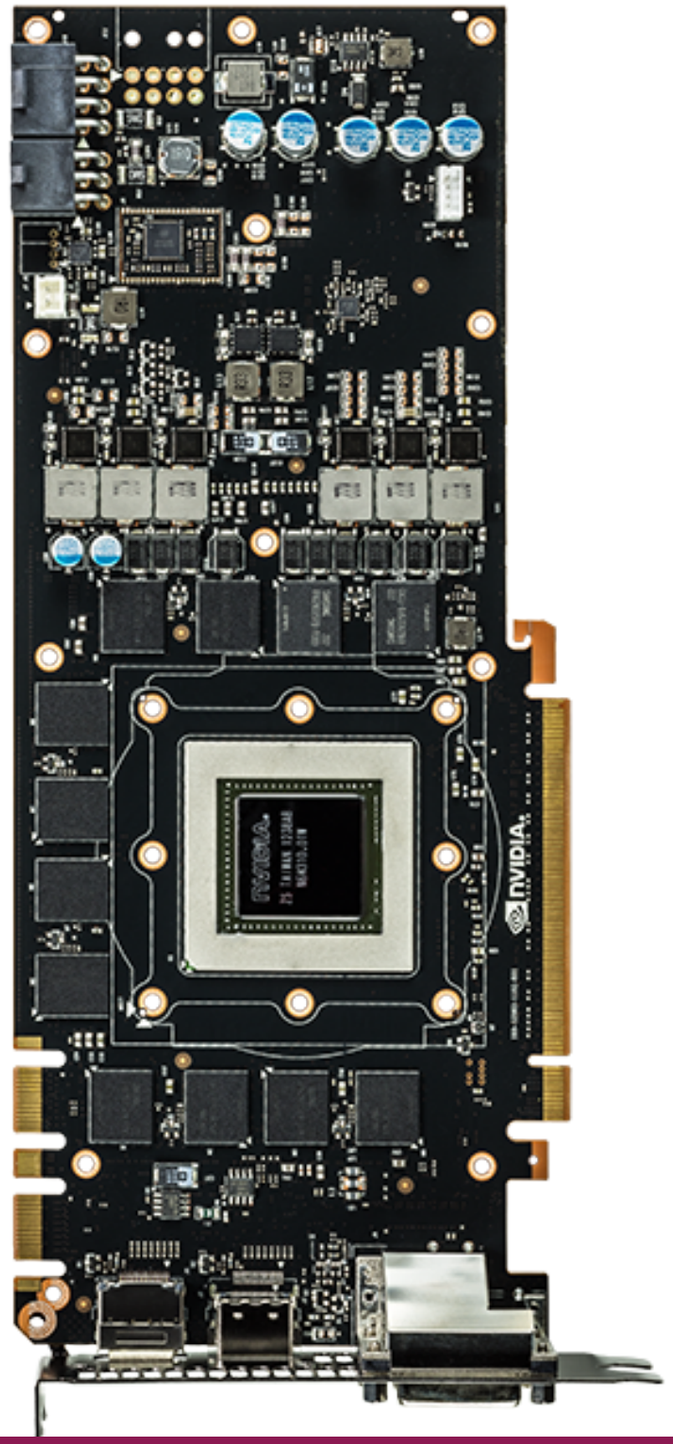
CUDA was released by NVIDIA in 2007.




*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



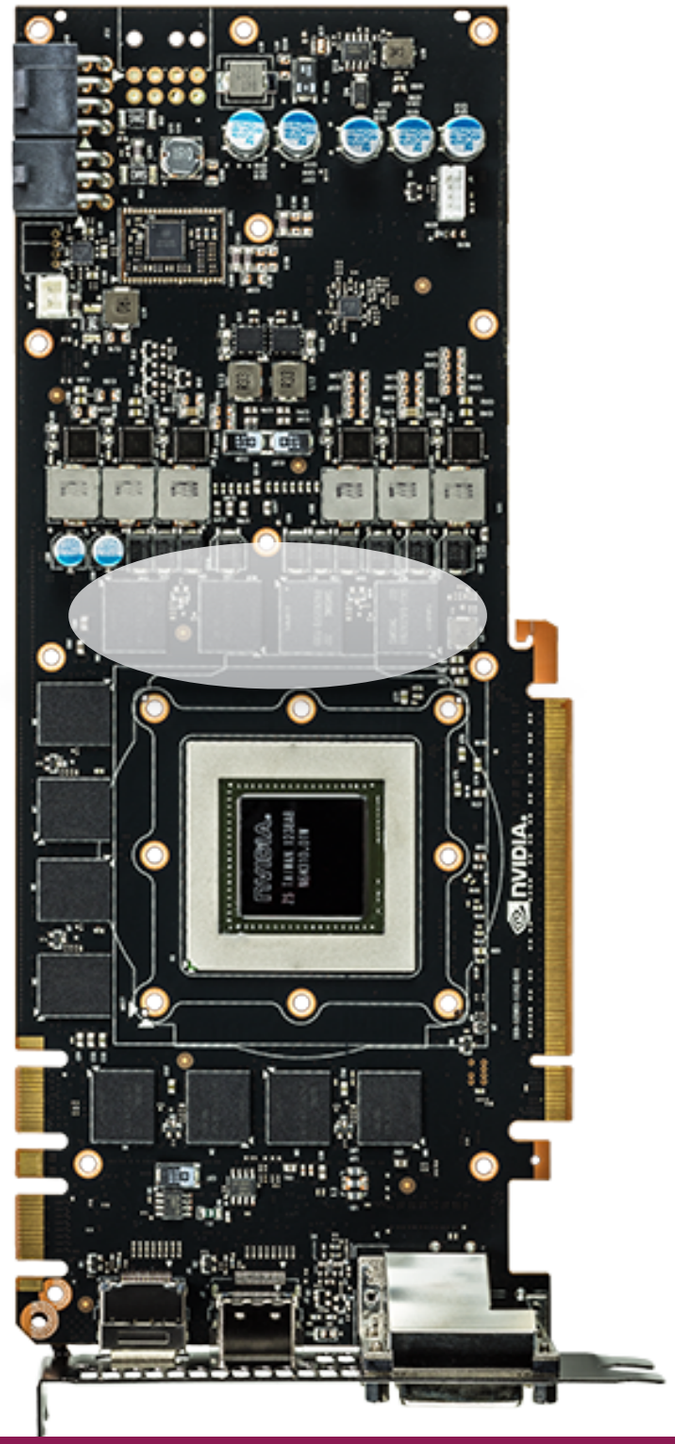

https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780

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

# CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE

1. cudaMalloc: allocate memory for a DEVICE array



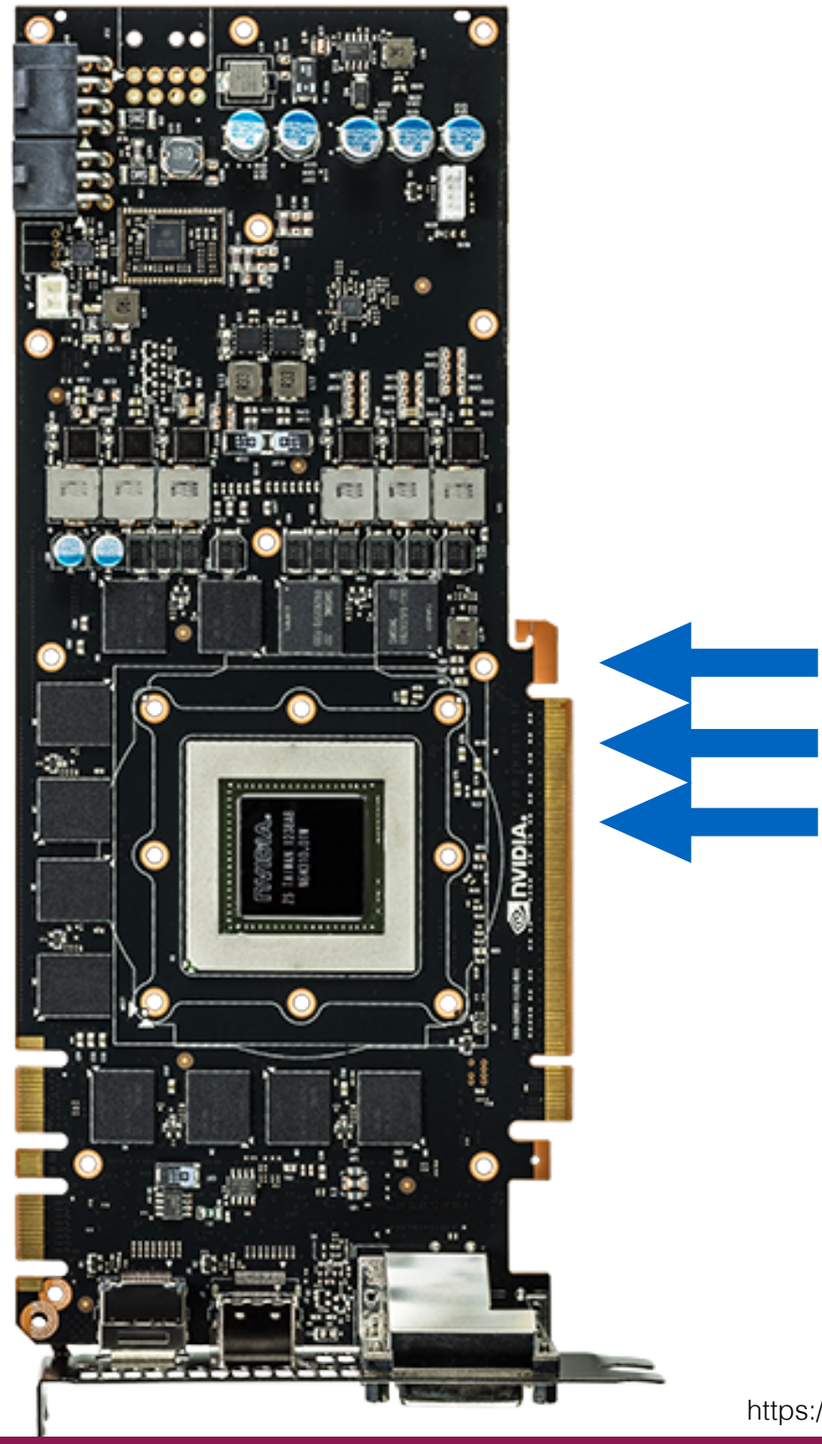
https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780

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

# CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE

1. cudaMalloc: allocate memory for a DEVICE array



2. cudaMemcpy: copy data from HOST to DEVICE array

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

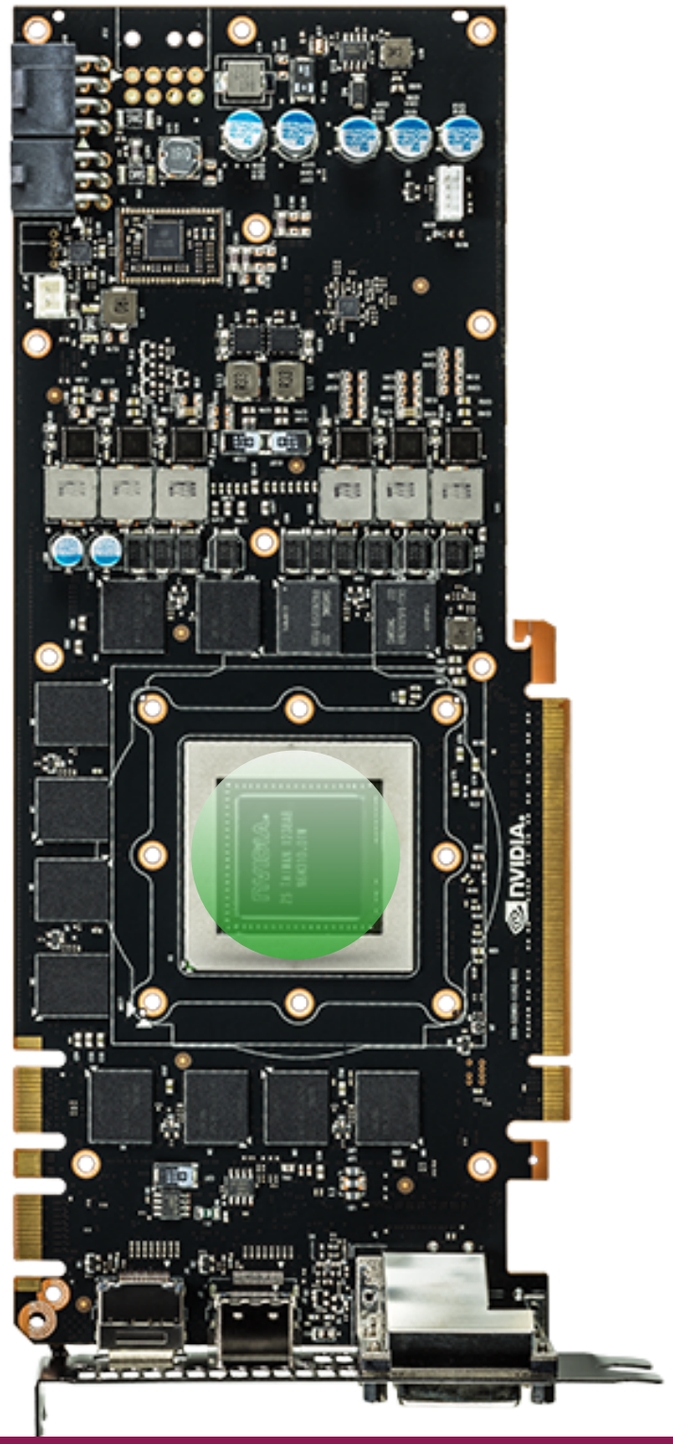
*Key observation: the DEVICE and HOST are asynchronous.*
*Operations are queued on the DEVICE.*

# 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

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

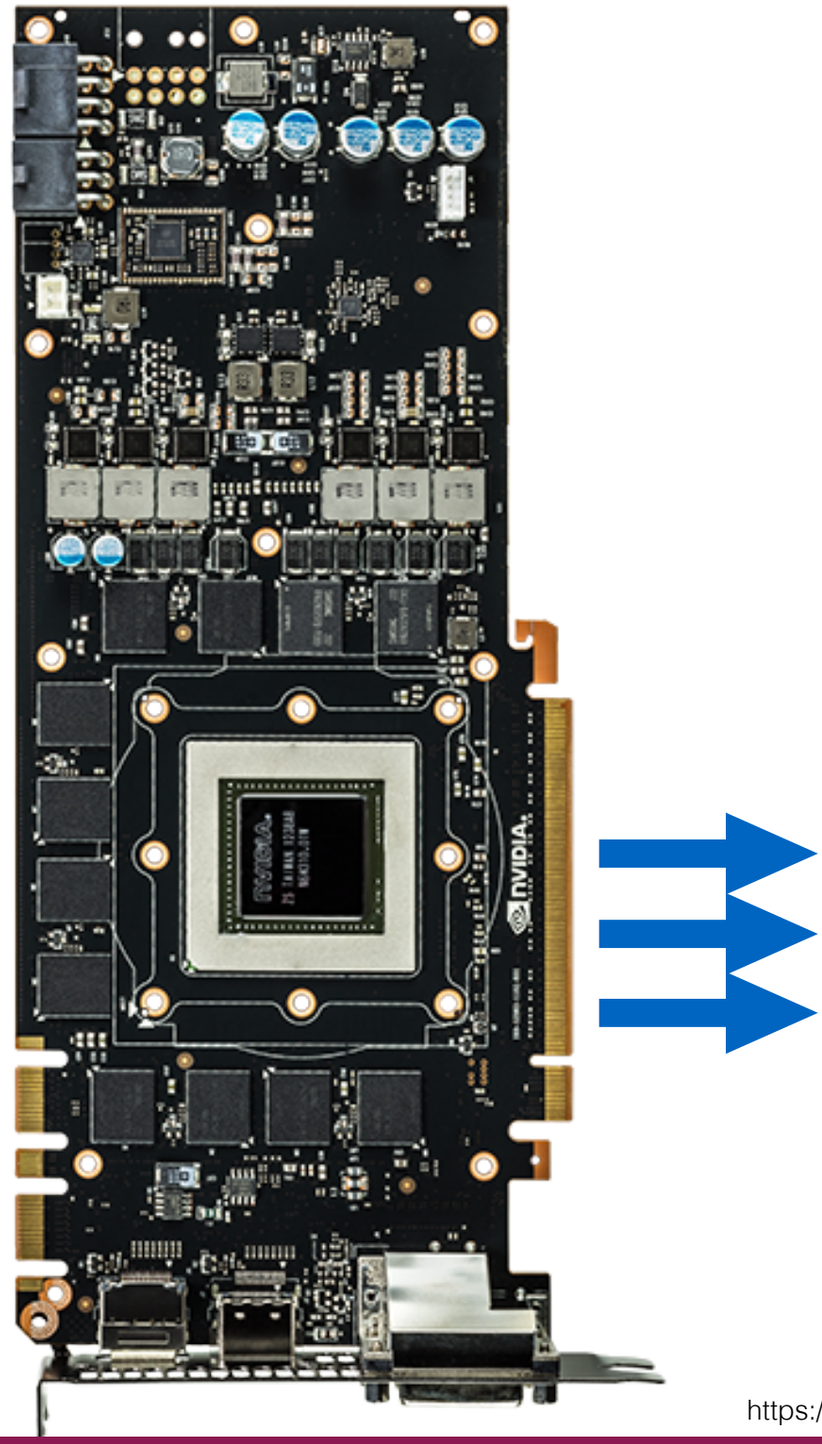
*Key observation: the DEVICE and HOST are asynchronous.*
*Operations are queued on the DEVICE.*

# 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

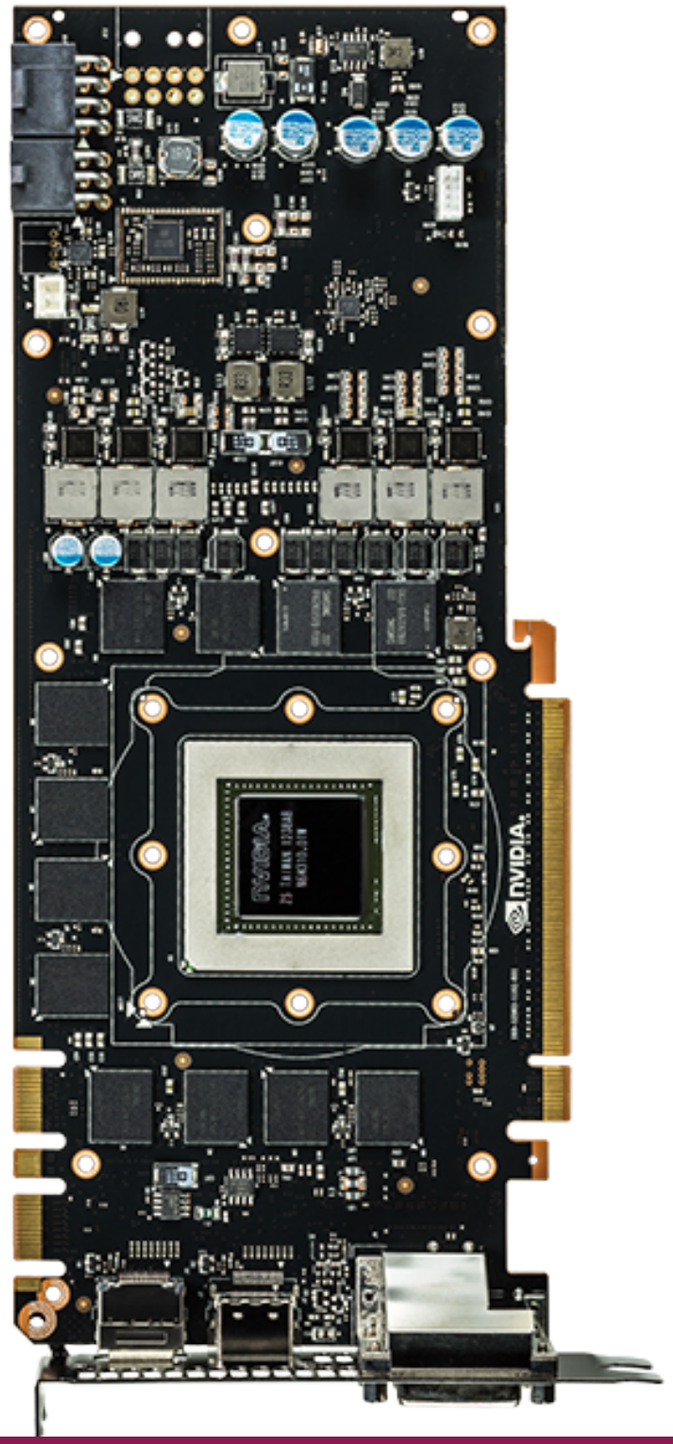
*Key observation: the DEVICE and HOST are asynchronous.*
*Operations are queued on the DEVICE.*

# 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 decides how to partition a compute task into kernel tasks:
  - Programmer then decides how to partition a kernel into independent work-blocks:
    - Work-block assigned to a core with sufficient resources to process it:
      - Each core processes the work-block 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:

2. Design thread-array:

3. Queue compute task on DEVICE:

4. Copy results from DEVICE to HOST:

*Key API calls: cudaMalloc, cudaMemcpy*

# 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:

3. Queue compute task on DEVICE:

4. Copy results from DEVICE to HOST:

*Key API calls: cudaMalloc, cudaMemcpy*

# 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:

4. Copy results from DEVICE to HOST:

*Key API calls: cudaMalloc, cudaMemcpy*

# 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:

*Key API calls: cudaMalloc, cudaMemcpy*

# 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

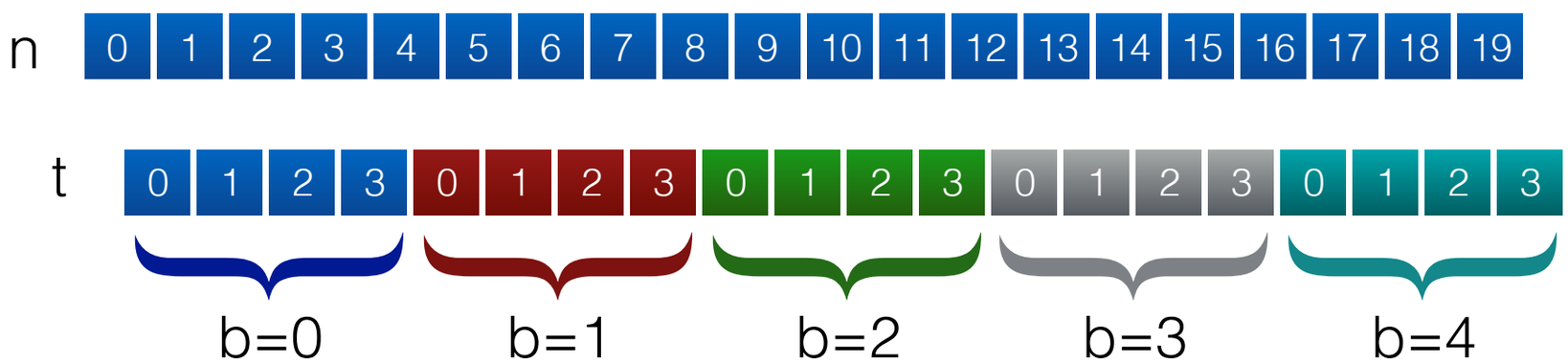
```
void serialSimpleKernel(int N, float *d_a){

  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;

  }
}
```

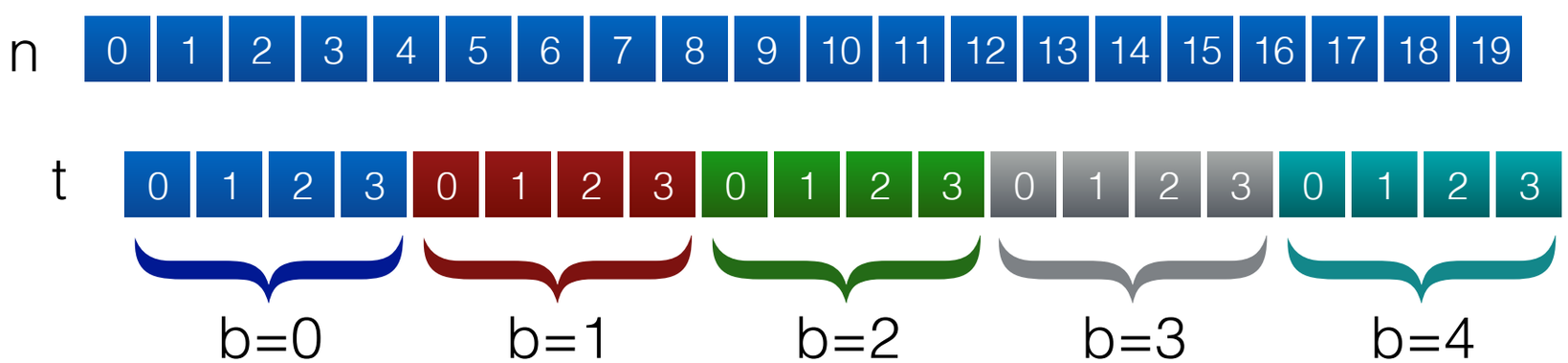



*We can think of splitting the n-loop into tiles of size 4: n=t+4b.*
*Here: block dimension = 4 and grid dimension = 5.*

# 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 <= gridDim*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;blockId.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 <= gridDim.x*blockDim.x*

# 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;blockId.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 <= gridDim.x*blockDim.x*

# 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;blockId.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 <= gridDim.x*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, gridDim

```
void tiledSerialSimpleKernel(int N, float *d_a){

  for(blockIdx.x=0;blockId.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 <= gridDim.x*blockDim.x*

# 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;blockId.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 <= gridDim.x*blockDim.x*

# 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;blockId.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 <= gridDim.x*blockDim.x*

# CUDA: simple array operation kernel

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

```
__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;
}
```

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: 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;
}
```

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: simple array operation kernel

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

*__global__ specifies kernel*

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

```
__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;
}
```

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: simple array operation kernel

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

*__global__ specifies kernel*

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

```
__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;
}
```

*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.*

# CUDA: array operation host code

GitHub, Inc. github.com/tcew/A

37 lines (25 sloc) | 970 Bytes

```
#include "cuda.h"

#include <stdio.h>
#include <stdlib.h>

__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;
}

int main(int argc,char **argv){
  int N = 512; // size of array for this DEMO

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

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

Note:
a. dimBlock does not need to be a multiple of 32.
b. CUDA will pad to multiple of 32
c. excess threads will be idled.

*TW: go to code on github for walk through:*
https://github.com/tcew/ATPESC17/blob/master/examples/cuda/simple/main.cu

# CUDA: code samples @ github

GitHub, Inc. github.com/tcew/ATPESC16/tree/master/examples/cuda/simple

This repository Search Pull requests Issues Gist

tcew / ATPESC16

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Branch: master ATPESC16 / examples / cuda / simple / Create new file Upload files Find file History

Tim Warburton adding material for ATPESC16 Latest commit db6ca0c 7 days ago

..

| | | |
|---|---|---|
| makefile | adding material for ATPESC16 | 7 days ago |
| simple.cu | adding material for ATPESC16 | 7 days ago |

© 2016 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

Make sure you can compile and run this code (see next slide) !

*github repo: https://github.com/tcew/ATPESC17*
*See: examples/cuda/simple*

# 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/ATPESC17

# if you haven't already done so, queue an interactive job request:
qsub -A ATPESC17 -I -n 1 -t 120 -q training

# find the source
cd ATPESC17/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/ATPESC17/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;blockId.z<gridDim.z;++blockIdx.z){     // loop over z-blocks
   for(blockIdx.y=0;blockId.y<gridDim.y;++blockIdx.y){   // loop over y-blocks
     for(blockIdx.x=0;blockId.x<gridDim.x;++blockIdx.x){ // loop over x-blocks

       // loop over thread indices in thread-block
       for(threadIdx.z=0;threadIdx.y<blockDim.z;++threadIdx.z){
         for(threadIdx.y=0;threadIdx.z<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: 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:

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

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

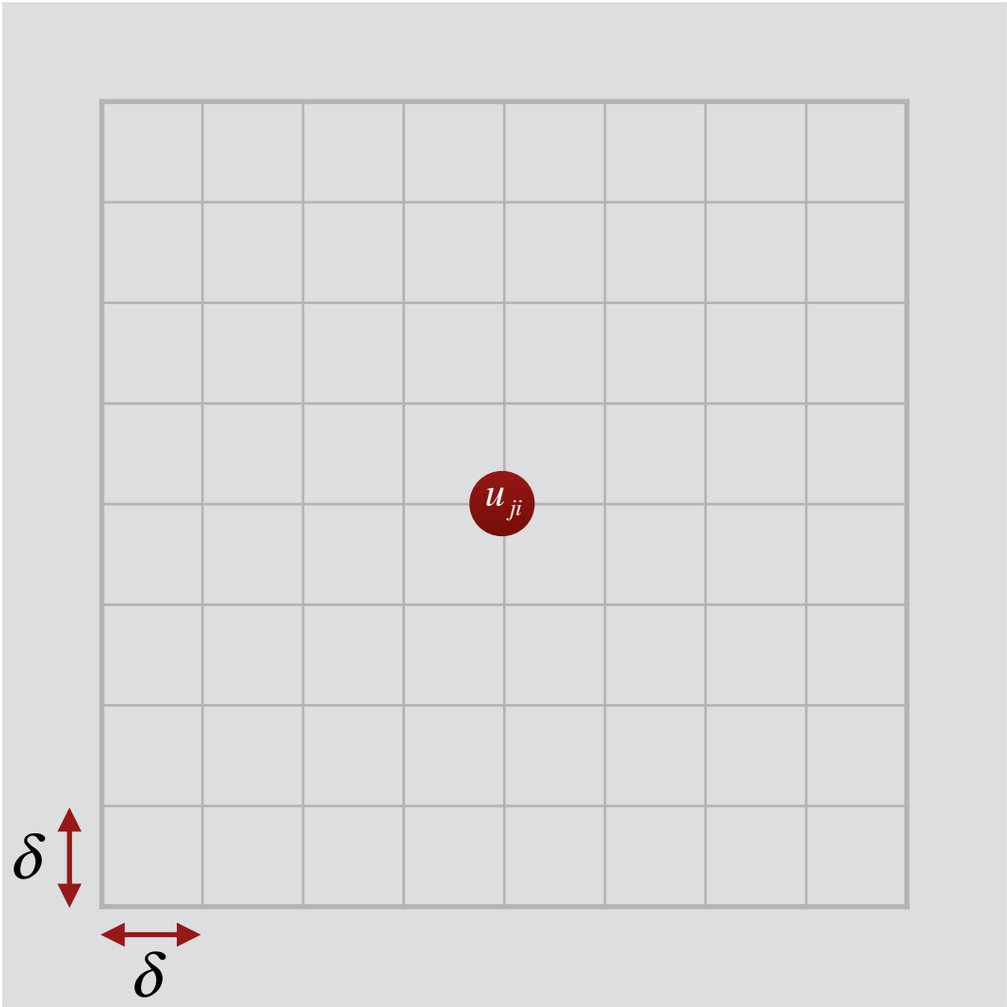
*Poisson problem is an archetypal building block for many physics packages.*

# 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:

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

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

Discrete Poisson problem (assuming Cartesian grid):

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

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

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

# CUDA: elliptic solver example

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

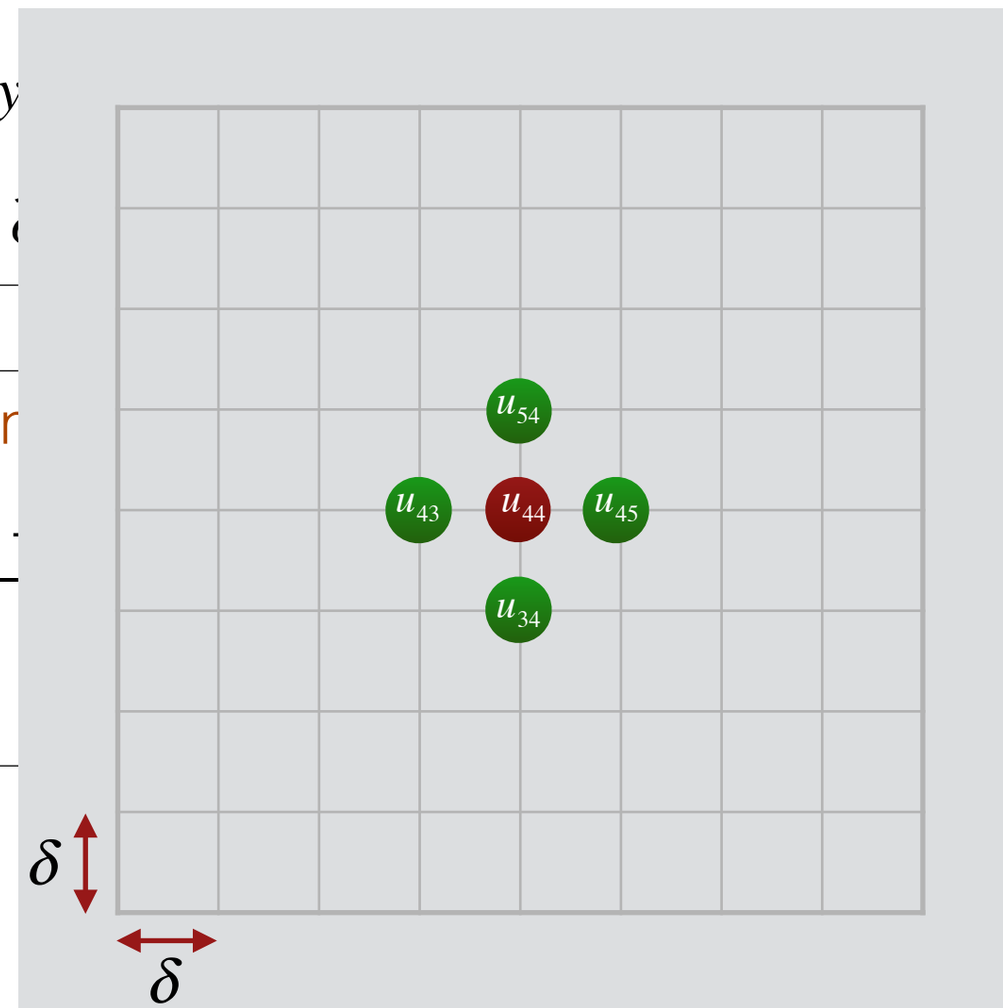
Elliptic Poisson problem:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y$$

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

Discrete Poisson problem (assumir

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




*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):

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

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

Jacobi iteration for discrete Poisson problem:

$$\left(\frac{u^k_{j(i+1)} - 2u^{k+1}_{ji} + u^k_{j(i-1)}}{\delta^2}\right) + \left(\frac{u^k_{(j+1)i} - 2u^{k+1}_{ji} + u^k_{(j-1)i}}{\delta^2}\right) = f_{ji} \quad \text{for } i, j = 1, \ldots, N$$

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

*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:

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

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

Iterate:

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

while:

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

*Source code: https://github.com/tcew/ATPESC17/examples/cuda/simple*

# CUDA: parallelism for solver example

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

Iterate:

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

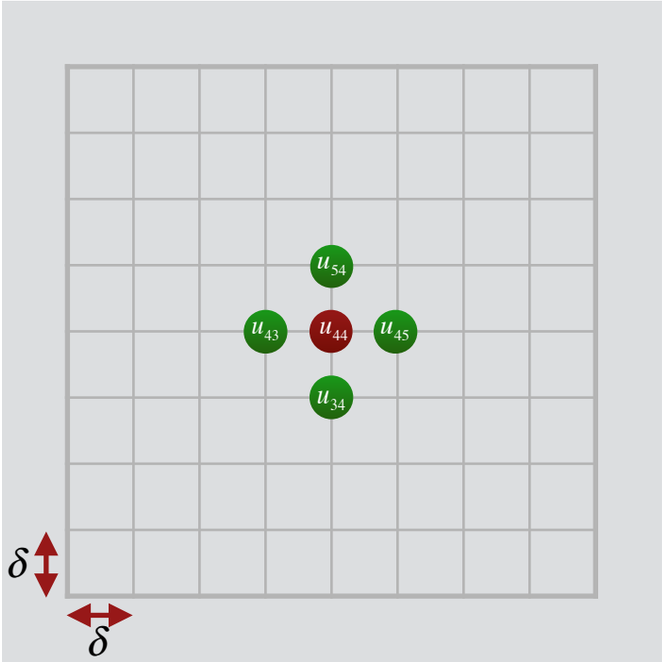
*The GPU works best when every thread is doing the same thing.*

# CUDA: parallelism for solver example

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

Iterate:

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




*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:

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

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:

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

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/ATPESC17/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!!

Thread-block tree reduction in pseudo-code:

```
t = thread index in thread block;
alive = number of threads in thread block;
s_sumu[t] = u[global thread index];

while(alive>1){

  synchronize threads in thread-block;

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

if(t==0) blocksumu[block index] = s_u[0];
```

| Thread \ Step | 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];
}
```

| Thread \ Step | 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];
}
```

| Thread \ Step | 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];
}
```

| Thread \ Step | 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];
}
```

| Thread \ Step | 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!!

| Thread \ Step | 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];
}
```

| Thread \ Step | 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

GitHub, Inc. github.com/tcew/ATPESC16/tree/master/examples/cuda/reduction

This repository Search Pull requests Issues Gist

tcew / ATPESC16

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Branch: master ATPESC16 / examples / cuda / reduction / Create new file Upload files Find file History

tcew fixed errors and typos Latest commit 924805f a minute ago

| | | |
|---|---|---|
| .. | | |
| main.cu | fixed errors and typos | a minute ago |
| makefile | adding standalone reduction cuda example | 10 minutes ago |

© 2016 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

*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/ATPESC17

**#2. Complete the skeleton code**

ATPESC17/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 if :**

*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 Batcher*

*Attribution is a little cloudy: *possibly Seymour Cray*

# … 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 | ? | ? |

*Adapted from Timothy Lanfear's CUDA Tutorial Slides*

# 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

A1 | CUDA GPU Occupancy Calculator

## CUDA GPU Occupancy Calculator

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

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 |

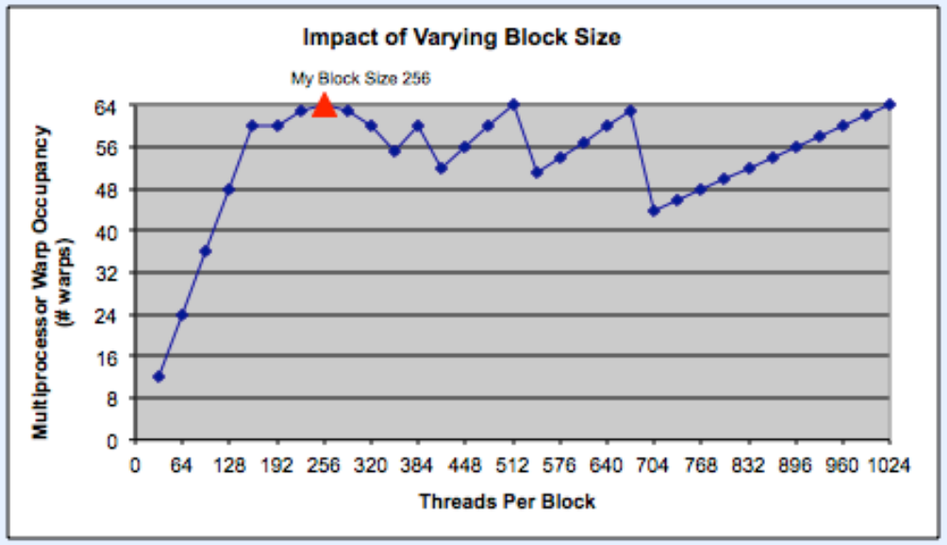
Note: SM is an abbreviation for (Streaming) Multiprocessor

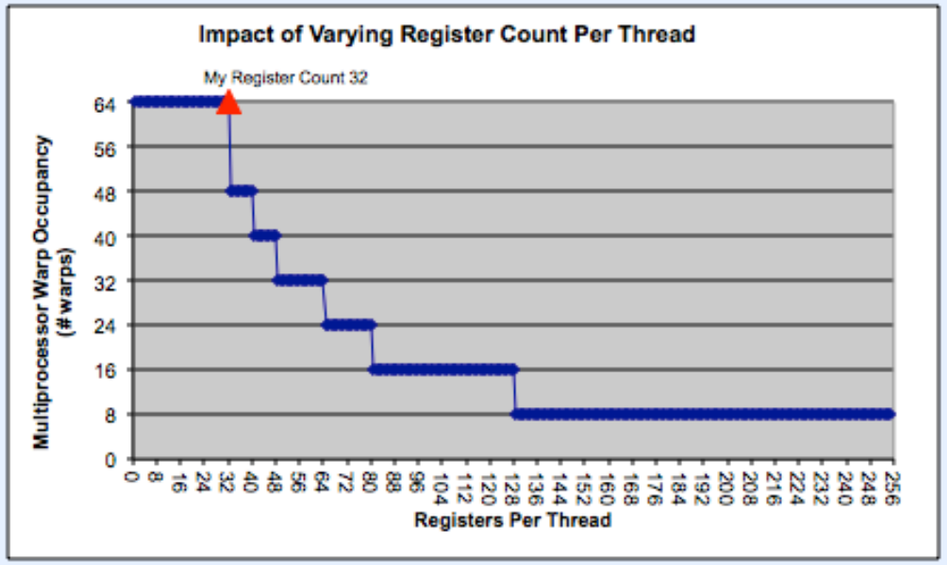
| Maximum Thread Blocks Per Multiprocessor | Blocks/SM | * Warps/Block | = Warps/SM |
|---|---|---|---|
| Limited by Max Warps or Max Blocks per Multiprocessor | 8 | 8 | 64 |
| Limited by Registers per Multiprocessor | 8 | 8 | 64 |
| Limited by Shared Memory per Multiprocessor | 12 | 8 | 96 |

Note: Occupancy limiter is shown in orange

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

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 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.*

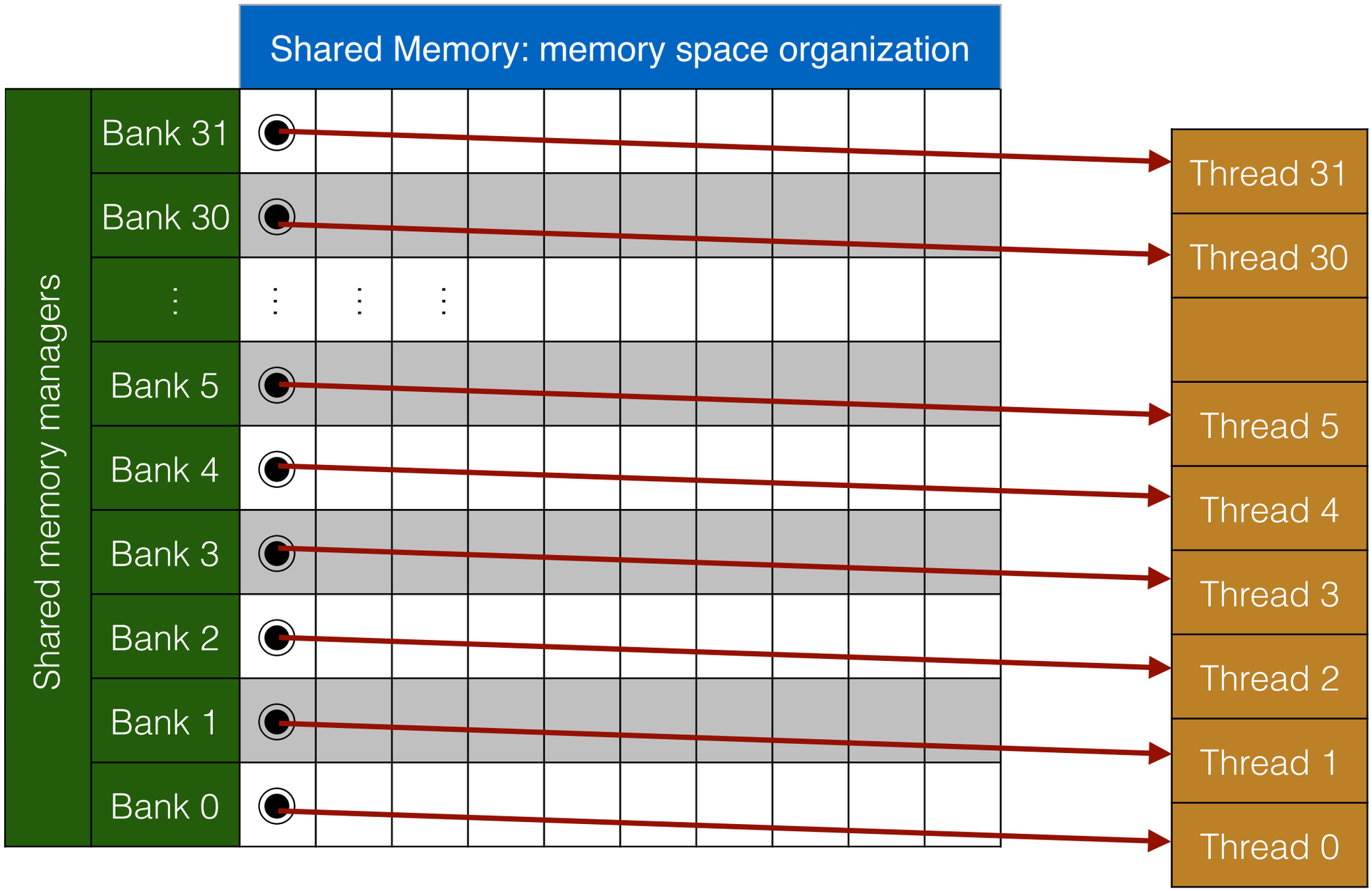
# CUDA: shared memory banks

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

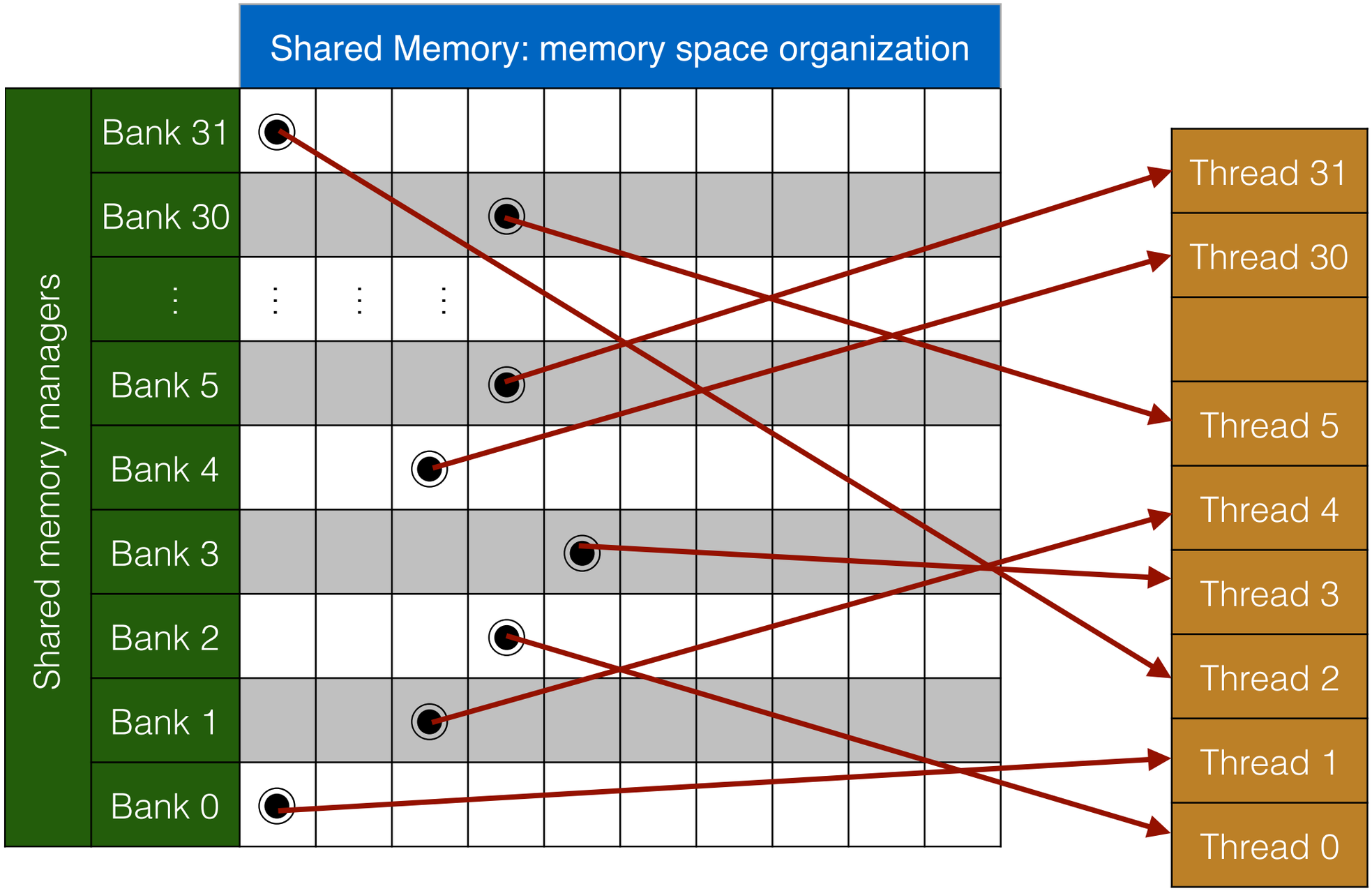
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.




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

# CUDA: shared memory banks

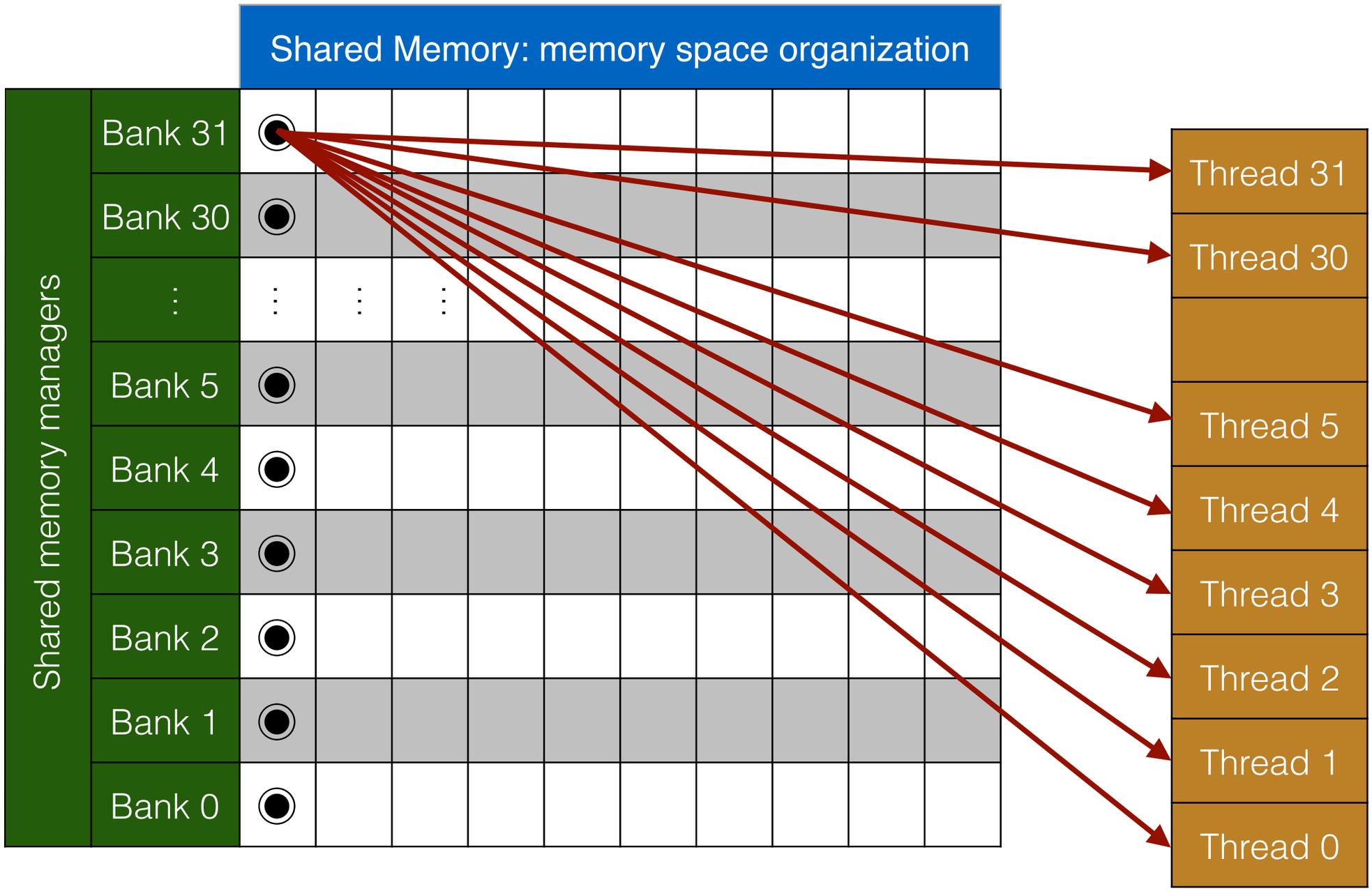
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.




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

# CUDA: shared memory broadcast

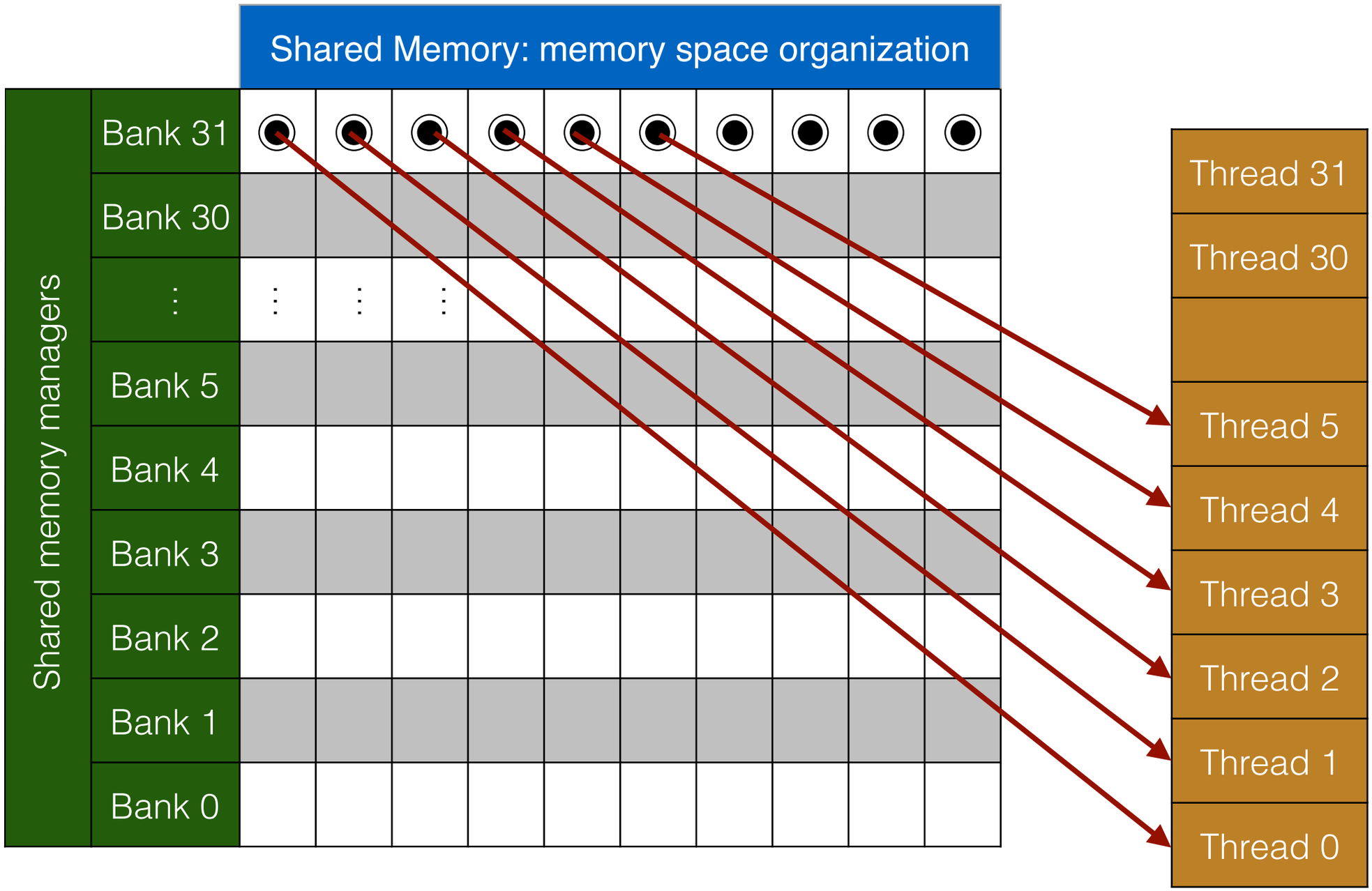
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.




*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.



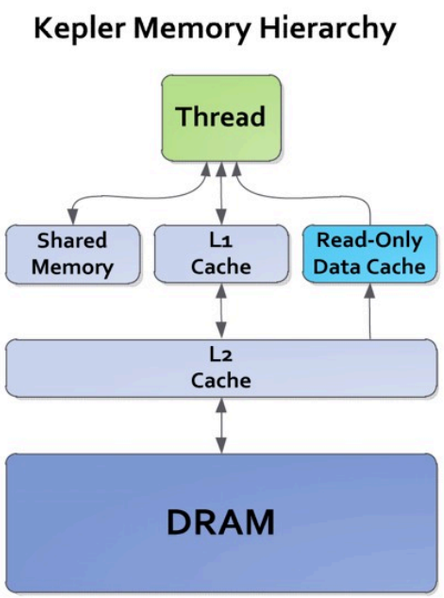

*BAD: all threads in the SIMD group access the same bank resulting in serialization.*

# 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.

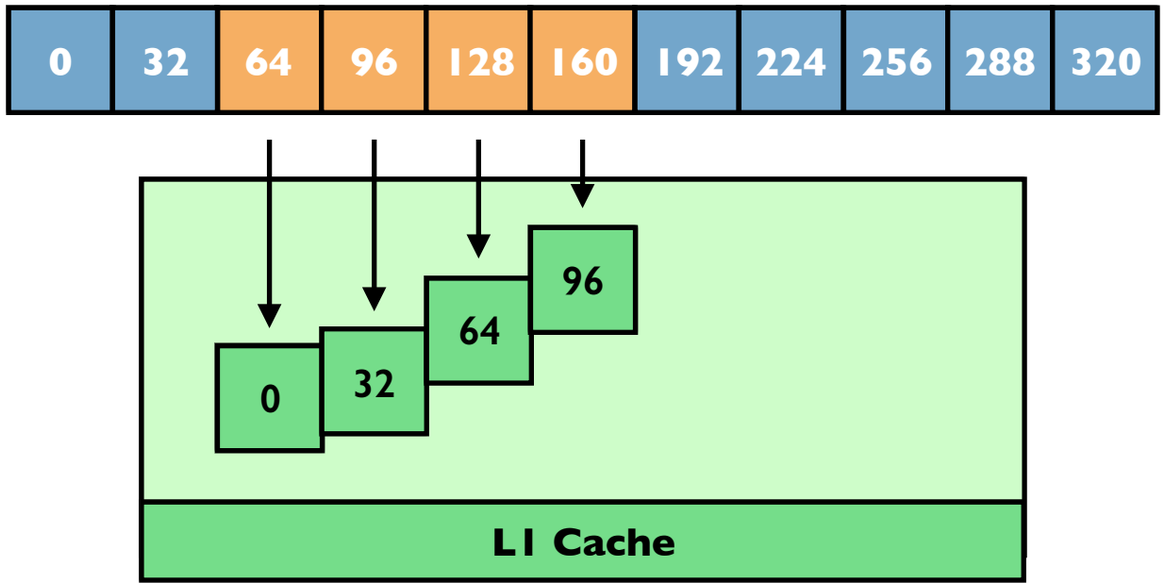



*Rule of thumb: avoid non unitary stride DEVICE (DRAM) array access.*
*Useful slides , these , and image credit: link*

# CPU Optimization Techniques

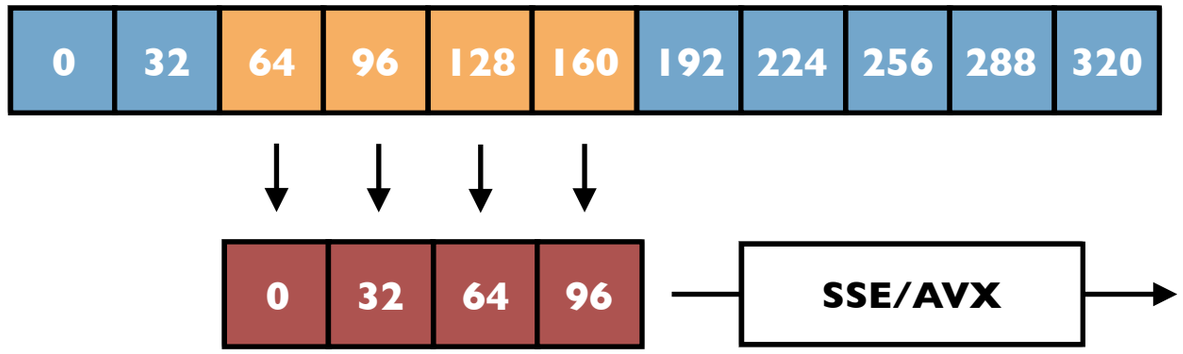
## Cache

- 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.

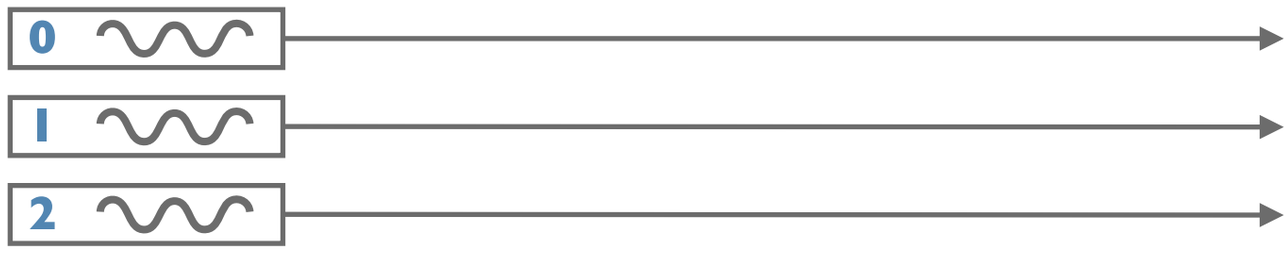



*Continuous memory accesses are used for both, cache storage and vectorization*

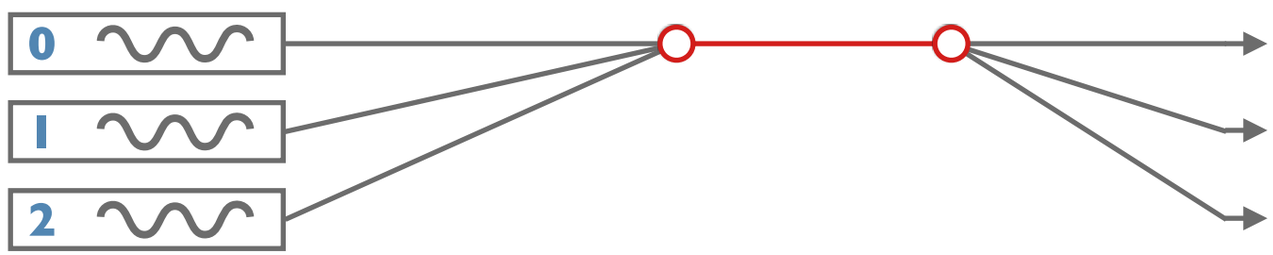
# CPU Optimization Techniques

## Multithreading

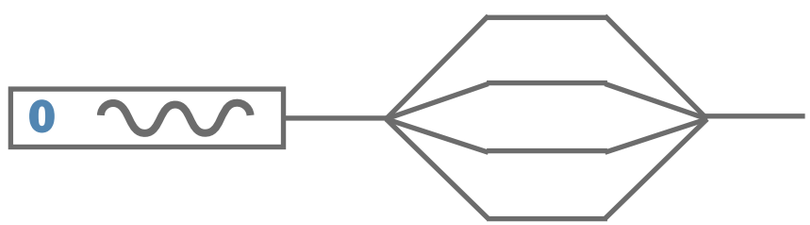
- Threads capable of fully parallelizing generic instructions (ignoring bandwidth).




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



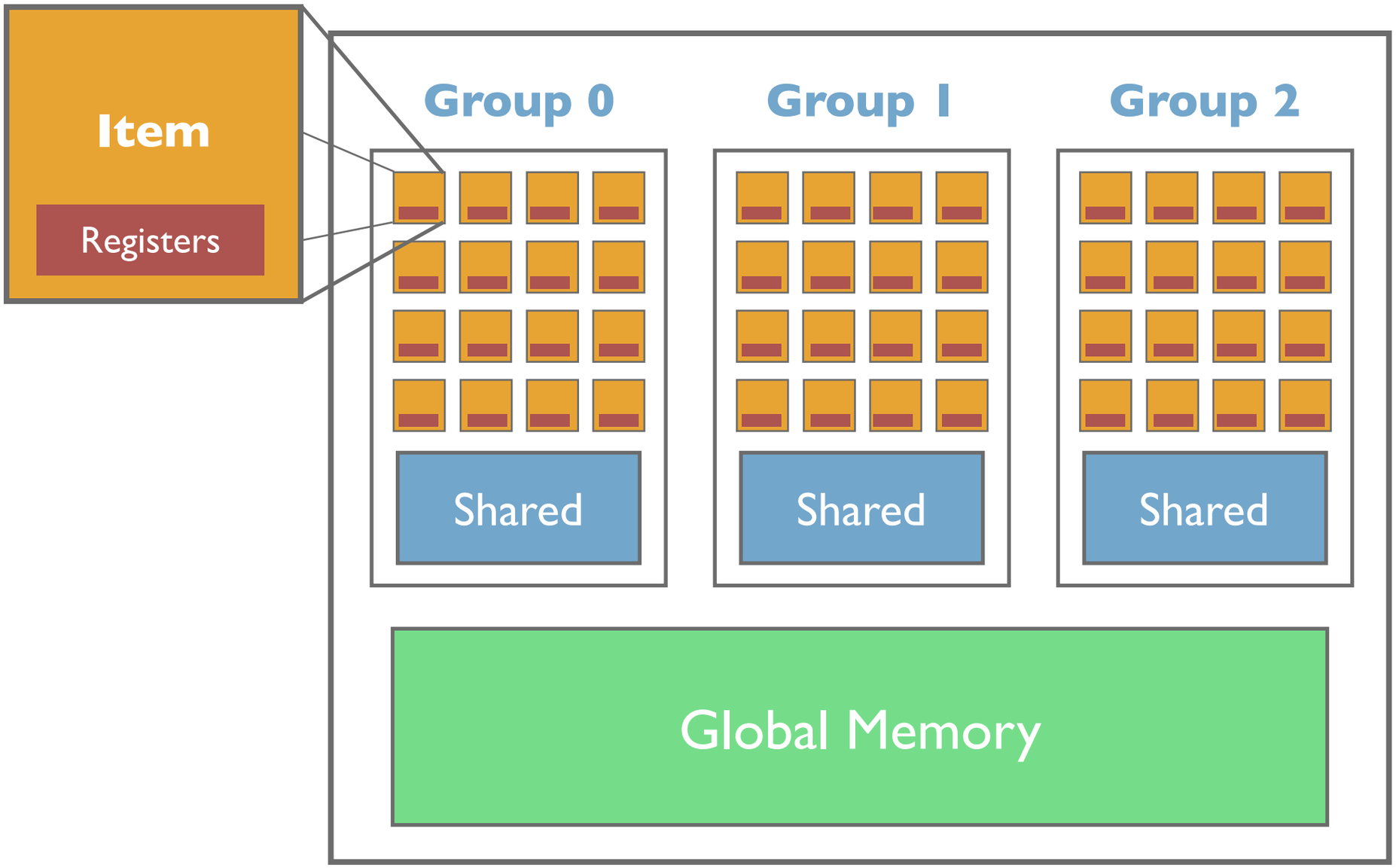

- SIMD Lanes



# GPU Optimization Techniques

## GPU Architecture

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

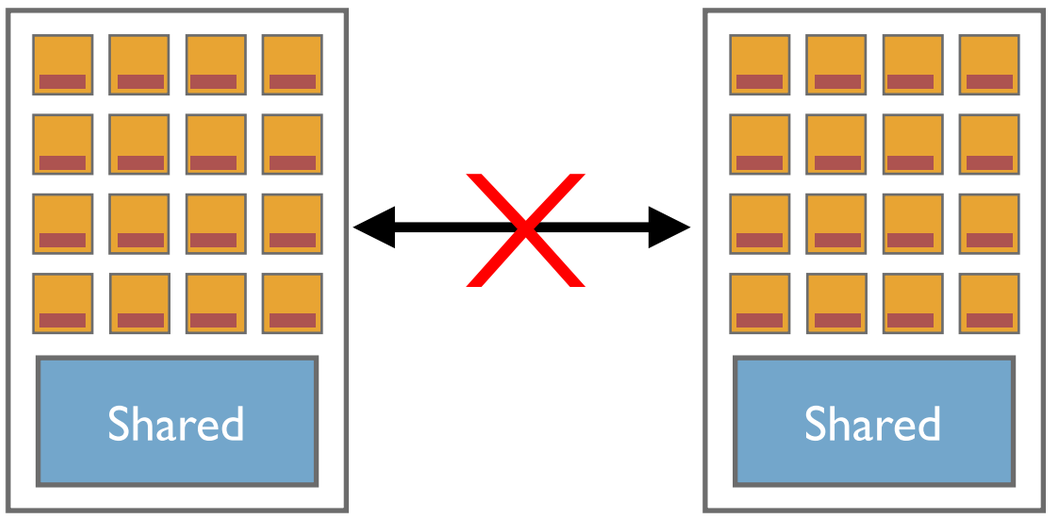



*Kernel code describes the work-item operations*

# GPU Optimization Techniques

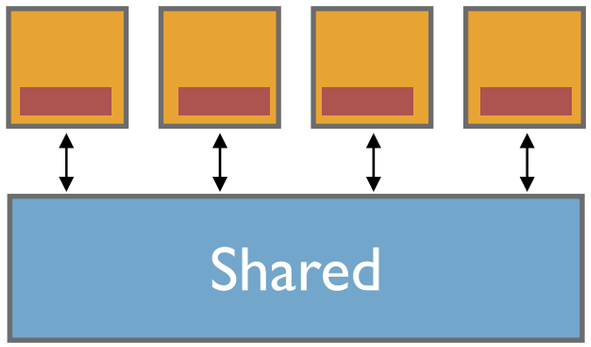
## Work-groups

- 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

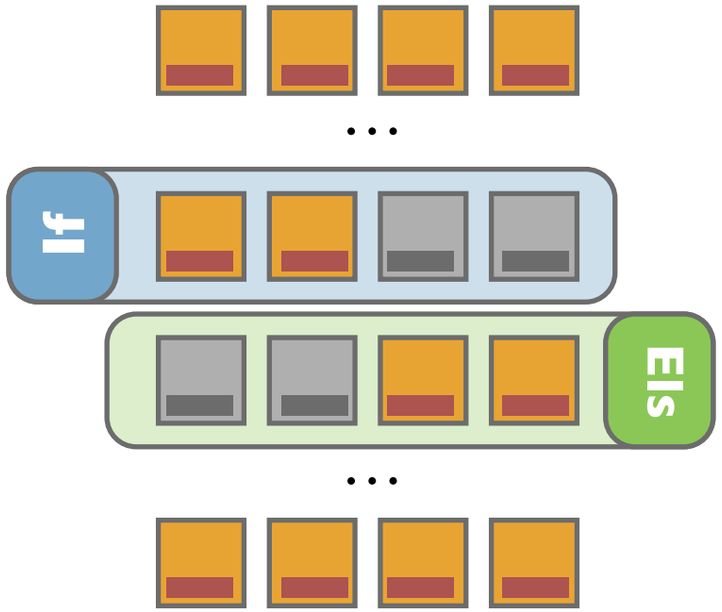
- Work-items are executed in parallel, able to barrier and share data using shared memory (& CUDA's shuffle).
- Avoid data race dependencies between work-items.



# GPU Optimization Techniques

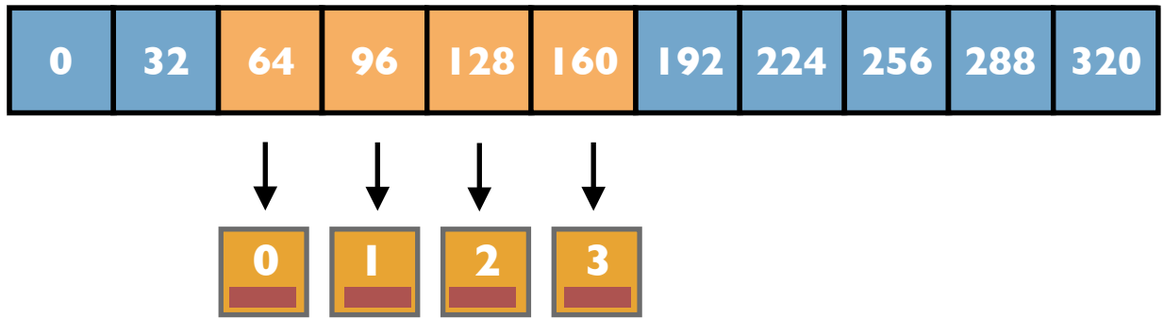
## Parallel Work-item Execution

- 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

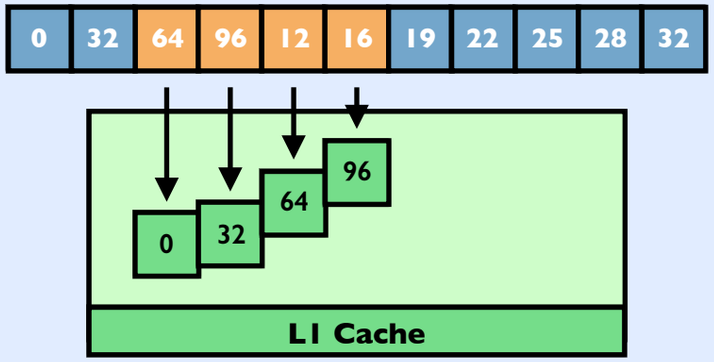
- Low individual bandwidth and high latency.
- Coalesced memory access on contiguous and aligned work-items.



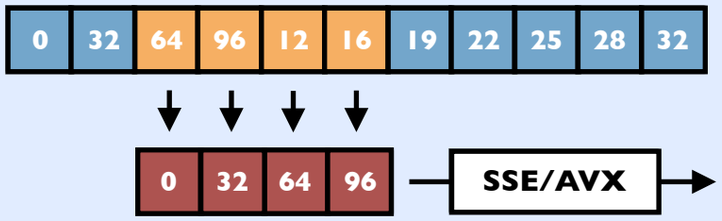
# CPU & GPU Similarities

## CPU Optimizations

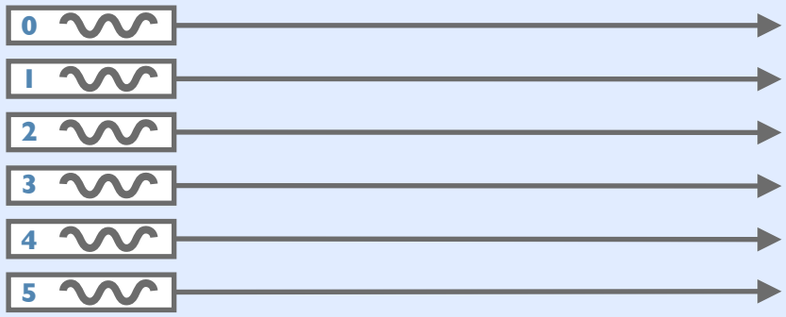
### Cache



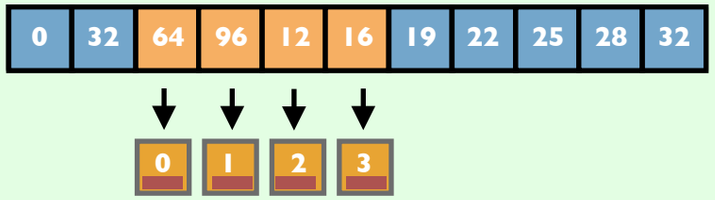

### Vectorization



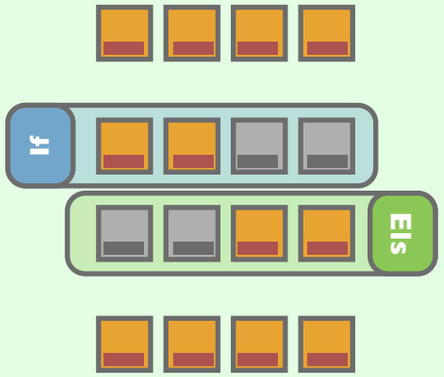

### Thread Independence



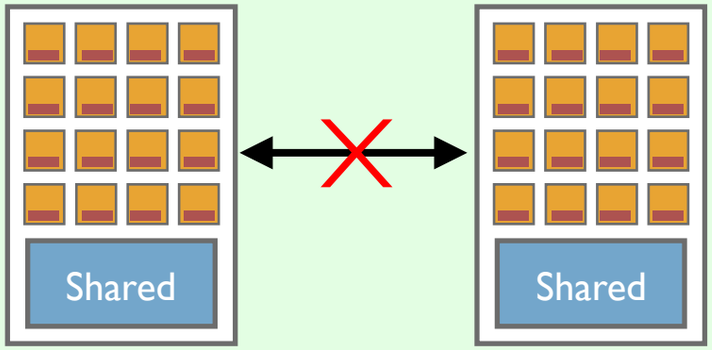

### Coalescing




### No Branching




### Work-group Independence




## GPU Optimizations

*Exposing vectorization / SIMD parallelism are vital in both architectures*

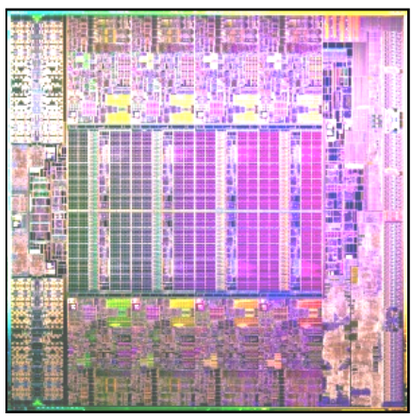
# Part 4: Portable programming models

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

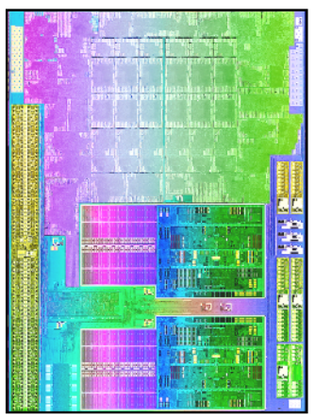
*Blinky was a MacBook Pro with discrete NVIDIA GPU and OpenMP, OpenCL, CUDA …*
*Latest MBP comes with AMD GPU and/or Intel Iris GPU, no CUDA, and default clang compilers omit OpenMP*

# Many-core: fragmentation

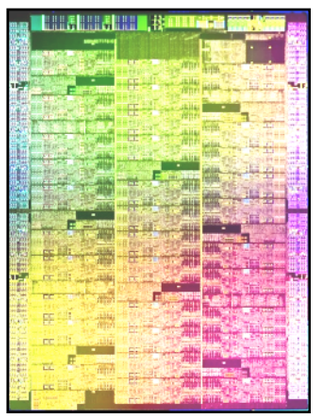
Zoo of competing architectures and programming models (with vendor bias)



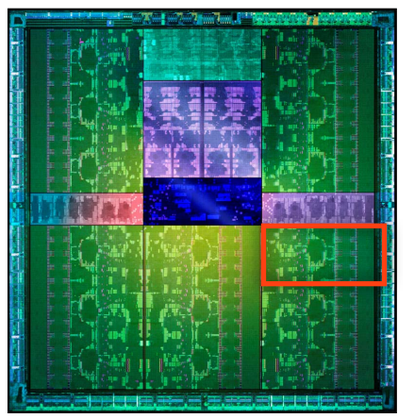
*Intel CPU*



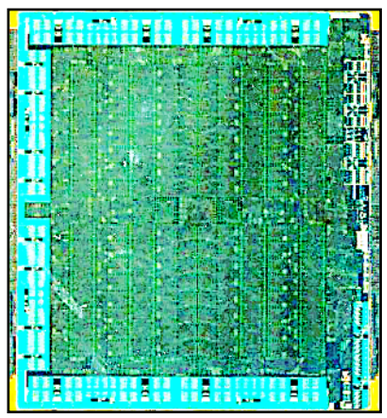
*AMD APU*



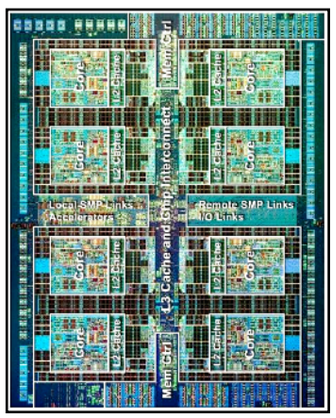
*Intel Xeon Phi*



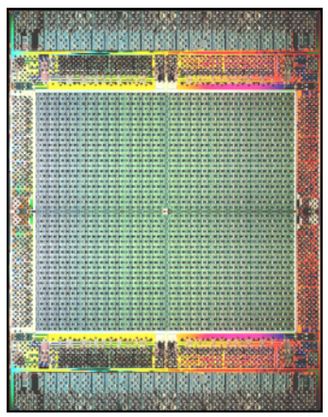
*NVIDIA GPU*



*AMD Tahiti GPU*




*IBM Power 7*

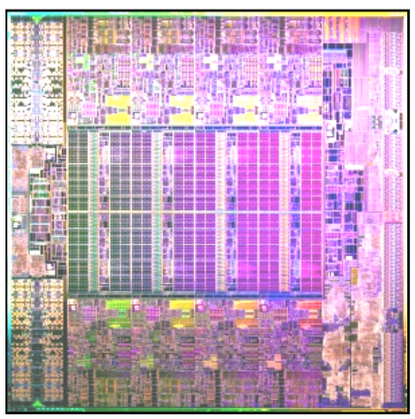


*Altera FPGA*

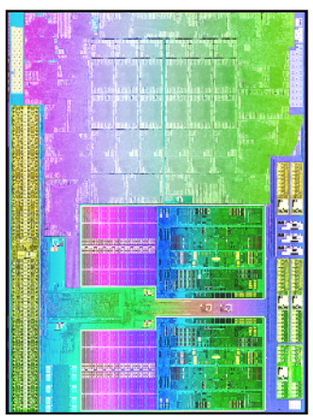
*Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming*

# Many-core: fragmentation

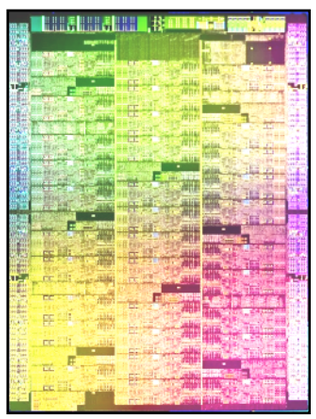
Zoo of competing architectures and programming models (with vendor bias)



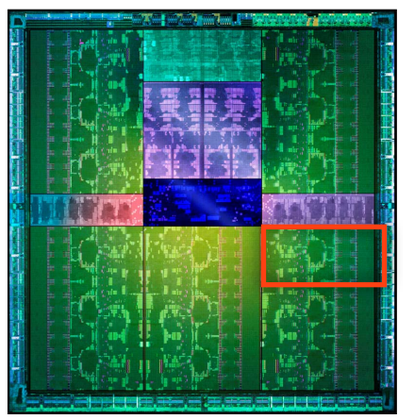
Intel CPU



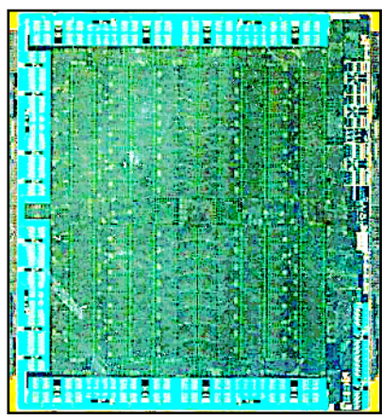
AMD APU



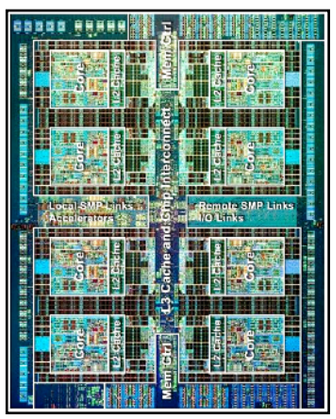
Intel Xeon Phi



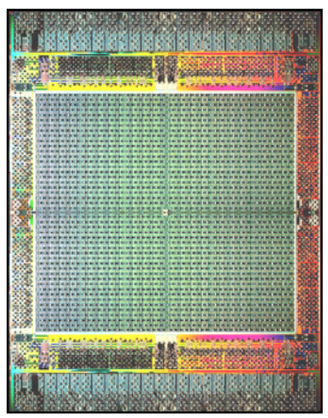
NVIDIA GPU



AMD Tahiti GPU




IBM Power 7



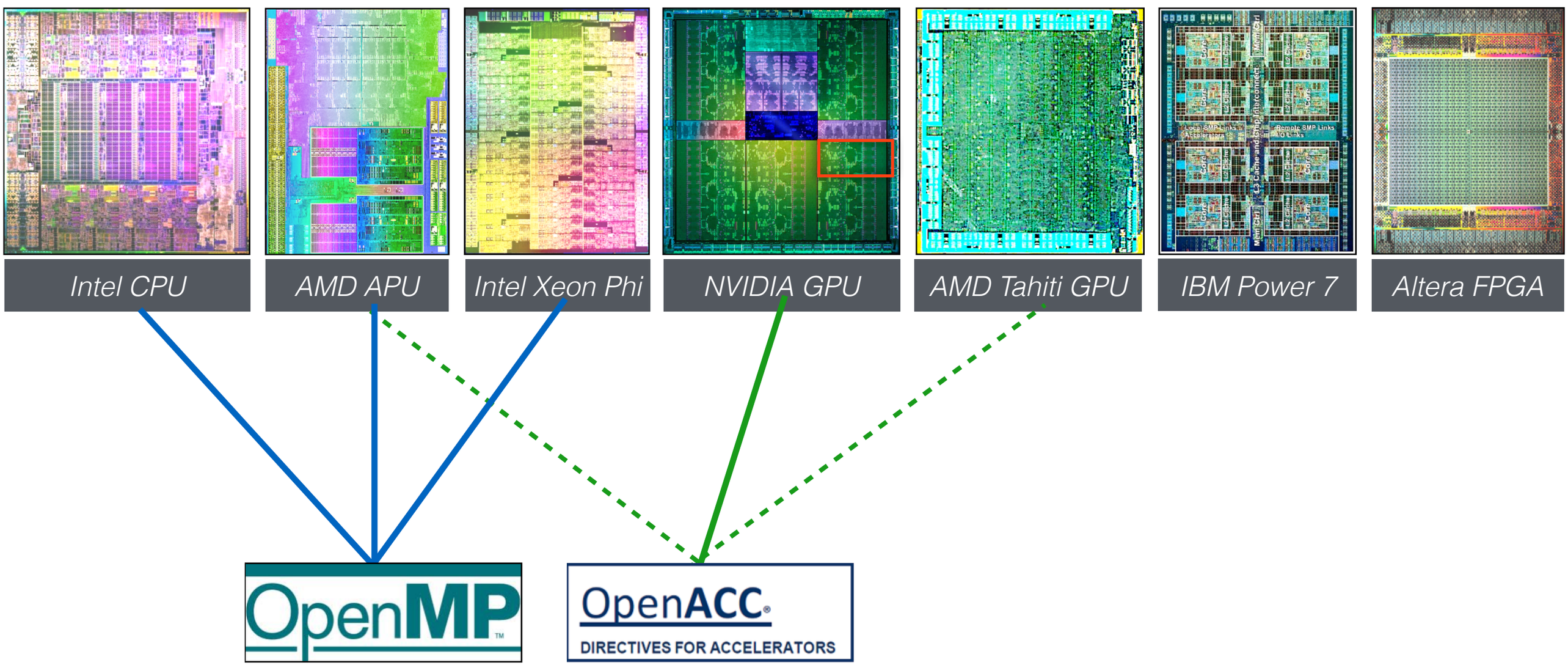
Altera FPGA

OpenMP

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

# Many-core: fragmentation

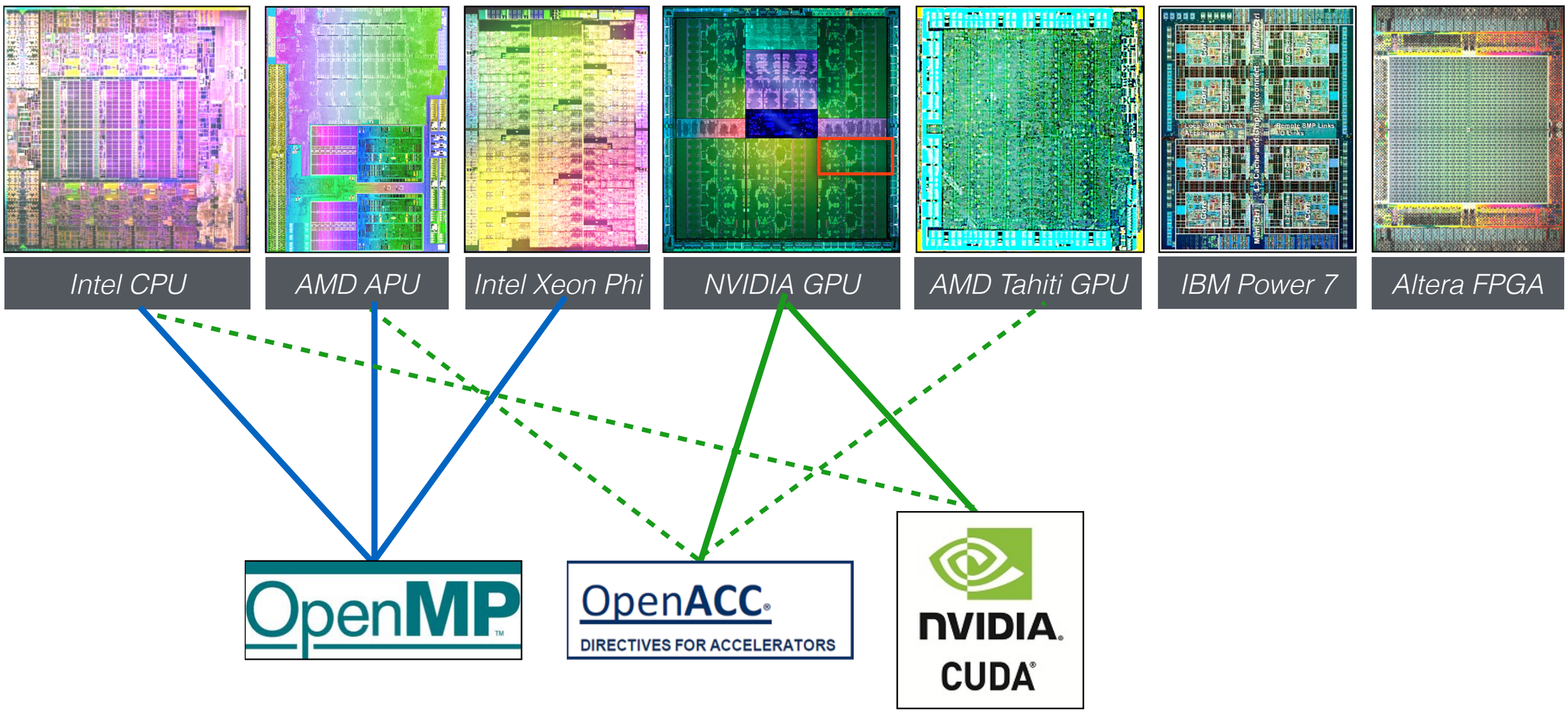
Zoo of competing architectures and programming models (with vendor bias)




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

# Many-core: fragmentation

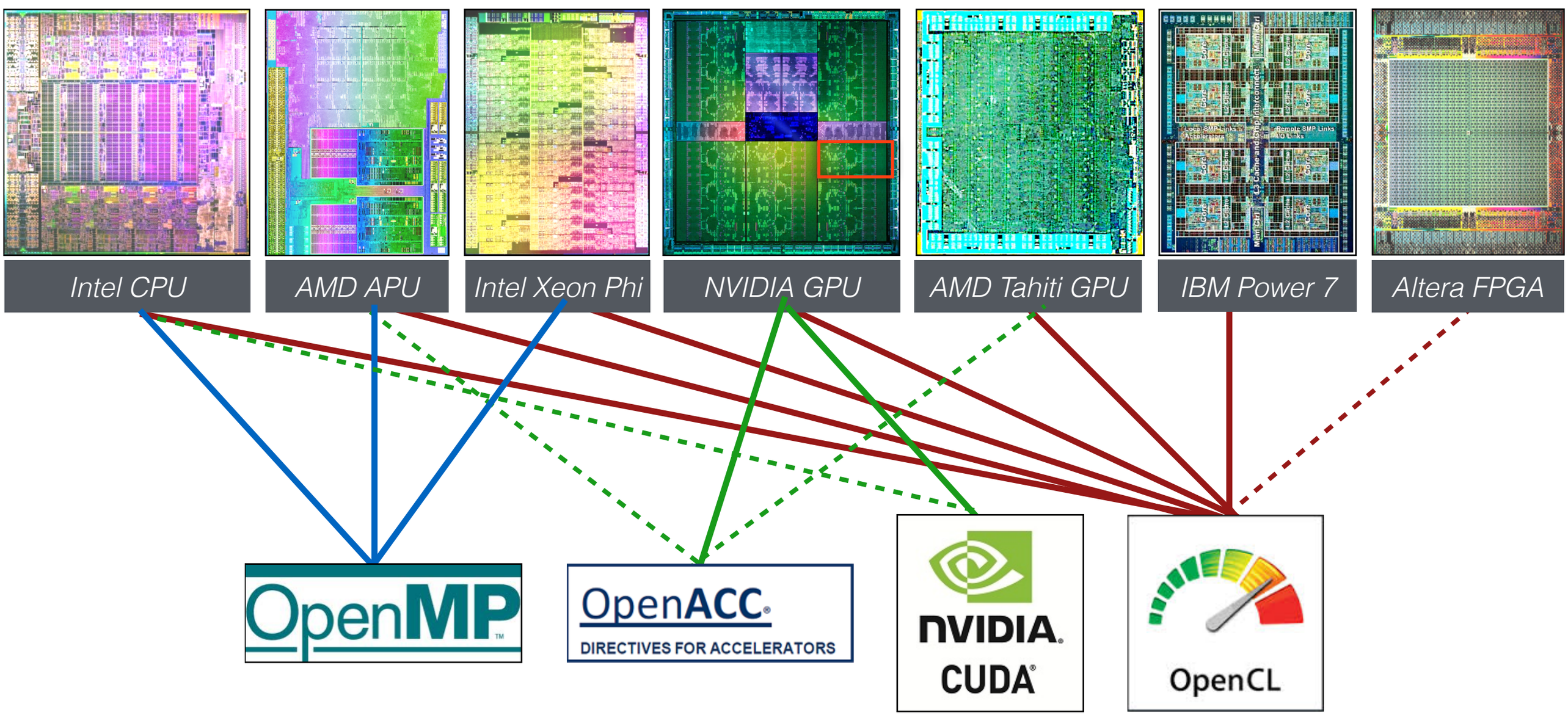
Zoo of competing architectures and programming models (with vendor bias)




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

# Many-core: fragmentation

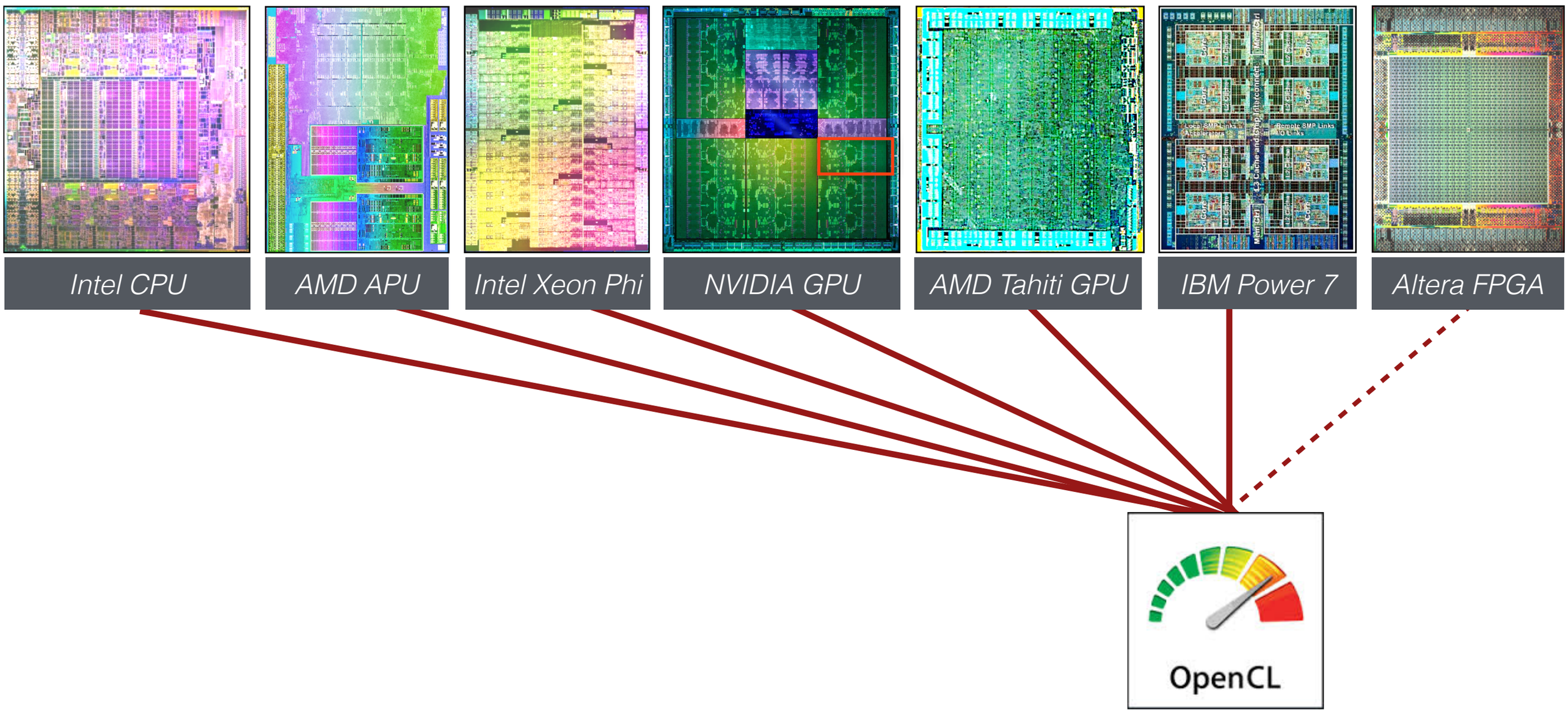
Zoo of competing architectures and programming models (with vendor bias)




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

# Many-core: fragmentation

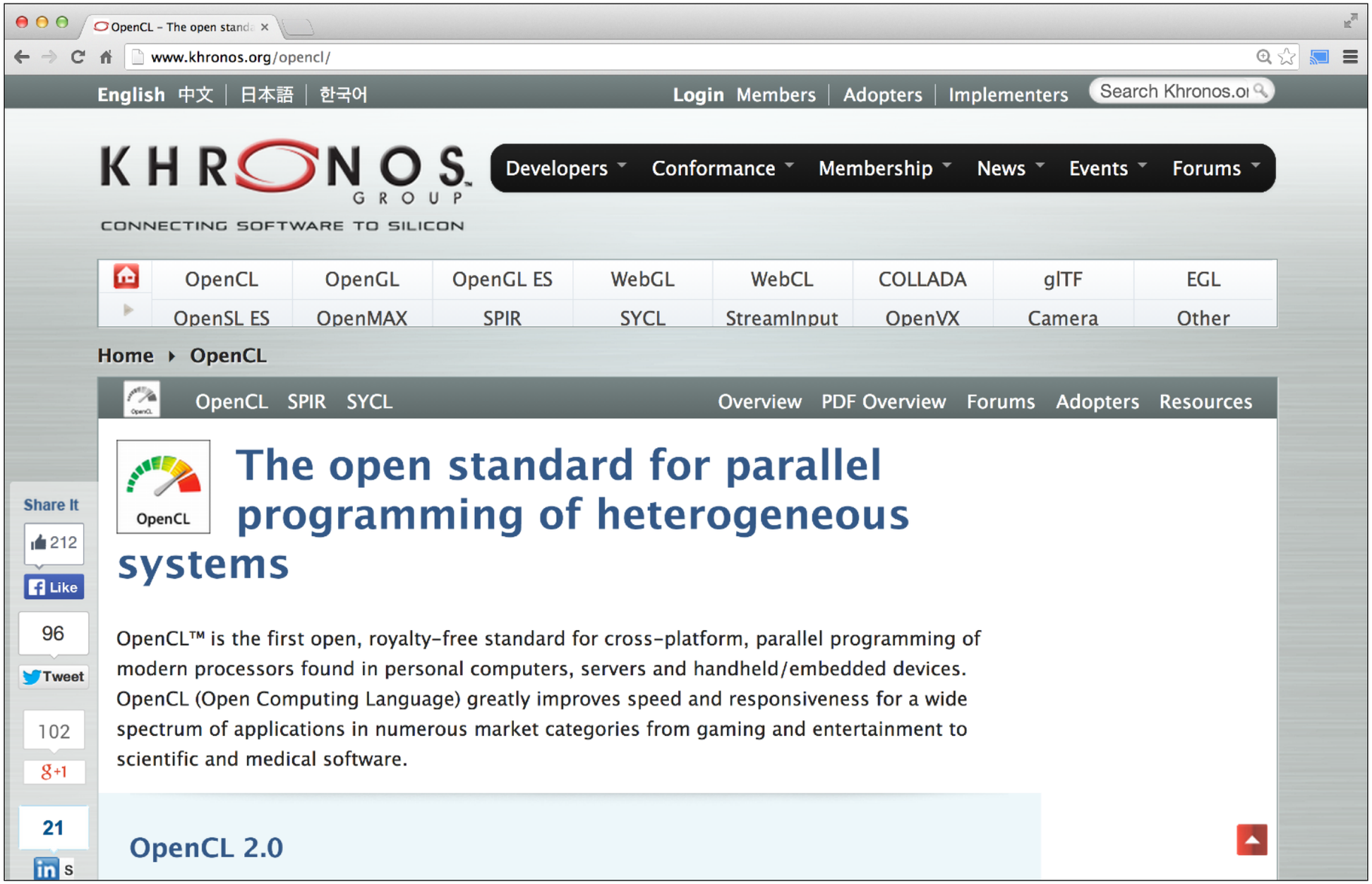
Zoo of competing architectures and programming models (with vendor bias)




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

# Part 4a: Open Compute Language (OpenCL)

# OpenCL: standards committee




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)

*The Khronos Group administers the OpenCL standard.*

# OpenCL: who ?

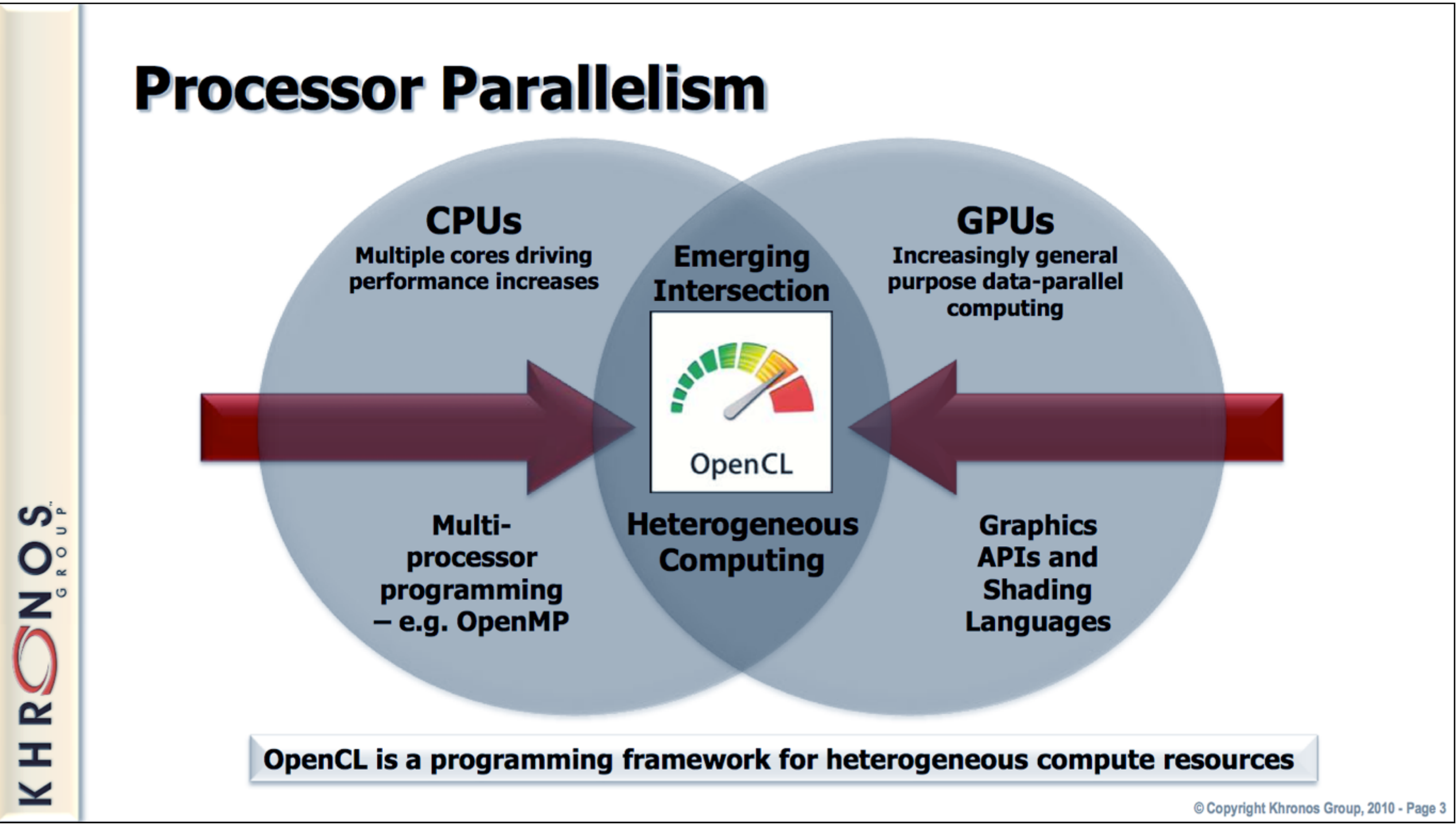
## OpenCL Working Group

- **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

© Copyright Khronos Group, 2010 - Page 4

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

# OpenCL: why ?



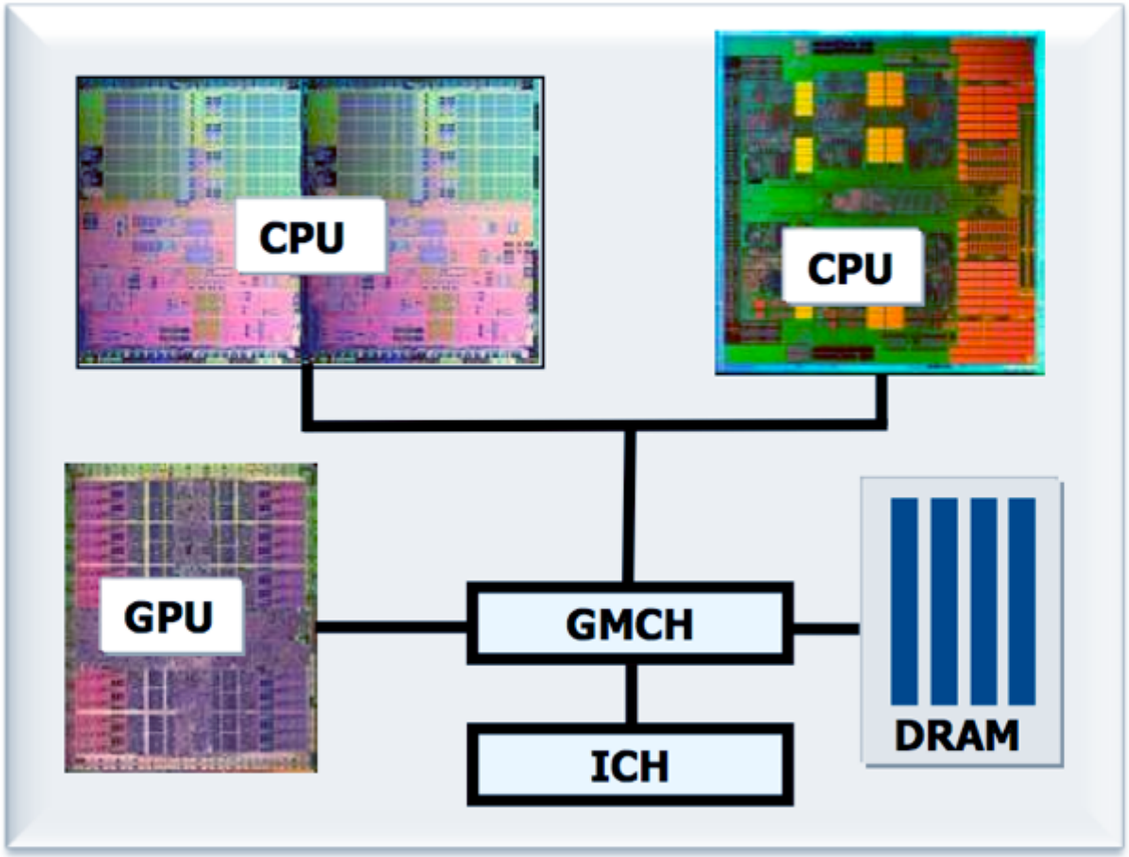

*Emphasis on heterogeneous computing.*
*Credit: Khronos Group*

# 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**



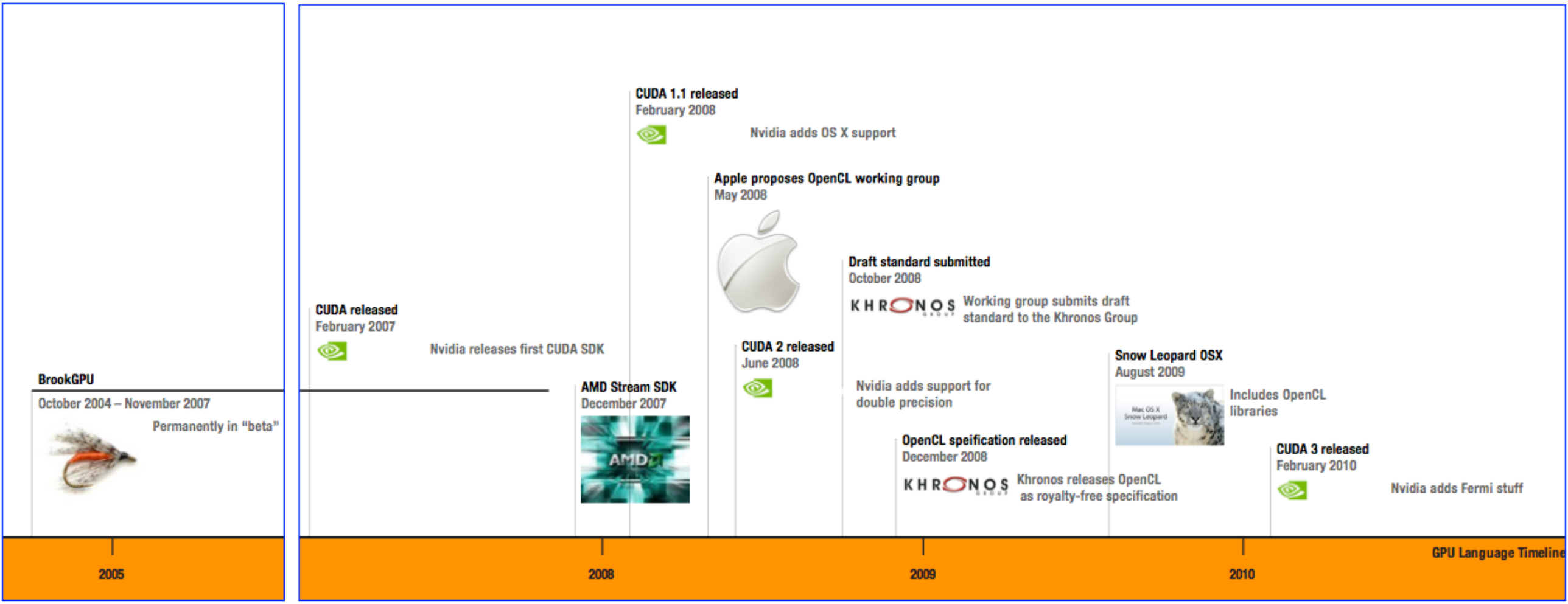

**GMCH = graphics memory control hub**
**ICH = Input/output control hub**

© Copyright Khronos Group, 2010 - Page 9

*Emphasis on heterogeneous computing.*
*Credit: Khronos Group*

# 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: | |
| blockIdx.x*blockDim.x + threadIdx.x | blockIdx.y*blockDim.y + 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. | get_global_size(0) |

*The rapid development of OpenCL helps explain the similarities*

# OpenCL: kernel language qualifiers

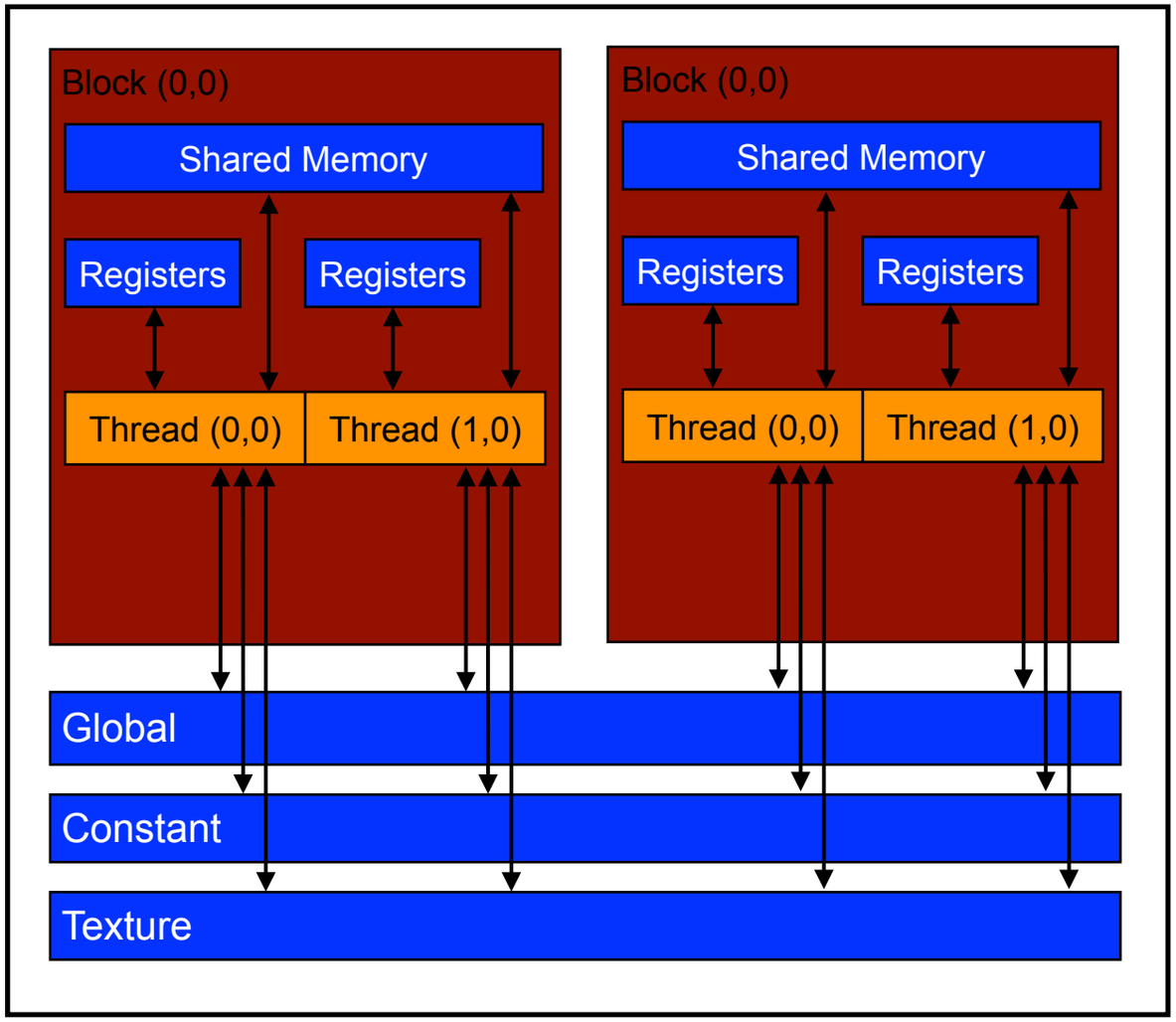
OpenCL is ** very ** closely related to CUDA

| CUDA | OpenCL |
|---|---|
| __global__ function | __kernel function |
| __device__ function | function |
| __constant__ variable | __constant variable |
| __device__ variable | __global variable |
| __shared__ variable | __local variable |

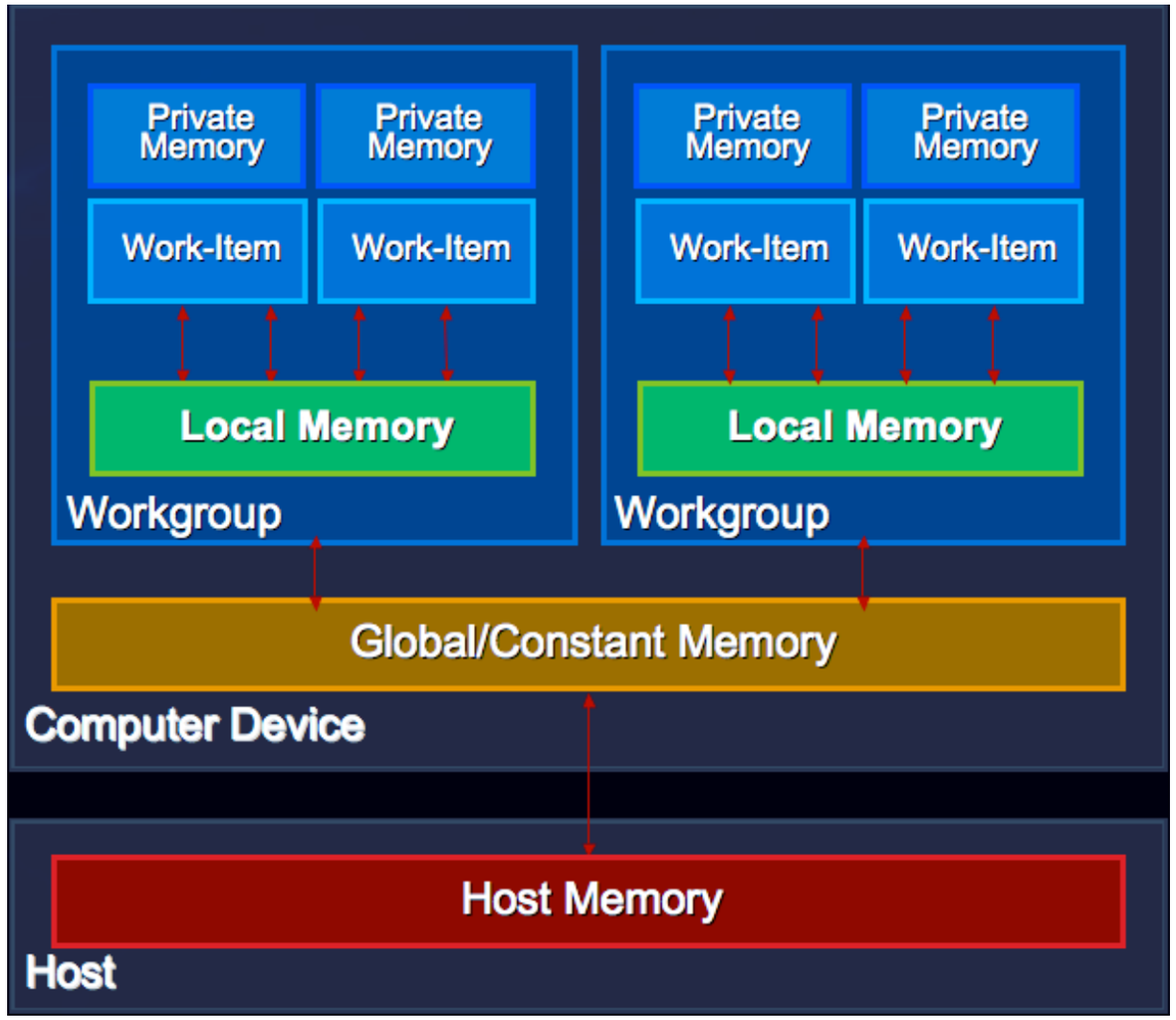
*The rapid development of OpenCL helps explain the similarities*

# 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”

CUDA

```
#include <cuda.h>

int main()
{

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

OpenCL

```
…
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.

CUDA

```
…

int dev = 0;
cudaSetDevice(dev);

…
```

OpenCL

```
…
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.

CUDA

```
…

// nada

…
```

OpenCL

```
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.

CUDA

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

OpenCL

```
// 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:

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

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

*I wasn’t kidding about flexibility.*

# OpenCL: building a kernel

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

CUDA

```
…

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

OpenCL

```
/* 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

# OpenCL: thanks for the memory

We next allocate array space on the DEVICE:

CUDA

```
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));
```

OpenCL

```
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.

CUDA

```
…

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);
…
```

OpenCL

```
/* 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:

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

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:

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

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!!

*More details on OpenCL reduction next time.*

# 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];
}
```

*More details on OpenCL reduction next time.*

# 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:

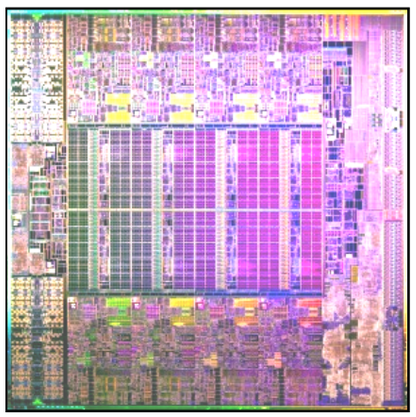
```
__kernel void partialReduce(   const int entries,
                      ?? const double *u,
                      ?? const double *newu,
                      ?? double *blocksum){

 __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.*

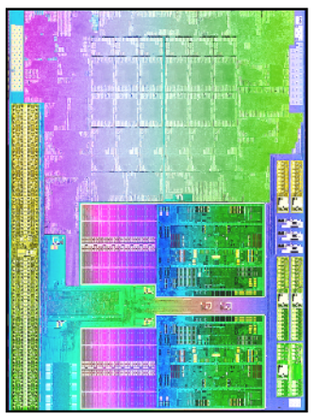
# Part 4b: Portability alternatives to OpenCL

# Many-core: updates

There are additional options

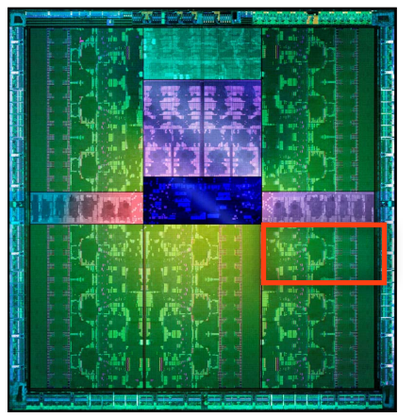


*Intel CPU*

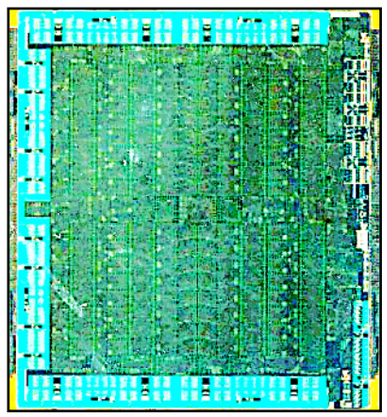


*AMD APU*

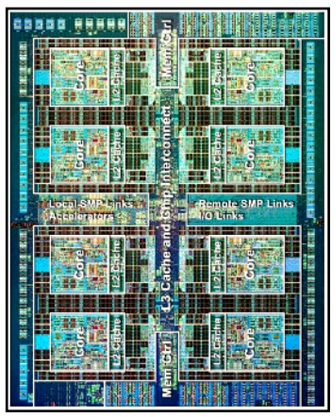
*Intel Xeon Phi*



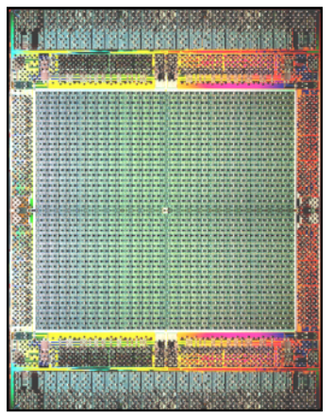
*NVIDIA GPU*



*AMD Tahiti GPU*




*IBM Power 7*

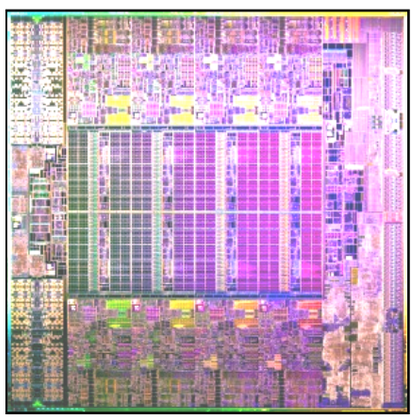
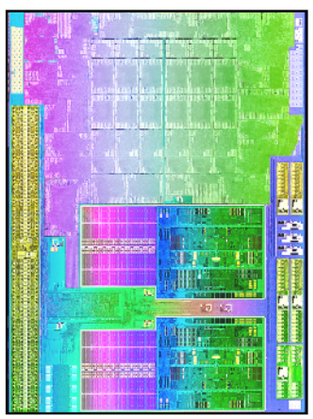
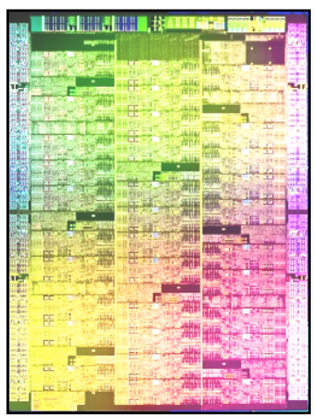
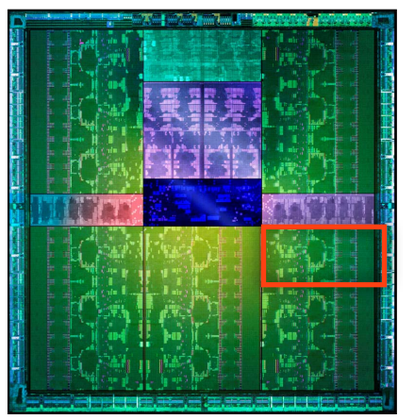
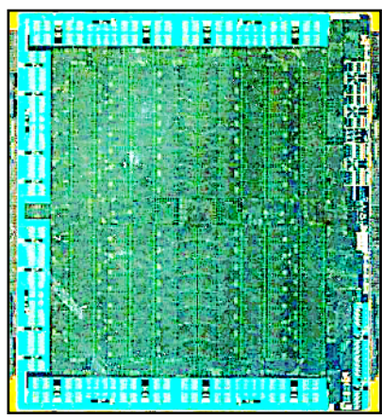
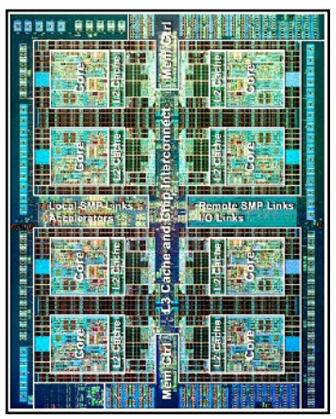


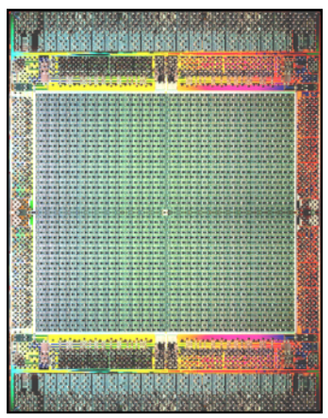
*Altera FPGA*

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

# Many-core: updates

There are additional options



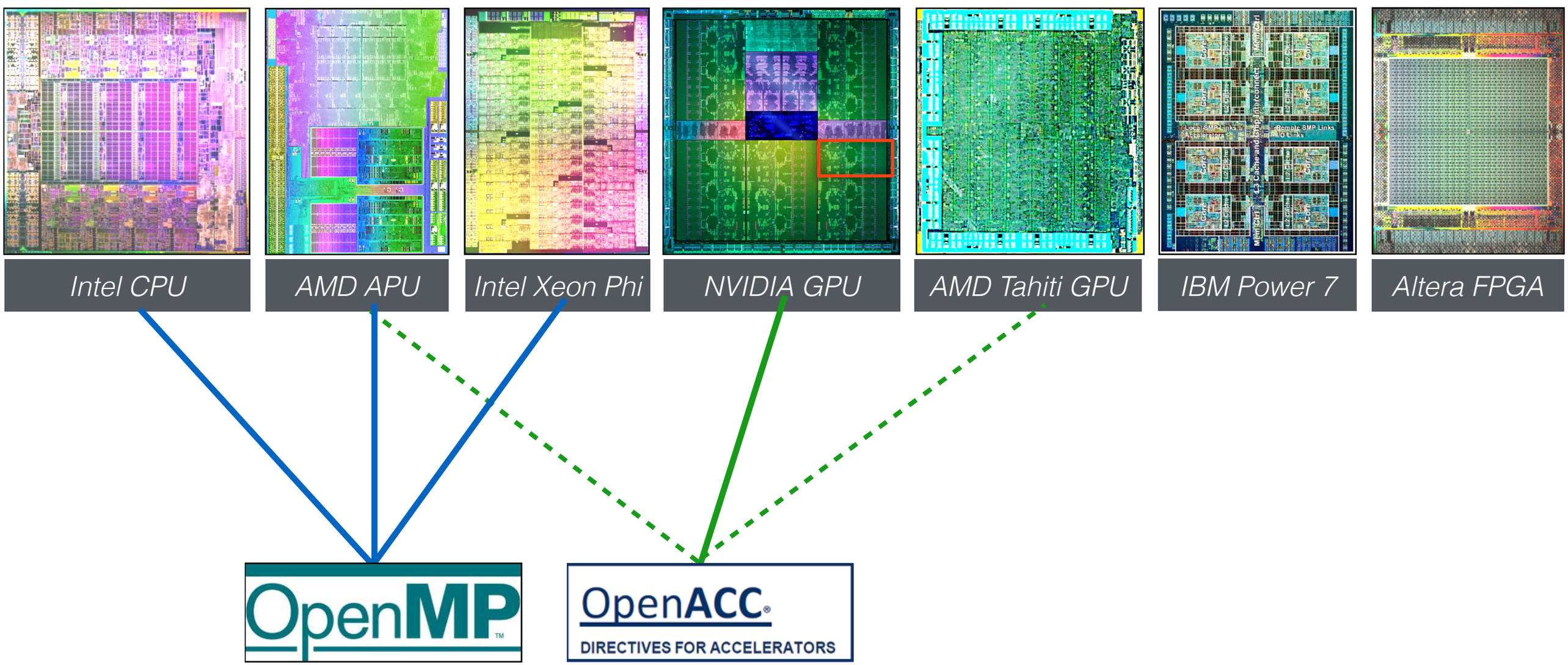
*Intel CPU* | *AMD APU* | *Intel Xeon Phi* | *NVIDIA GPU* | *AMD Tahiti GPU* | *IBM Power 7* | *Altera FPGA*

OpenMP™

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

# Many-core: updates

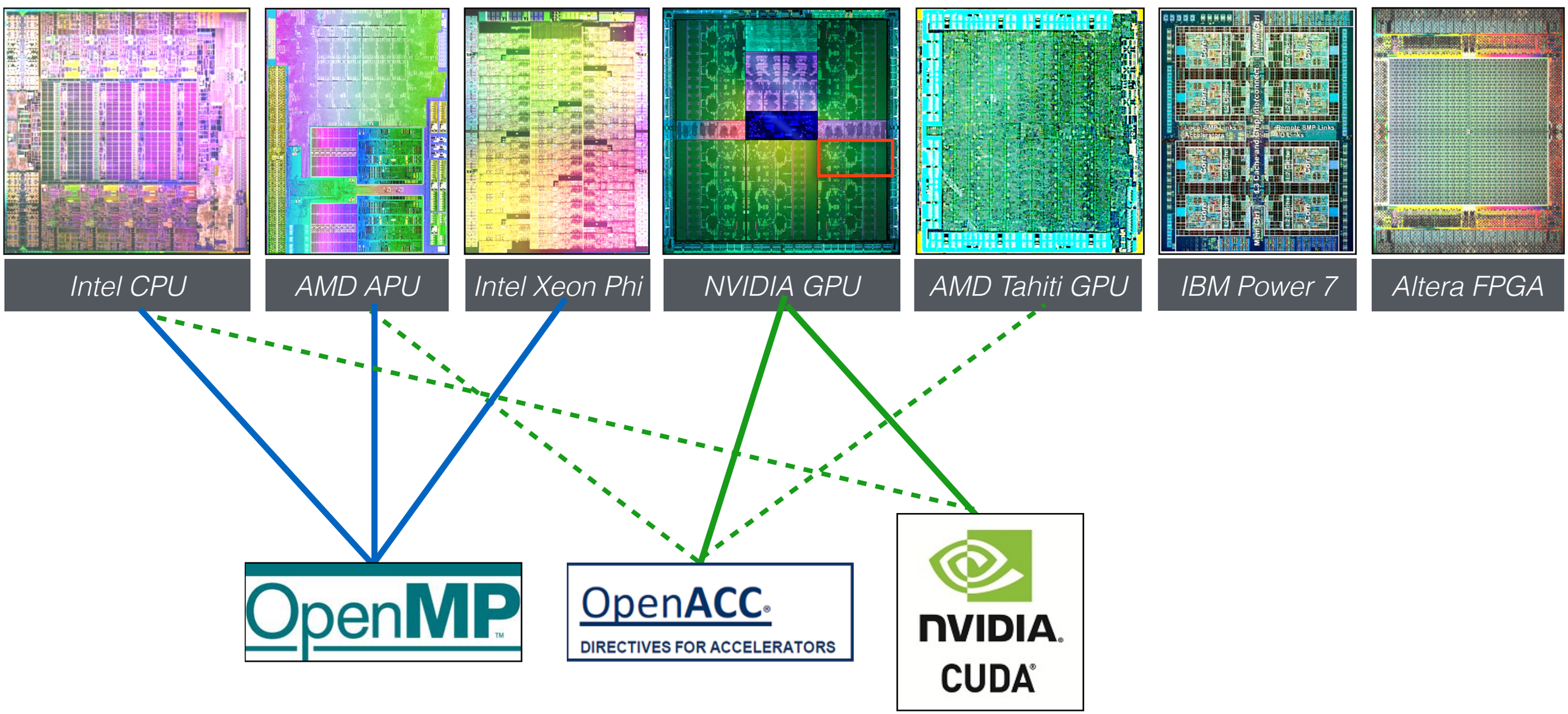
There are additional options




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

# Many-core: updates

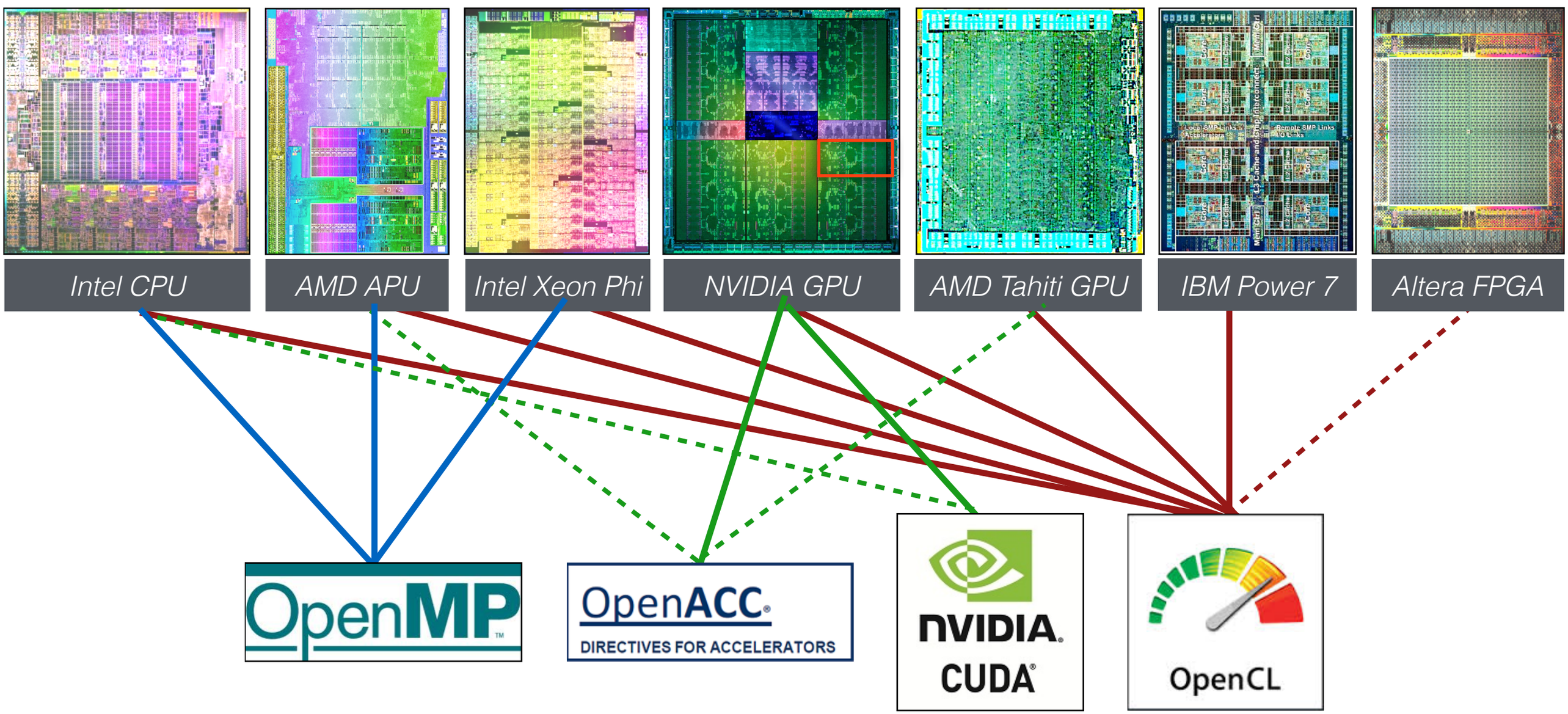
There are additional options




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

# Many-core: updates

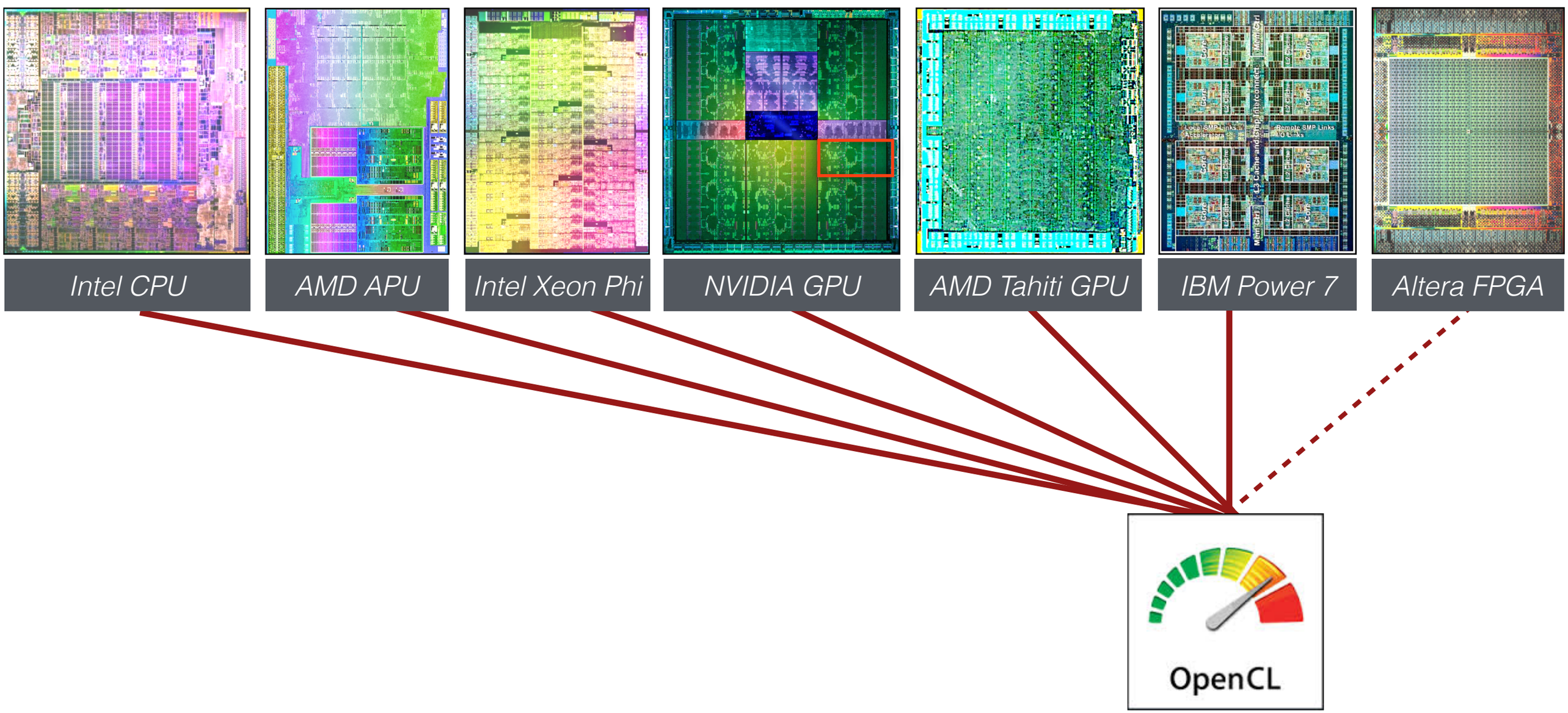
There are additional options




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

# Many-core: updates

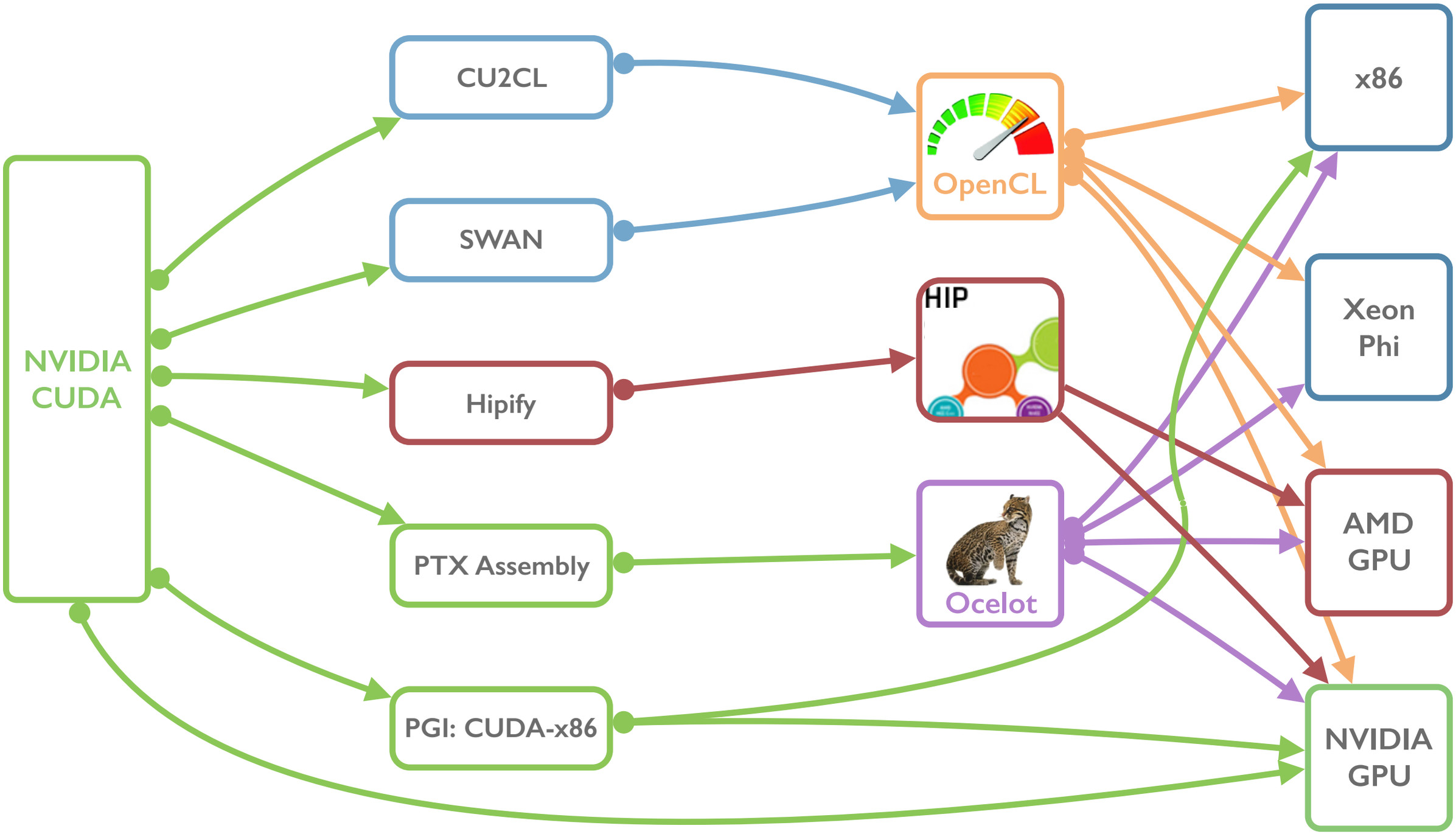
There are additional options




*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

OpenMP

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

Code taken from:
WHAT'S NEW IN OPENACC 2.0 AND OPENMP 4.0, GTC '14

# 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

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

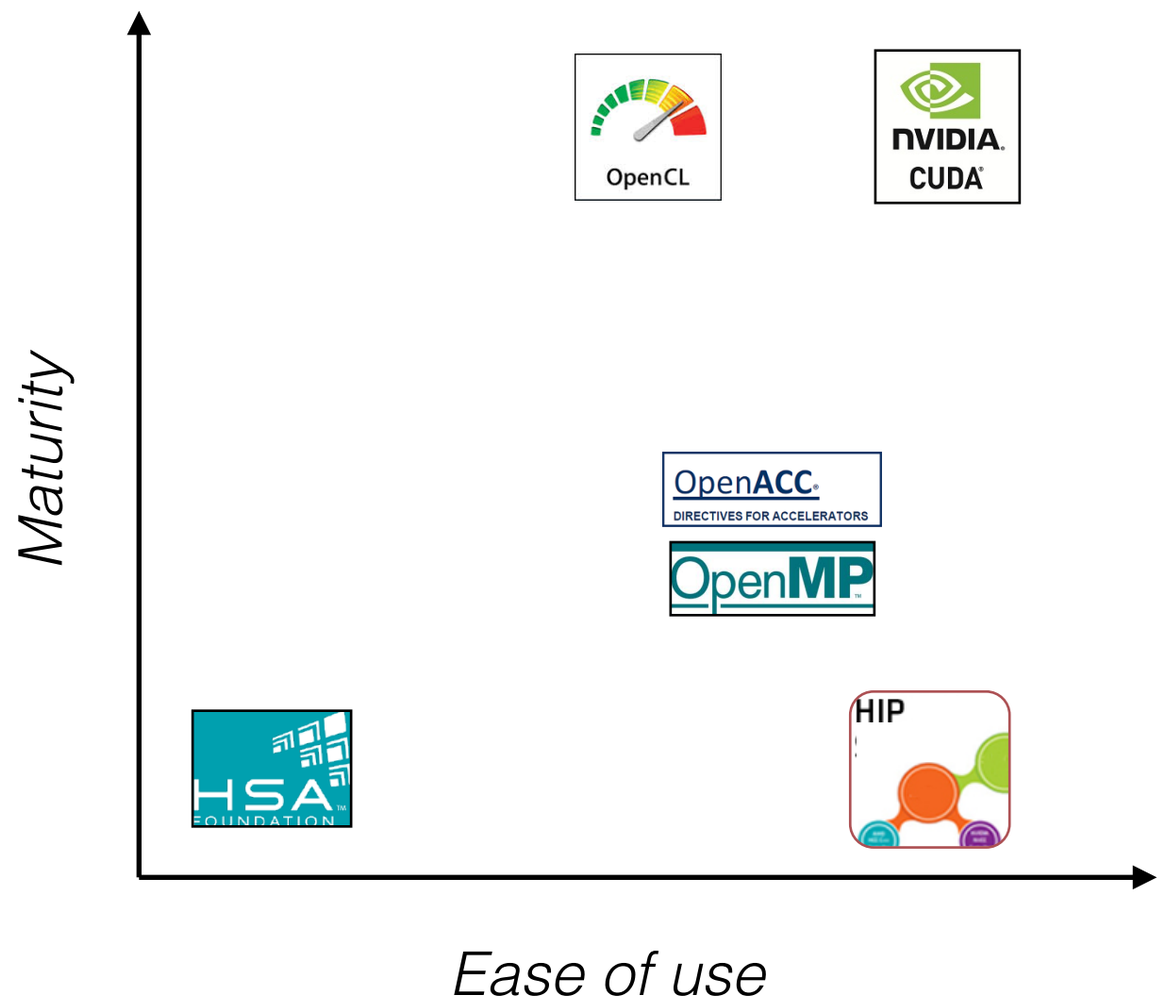
OpenACC

```
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;
  }
};
```

Code taken from:
WHAT'S NEW IN OPENACC 2.0 AND OPENMP 4.0, GTC '14

# 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 + ? |

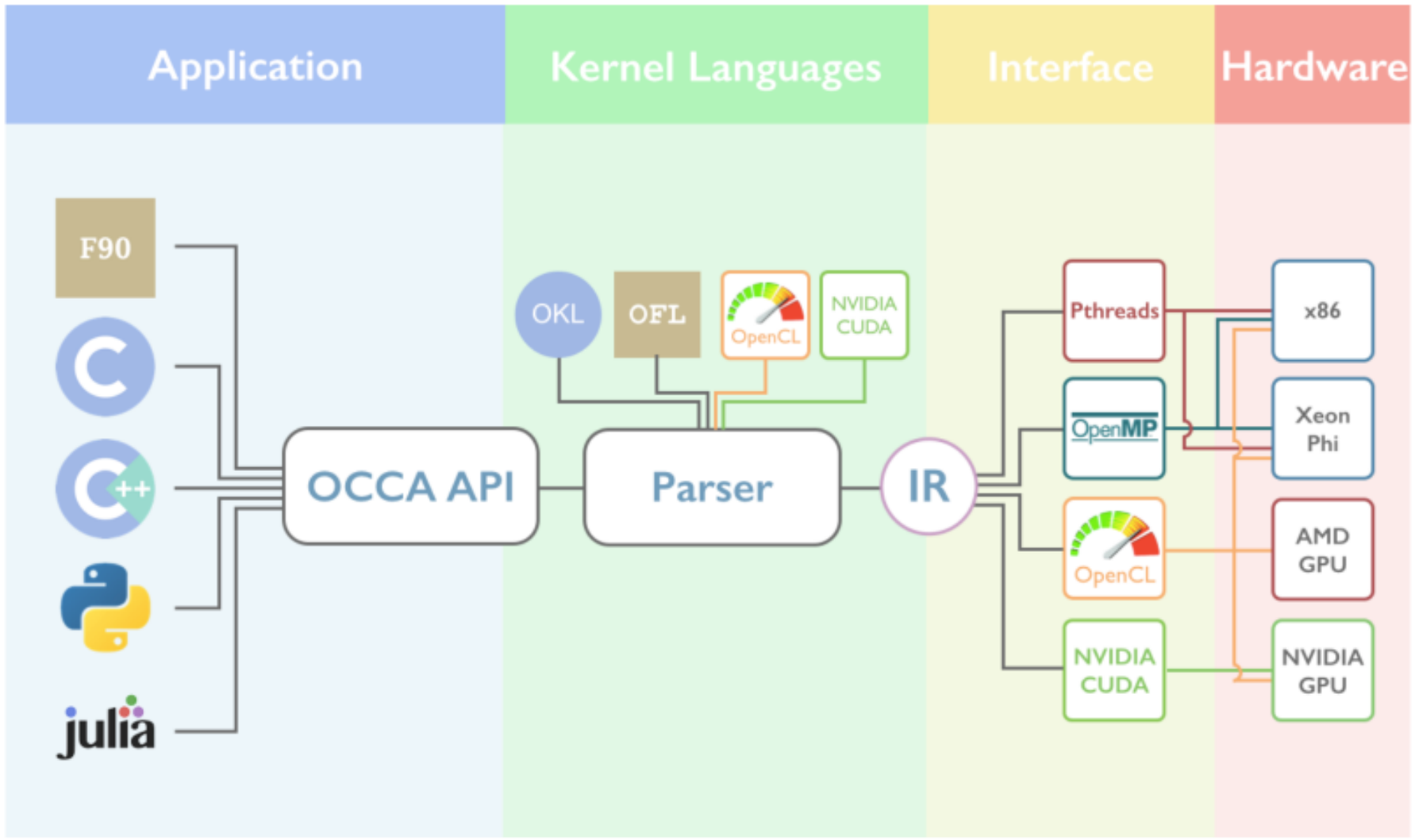
*Should it even matter what “X” is ?*

# Part 4c: OCCA
# Open Concurrent Compute Abstraction

# OCCA: easy portability

libocca.org

Downloads | Get Started | Tutorials | Documentation | About




OCCA (Open Concurrent Compute Abstraction) is an open-source (MIT-license) library used to program current multi-core/many-core architectures. Devices (such as CPUs, GPUs, Intel's Xeon Phi, FPGAs, etc) are abstracted using an offload model for application development and programming for the devices is done

github.com/libocca/occa
libocca.org

# What does OCCA provide ?

Open Concurrent Compute Architecture

API:

- Single, unified, library for programming heterogeneous devices.
- Flexibility at run-time to choose background thread programming model.
- Caching of compiled kernel binaries.

Portability:

- The OCCA-IR kernel language is portable across OpenMP, OpenCL, CUDA.
- The OCCA parser translates the OCCA Kernel Language (OKL) to the OCCA-IR.

Lightweight Framework:

- 50+30 KLOC (most auto-generated).
- Self-contained: dependencies are the backend compilers/drivers/runtimes.

Accessible:

- Exposes loop parallel paradigm … without an exotic programming model.
- C, C++, F90, Python … library front ends.

# 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.

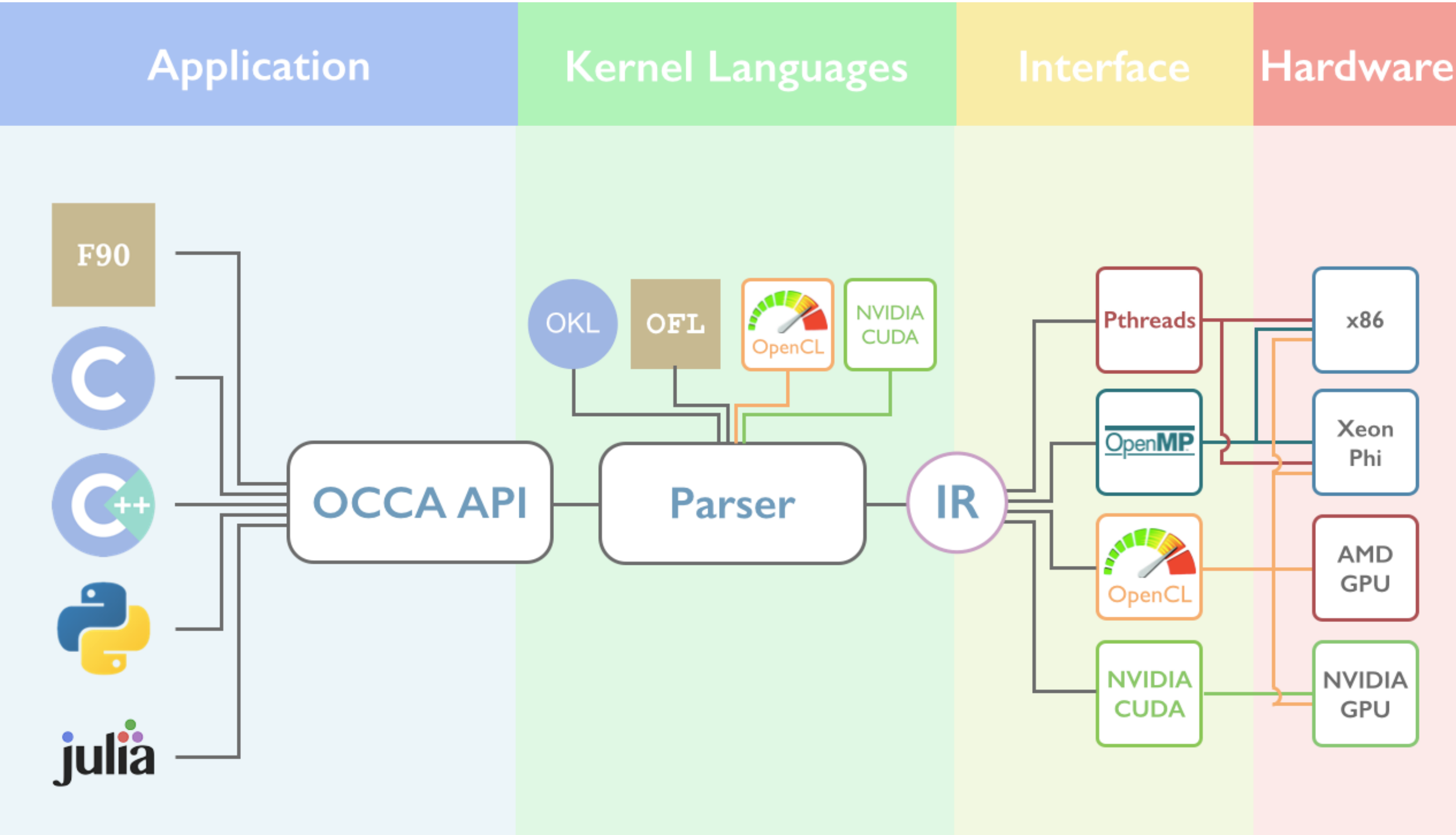
Auto-distribute:

- 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: easy portability




*github.com/libocca/occa*
*libocca.org*

# OCCA: give it a spin & live demo

Building the OCCA library:

```
git clone https://github.com/libocca/occa
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 |
| VexCL | Vector class | C++ | - | CUDA & OpenCL |
| RAJA | Library | C++ | C++ Lambdas | CUDA, OpenMP, OpenACC |
| OCCA | API, Source-to-source, Kernel Languages | C,C++,C#, F90, Python,MATLAB, Julia | OpenCL, CUDA,& custom unified kernel language | CUDA, OpenCL, pThreads,OpenMP, Intel COI |
| 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){    // Work-group implicit loops

        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

The concept of iterating over groups and items is simple

# 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){     // Work-group implicit loops

        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

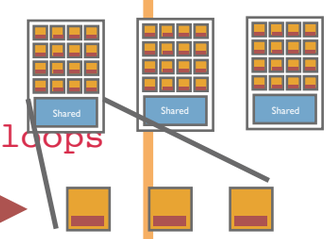
The concept of iterating over groups and items is simple

# 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){     // Work-group implicit loops

        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

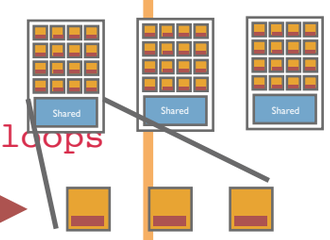
The concept of iterating over groups and items is simple

# 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){     // Work-group implicit loops

        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

NVIDIA CUDA

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

The concept of iterating over groups and items is simple

# 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(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

Data dependencies are found through a variable dependency graph

# 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){
    }
  }



  ...
}
```

OKL

Data dependencies are found through a variable dependency graph

# 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){
    }
  }



  ...
}
```

OKL

Data dependencies are found through a variable dependency graph

# 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){
    }
  }
}
```

OKL

Data dependencies are found through a variable dependency graph

# 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){
    }
  }
}
```

OKL

Data dependencies are found through a variable dependency graph

# 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(…){

  if(expr){
    for(outer){
      for(inner){
      }
    }
  else{
    for(outer){
      for(inner){
      }
    }
  }

  while(expr){
    for(outer){
      for(inner){
      }
    }
  }

}
```

OKL

Data dependencies are found through a variable dependency graph

# 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

Local barriers are auto-inserted

# 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)

```
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

# 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
```

```
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

# 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
```

```
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

# 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
   }
}
```

?

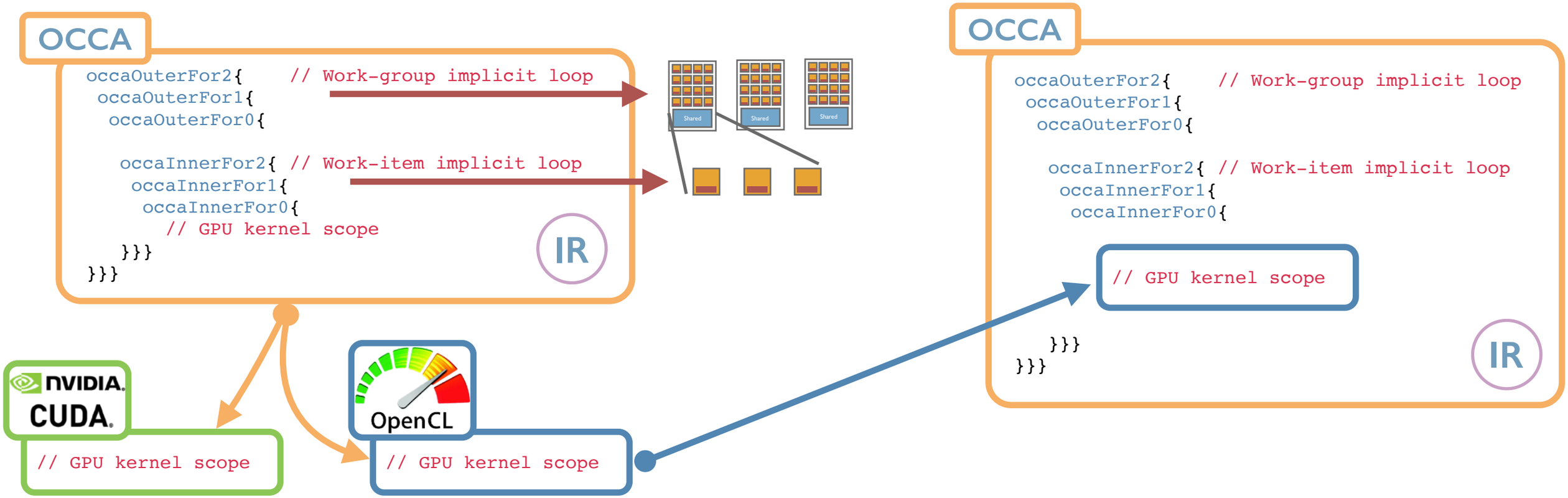
OKL

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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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 kernel language syntax & macros: http://www.github.com/tcew/OCCA2*

# 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];0

  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(“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';
```

*OCCA kernel language syntax & macros: http://www.github.com/tcew/OCCA2*
*Example HOST & DEVICE code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors*

# OCCA: example adding two vectors

```
#include <iostream>

#include "occa.hpp"

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

  for(int i = 0; i < N; ++
    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("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];
    }
  }
}
```

*OCCA kernel language syntax & macros: http://www.github.com/tcew/OCCA2*
*Example HOST & DEVICE code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors*

# Native Host Codes

```
#include <iostream>

#include "occa.hpp"

int main(int argc, char **argv){
  float *a  = new float[5];
  float *b  = new float[5];
  float *ab = new float[5];

  for(int i = 0; i < 5; ++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(“OpenCL”, 0, 0); // (Platform, Device) = (0, 0)

  o_a  = device.malloc(5*sizeof(float));
  o_b  = device.malloc(5*sizeof(float));
  o_ab = device.malloc(5*sizeof(float));

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

  addVectors = device.buildKernelFromSource(“addVectors.occa", "addVectors");

  int dims = 1;
  int itemsPerGroup(2);
  int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

  addVectors.setWorkingDims(dims, itemsPerGroup, groups);

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

*All the HOST codes use the same kernel.*
*Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors*

# Native Host Codes

```c
#include "stdlib.h"
#include "stdio.h"

#include "occa_c.h"

int main(int argc, char **argv){
  int i;

  float *a  = (float*) malloc(5*sizeof(float));
  float *b  = (float*) malloc(5*sizeof(float));
  float *ab = (float*) malloc(5*sizeof(float));

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

  occaDevice device;
  occaKernel addVectors;
  occaMemory o_a, o_b, o_ab;

  device = occaGetDevice("OpenCL", 0, 0);

  o_a  = occaDeviceMalloc(device, 5*sizeof(float), NULL);
  o_b  = occaDeviceMalloc(device, 5*sizeof(float), NULL);
  o_ab = occaDeviceMalloc(device, 5*sizeof(float), NULL);

  addVectors = occaBuildKernelFromSource(device,
                                         "addVectors.occa", "addVectors",
                                         occaNoKernelInfo);

  int dims = 1;
  occaDim itemsPerGroup, groups;

  itemsPerGroup.x = 2;
  groups.x        = (5 + itemsPerGroup.x - 1)/itemsPerGroup.x;

  occaKernelSetWorkingDims(addVectors,
                           dims, itemsPerGroup, groups);

  occaCopyPtrToMem(o_a, a, 5*sizeof(float), 0);
  occaCopyPtrToMem(o_b, b, occaAutoSize, occaNoOffset);

  occaKernelRun(addVectors,
                occaInt(5), o_a, o_b, o_ab);

  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

*All the HOST codes use the same kernel.*
*Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors*

# Native Host Codes

```julia
require( bytestring(ENV["OCCA_DIR"], "/lib/occa.jl") )

entries = 5

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

# Dynamic range?
a  = Float32[1 - i for i in 1:entries]
b  = Float32[i     for i in 1:entries]
ab = Float32[0     for i in 1:entries]

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

addVectors = occa.buildKernelFromSource(device,
                                        "addVectors.occa",
                                        "addVectors")

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

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

occa.runKernel(addVectors,
               (entries, Int32),
               o_a, o_b, o_ab)

occa.memcpy(ab, o_ab)

println(ab)
```

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

#include "

int main(
  int i;

  float *a
  float *b
  float *a

  for(i =
    a[i]
    b[i]
    ab[i]
  }

  occaDev
  occaKer
  occaMem

  device =

  o_a  = o
  o_b  = o
  o_ab = o

  addVect

  int dim
  occaDim

  itemsPe
  groups.

  occaKer

  occaCop
  occaCop

  occaKern

  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

*All the HOST codes use the same kernel.*
*Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors*

# Native Host Codes

```python
from ctypes import *
import occa

entries = 5

a  = [i     for i in xrange(entries)]
b  = [1 - i for i in xrange(entries)]
ab = [0     for i in xrange(entries)]

device = occa.device("OpenCL", 0, 0)

o_a  = device.malloc(a , c_float)
o_b  = device.malloc(b , c_float)
o_ab = device.malloc(ab, c_float)

addVectors = device.buildKernelFromSource("addVectors.occa",
                                          "addVectors")

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

addVectors.setWorkingDims(dims, itemsPerGroup, groups)

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

o_ab.copyTo(ab, c_float)

print ab
```

```julia
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
               (ent
               o_a, o_b, o_ab)

occa.memcpy(ab, o_ab)

println(ab)
```

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

#include "

int main(
  int i;

  float *a
  float *b
  float *a

  for(i =
    a[i]
    b[i]
    ab[i]
  }

  occaDev
  occaKer
  occaMem

  device =

  o_a  = o
  o_b  = o
  o_ab = o

  addVecto



  int dims
  occaDim

  itemsPe
  groups.

  occaKer


  occaCop
  occaCop

  occaKern


  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

*All the HOST codes use the same kernel.*
*Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors*

# Native

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

int main(
  int i;

  float *a
  float *b
  float *a

  for(i =
    a[i]
    b[i]
    ab[i]
  }

  occaDev
  occaKer
  occaMem

  device =

  o_a  = o
  o_b  = o
  o_ab = o

  addVect


  int dim
  occaDim

  itemsPe
  groups.

  occaKer


  occaCop
  occaCop

  occaKern

  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

```
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
               (ent
               o_a, o_b, o_ab)

occa.memcpy(ab, o_ab)

println(ab)
```

```
from ctypes
import occa

entries = 5

a  = [i
b  = [1 - i
ab = [0

device = occ

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*

# Native Host Codes

```python
from ctypes import *
import occa

entries = 5

a  = [i     for i in xrange(entries)]
b  = [1 - i for i in xrange(entries)]
ab = [0     for i in xrange(entries)]

device = occa.device("OpenCL", 0, 0)

o_a  = device.malloc(a , c_float)
o_b  = device.malloc(b , c_float)
o_ab = device.malloc(ab, c_float)

addVectors = device.buildKernelFromSource("addVectors.occa",
                                          "addVectors")

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

addVectors.setWorkingDims(dims, itemsPerGroup, groups)

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

o_ab.copyTo(ab, c_float)

print ab
```

```julia
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
               (ent
               o_a, o_b, o_ab)

occa.memcpy(ab, o_ab)

println(ab)
```

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

#include "

int main(
  int i;

  float *a
  float *b
  float *a

  for(i =
    a[i]
    b[i]
    ab[i]
  }

  occaDev
  occaKer
  occaMem

  device =

  o_a  = o
  o_b  = o
  o_ab = o

  addVecto



  int dims
  occaDim

  itemsPe
  groups.

  occaKer


  occaCop
  occaCop

  occaKern


  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

*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){

      // 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 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;

  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 datafloat
                     __global const datafloat
                     __global datafloat *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]);
  }
}
```

*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…)*

# 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){

      // G
      // i
      cons

      newu

    }
  }
}
```

### 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;

                                                  {

                                                  onto (N+2)x(N+2) grid
                                                  1)*(N + 2) + (i + 1);

                                                  s[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 datafloat
                     __global const datafloat
                     __global datafloat *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]);
  }
}
```

### 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]);
  }}}}}
}
```

*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…)*

# 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){

      // G
      // i
      cons

      newu

    }
  }
}
```

### 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;

{

onto
1)*(N

s[id
u[id
u[id
u[id
u[id
u[id
```

### 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);

x(N+2) grid
 + (i + 1);

2)]
2)]

;
```

### 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:

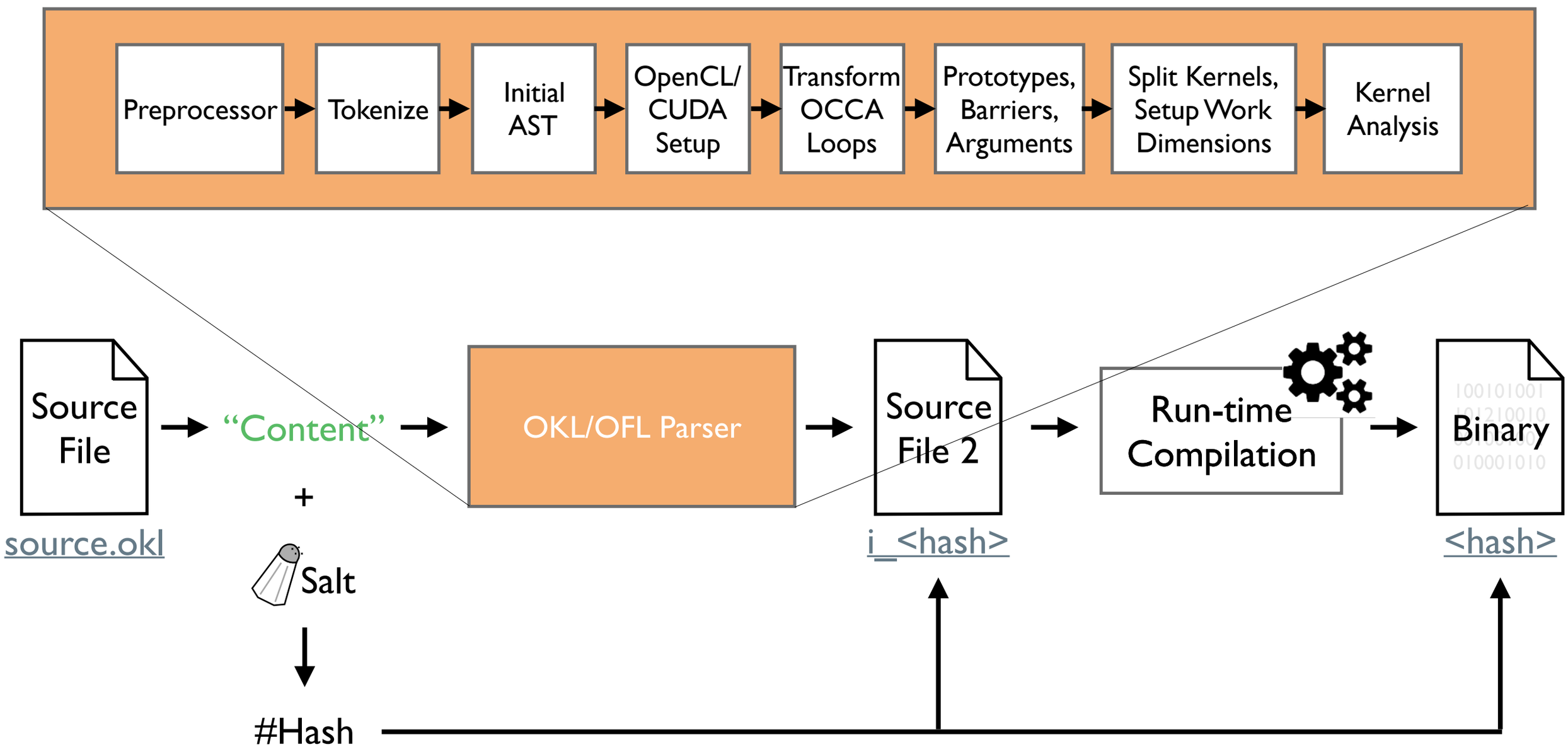
```
kernel void jacobi(const int N,
                   const datafloat *rhs,
                   const datafloat *u,
                   datafloat *newu){

  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

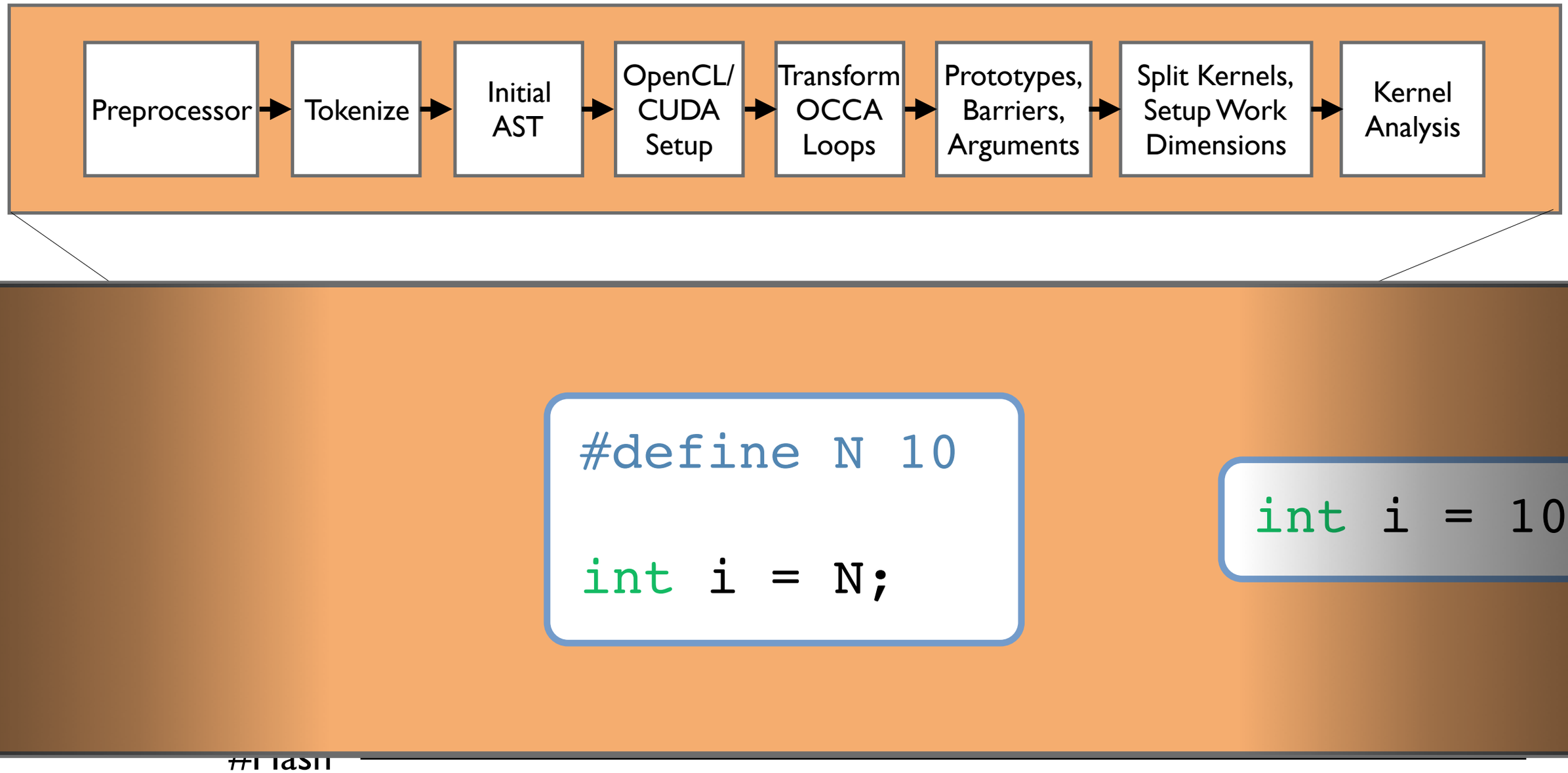



*Custom compilation tools tailored for code manipulation and analysis*

# Behind the Scenes: caching and hashing

## Source-to-Source Compilation

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

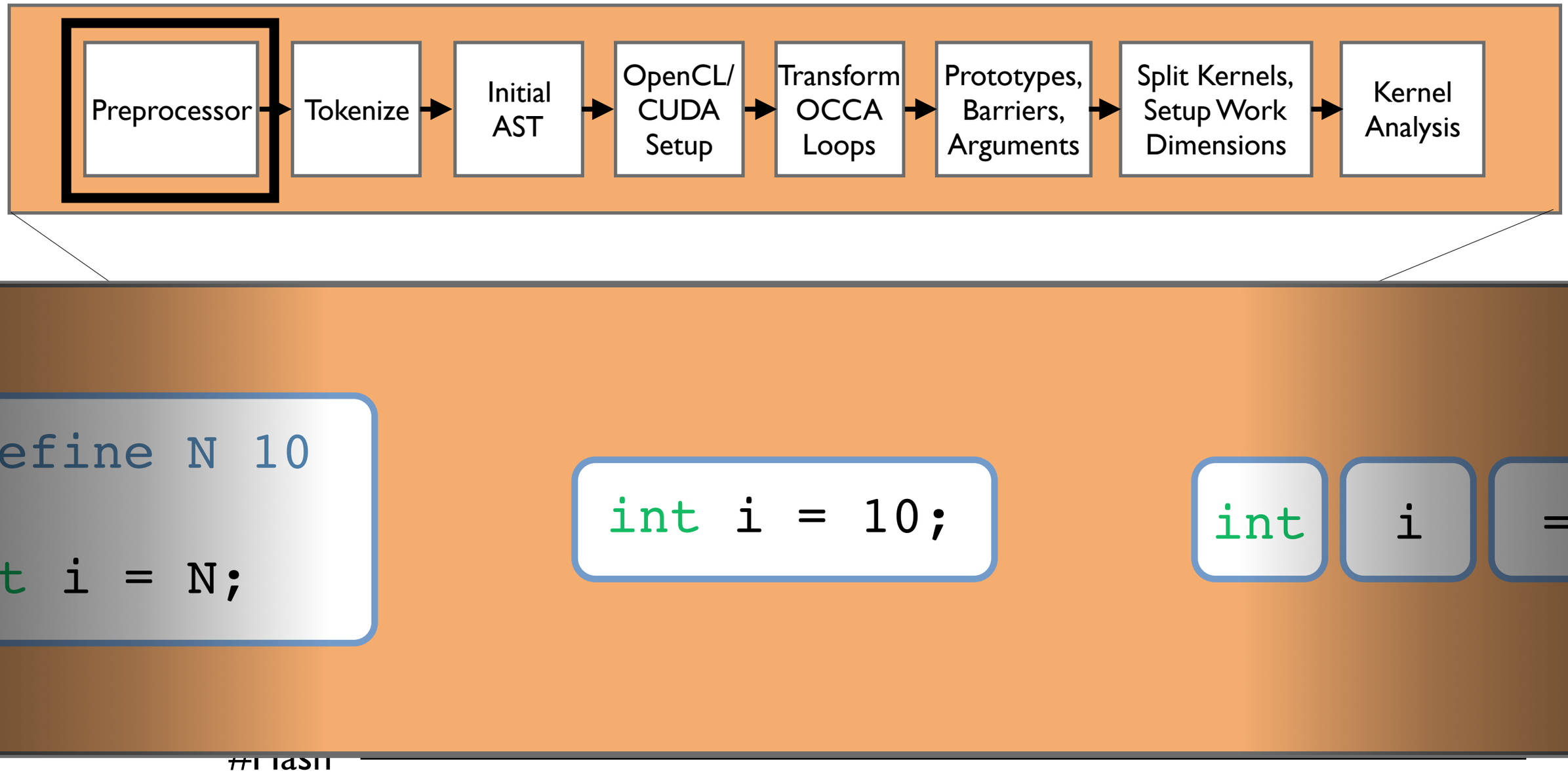



*Custom compilation tools tailored for code manipulation and analysis*

# OCCA Infrastructure

## Source-to-Source Compilation

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

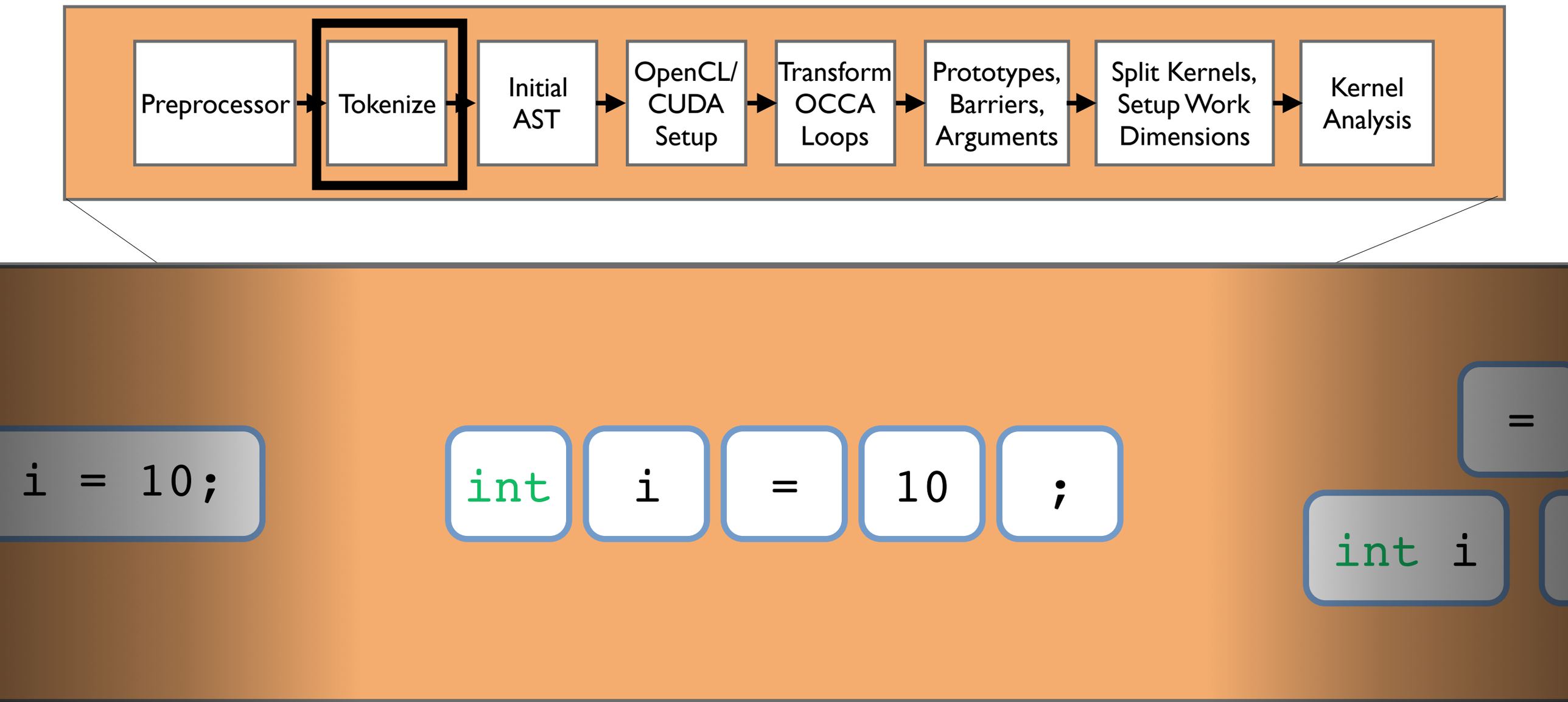



*Custom compilation tools tailored for code manipulation and analysis*

# OCCA Infrastructure

## Source-to-Source Compilation

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

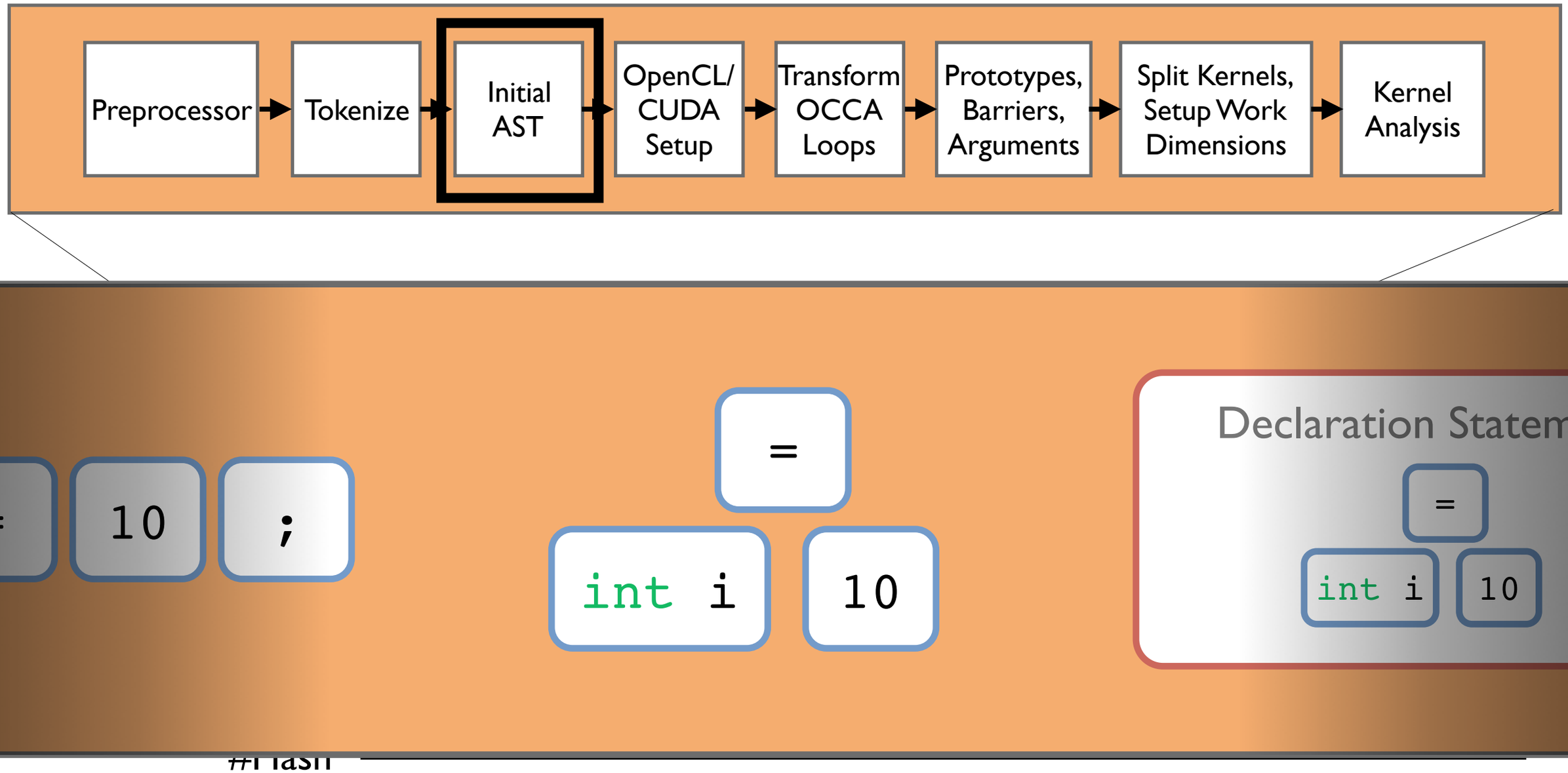



*Custom compilation tools tailored for code manipulation and analysis*

# OCCA Infrastructure

## Source-to-Source Compilation

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

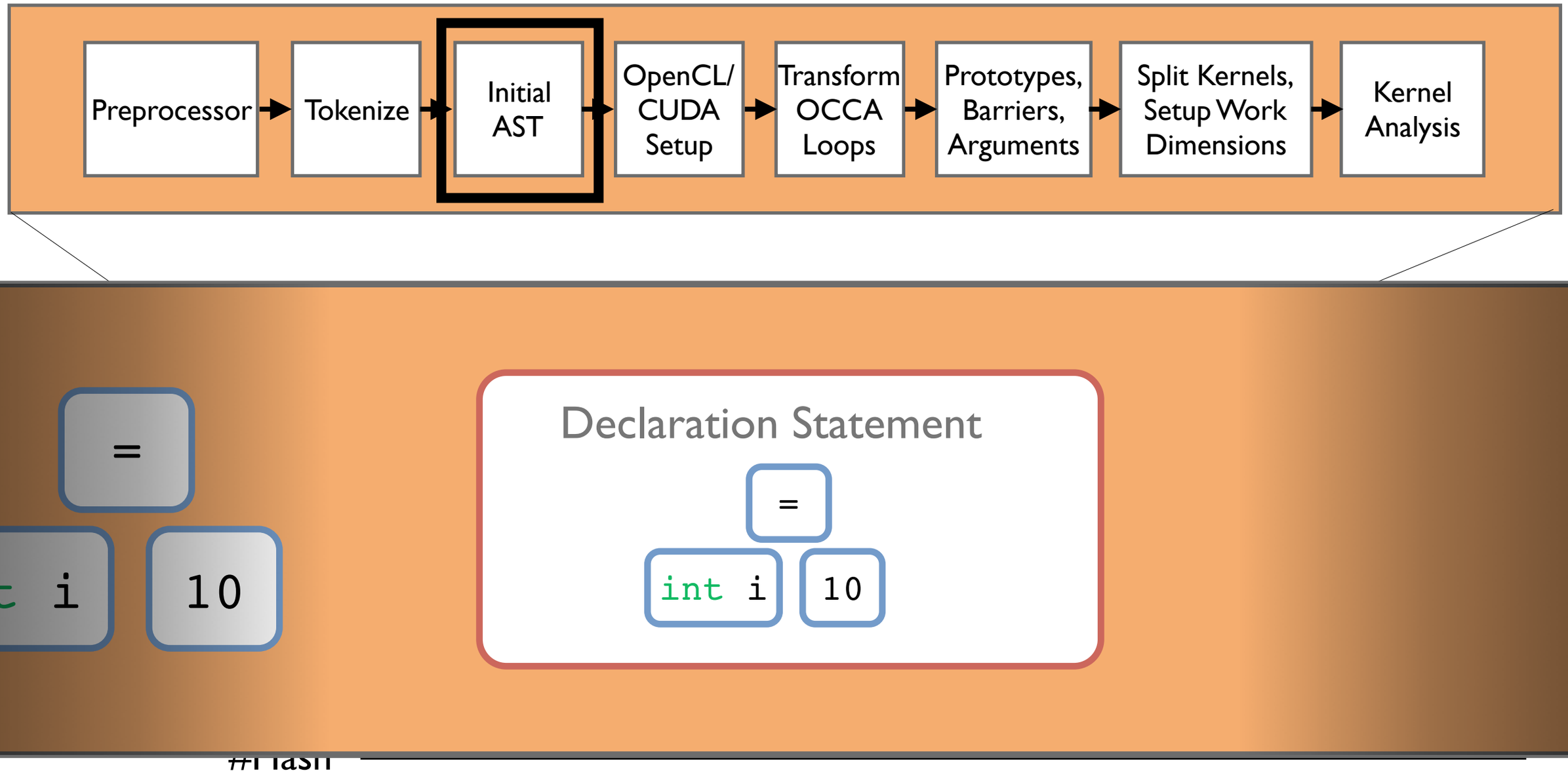



*Custom compilation tools tailored for code manipulation and analysis*

# OCCA Infrastructure

## Source-to-Source Compilation

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

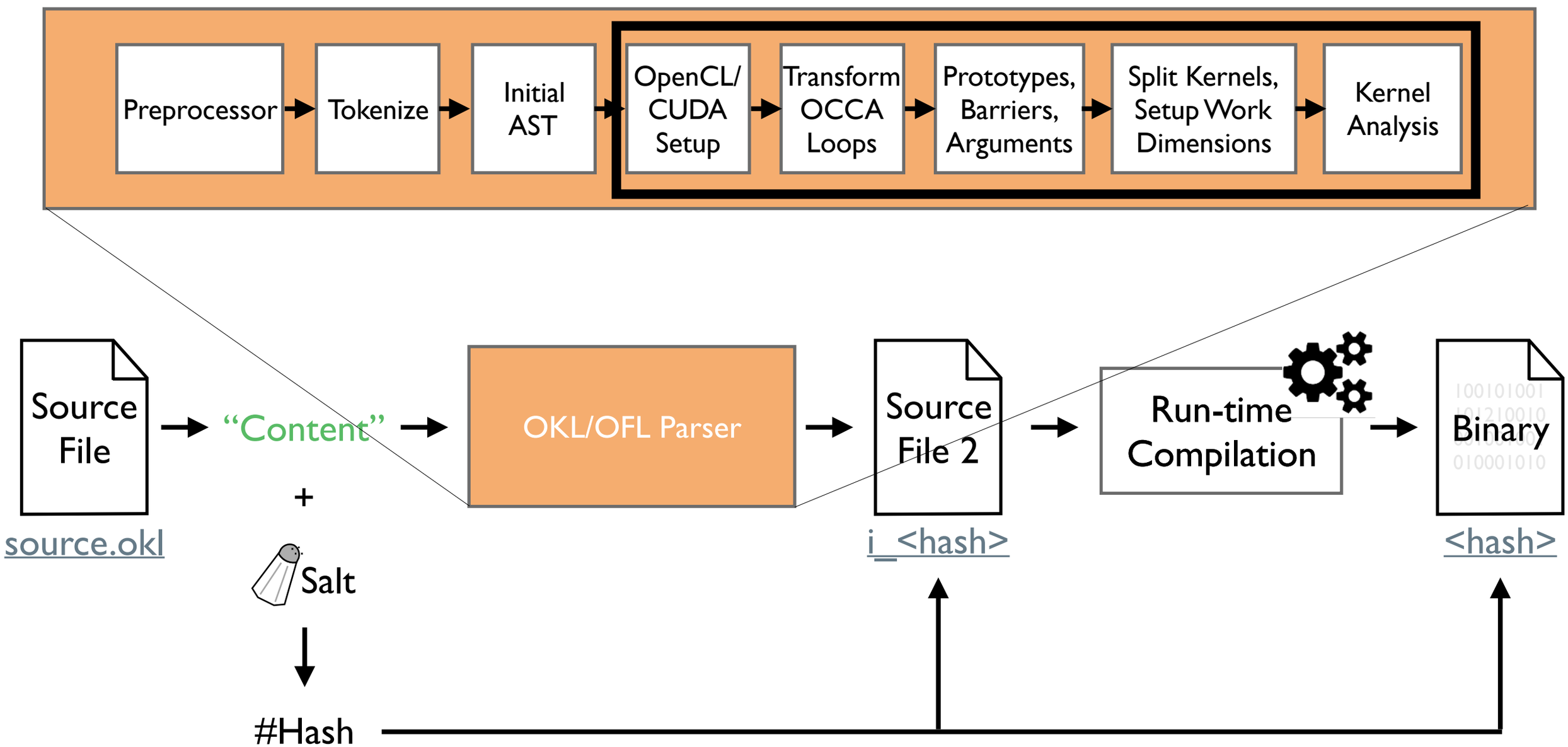



*Custom compilation tools tailored for code manipulation and analysis*

# 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.

# Hands On #3: *OCCA flow simulation*

In this exercise you will create a flow simulation

Work in teams of two.

# OCCA Flow Simulation: instructions

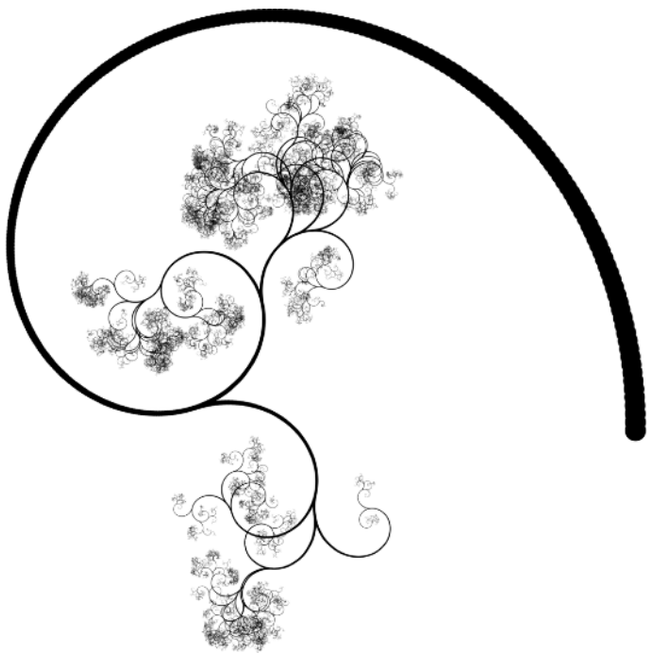
**#1. build the OCCA library**:

```
# clone the OCCA repo
git clone https://github.com/libocca/occa

# change into the OCCA directory
cd occa

# build OCCA
make -j

# add OCCA_DIR to env
export OCCA_DIR=`pwd`
```

**#2. build the OCCA LBM code**:

```
# clone the ATPESC17 repo
git clone https://github.com/tcew/ATPESC17

# change into the lbm directory
cd ATPESC17/lbm

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

**#3. save a png format image with white background to the lbm directory**:

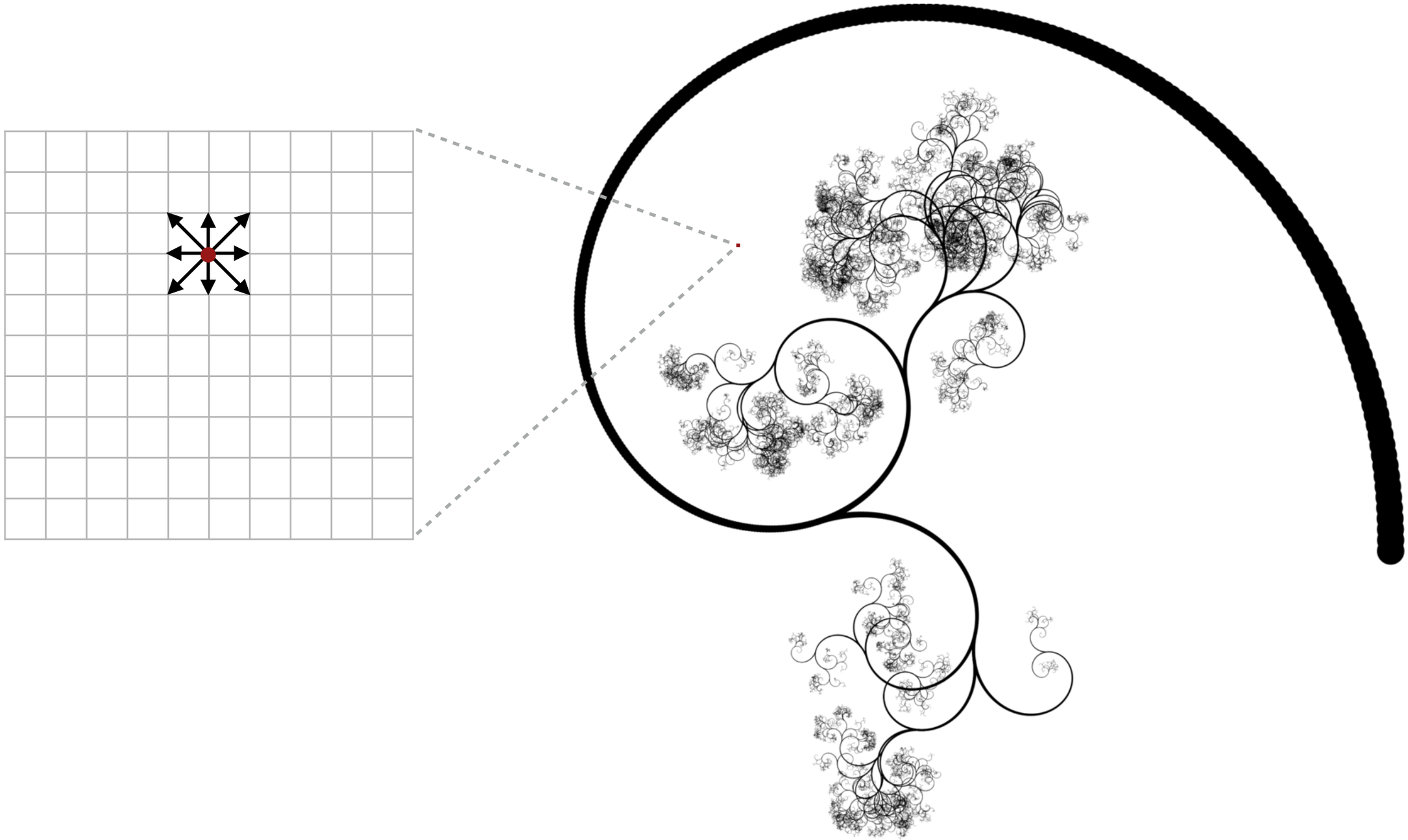


**#4. run the lbm code with your png image**

```
#  (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



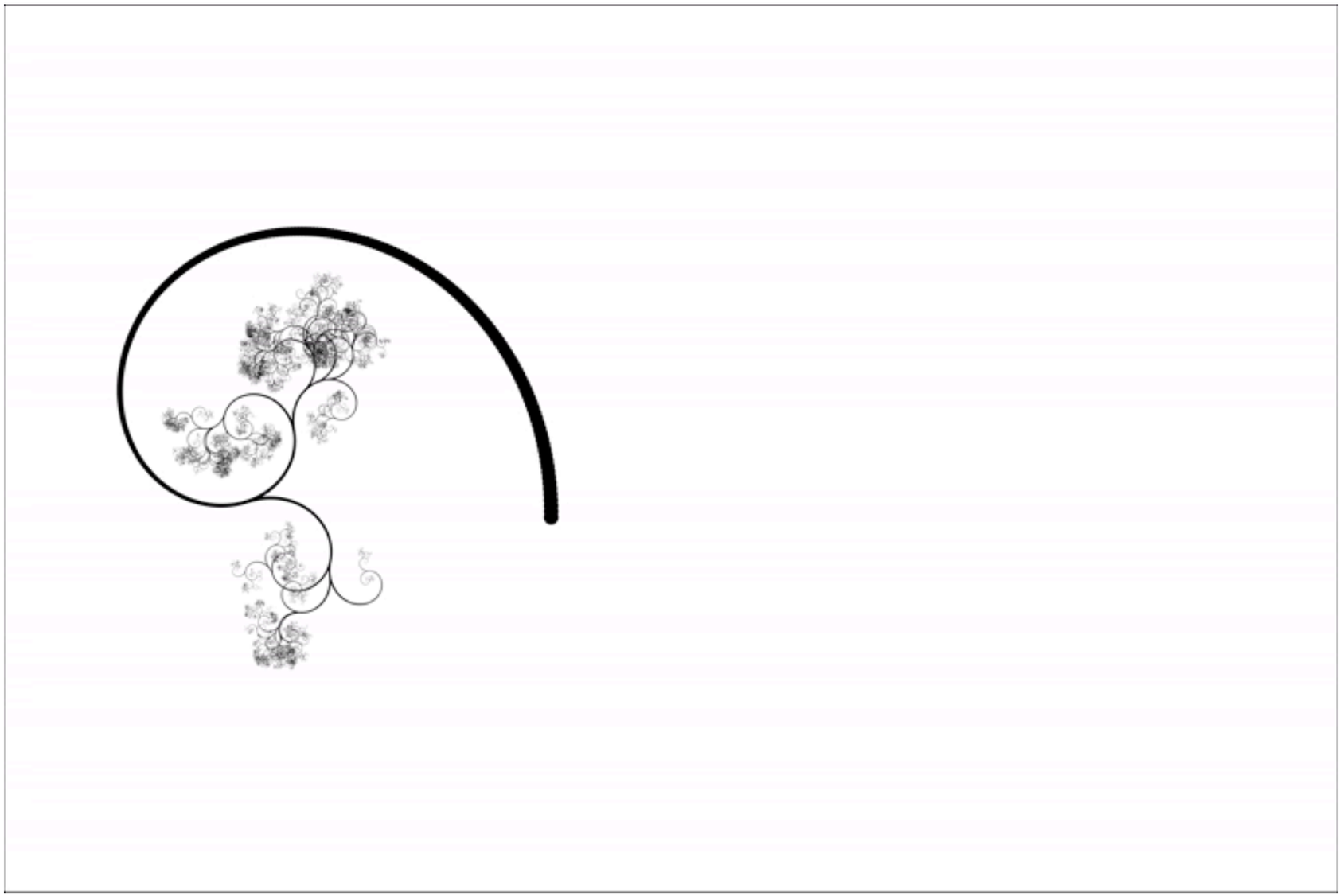
Details: a D2Q9 lattice Boltzmann method is implemented using an OCCA update kernel that uses a single thread to updates the 9 particle densities at each lattice node.

# 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**:



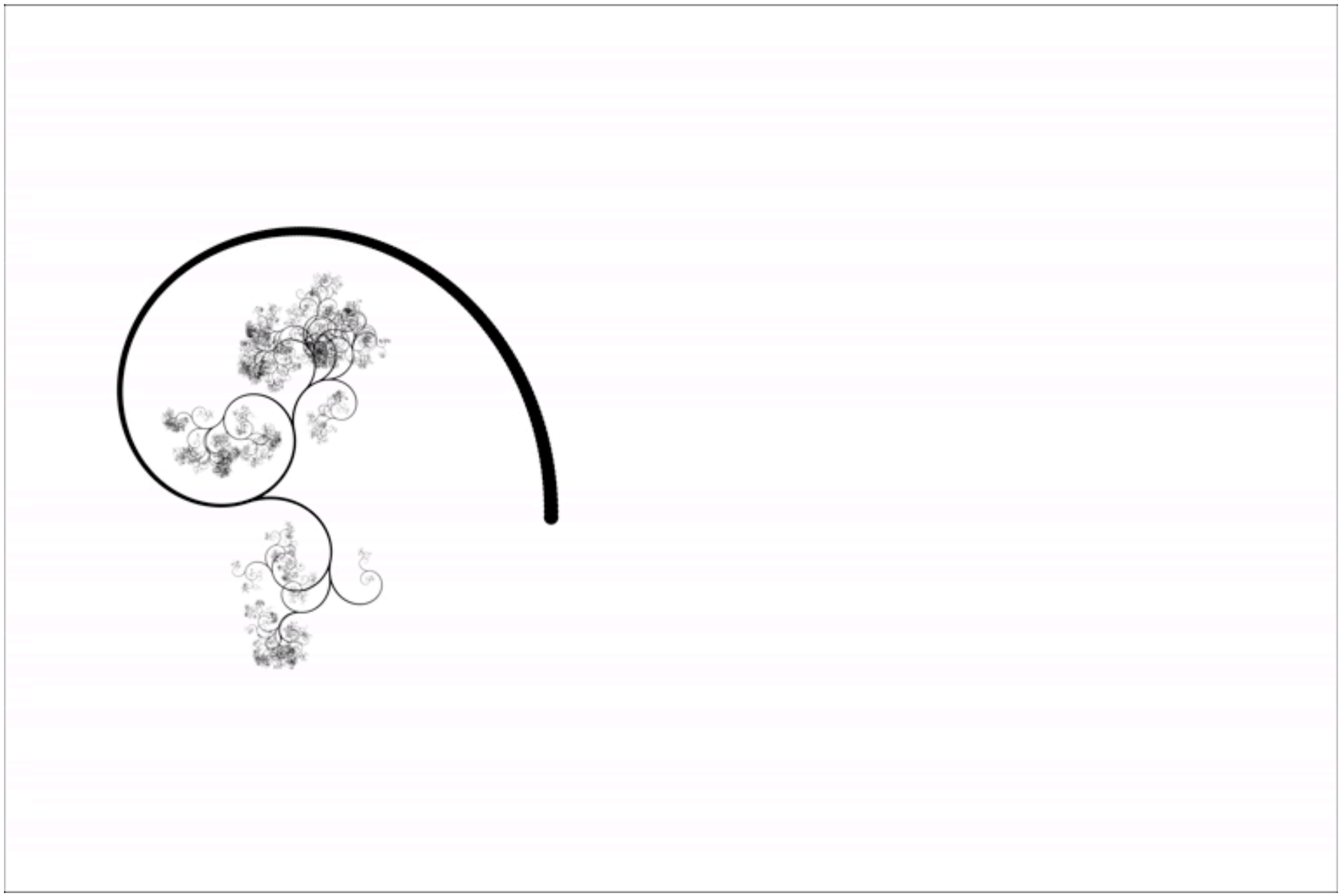
Raise your hand and demo your movie when done.

# OCCA Flow 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 ?

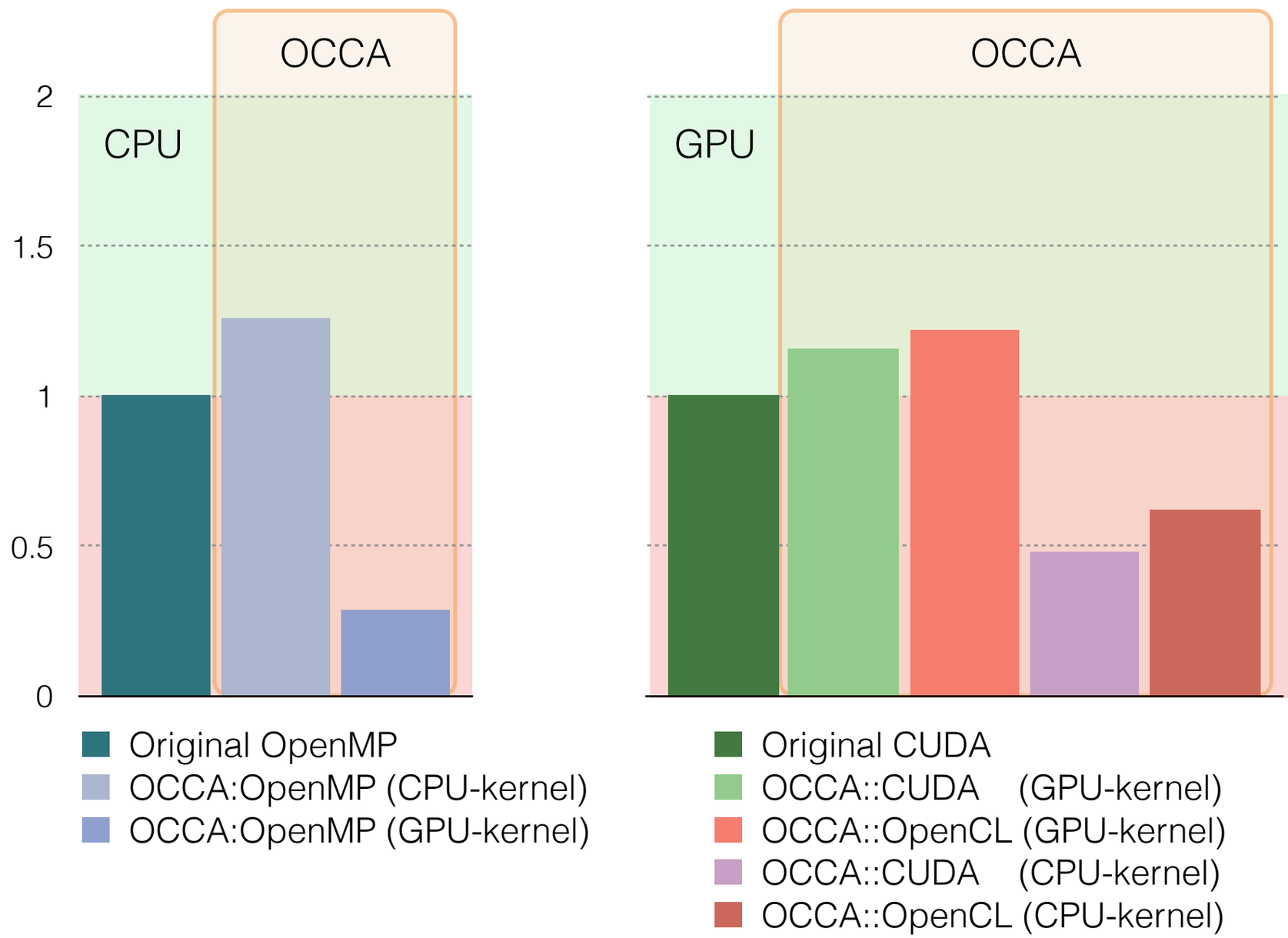
**#11. try installing and running on your laptop - this might be tricky.**

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

# Part 4d: OCCA Application Examples

# OCCA: apps & benchmarks

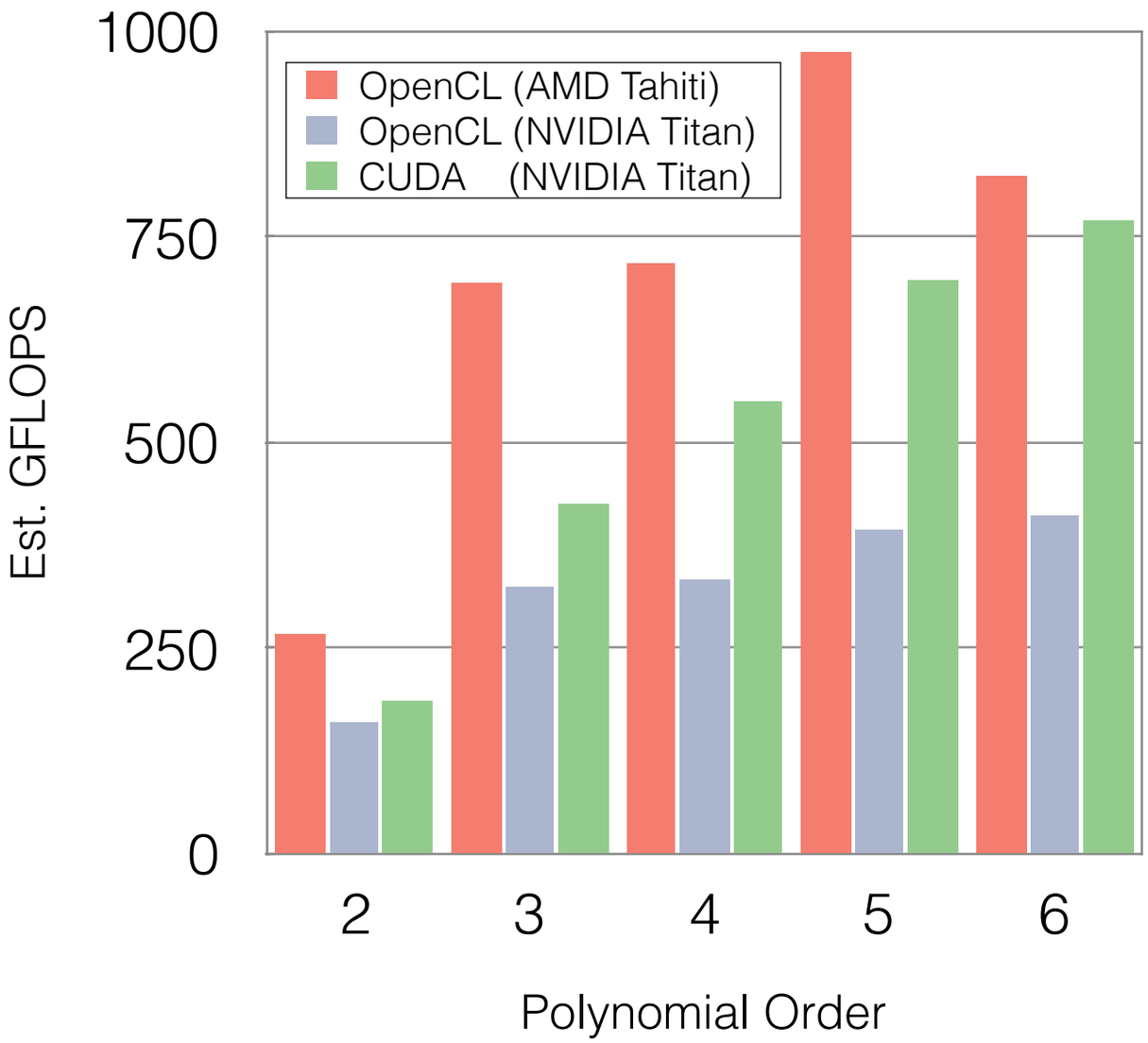
High-order finite difference for Reverse Time Migration (imaging algorithm)

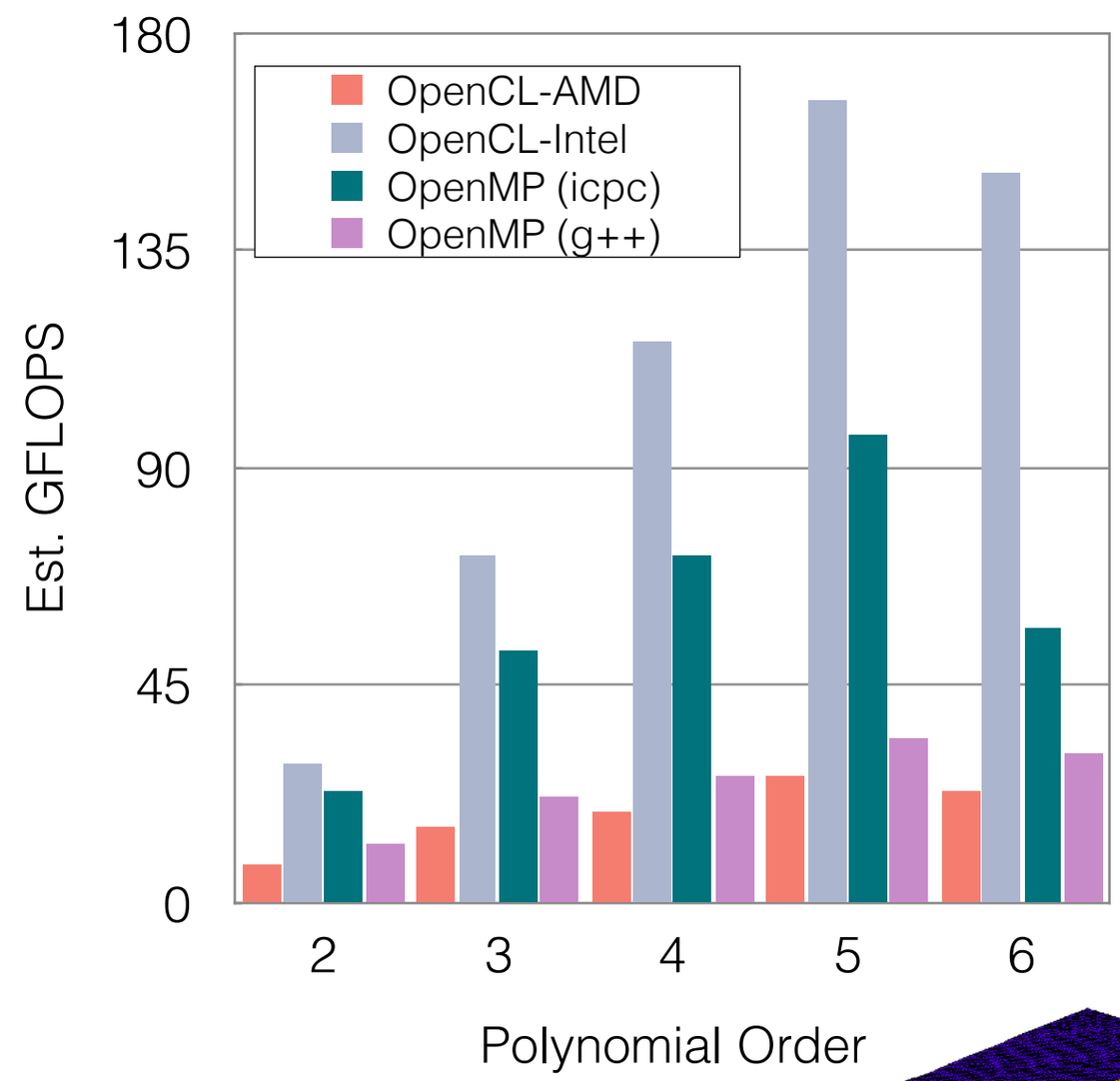


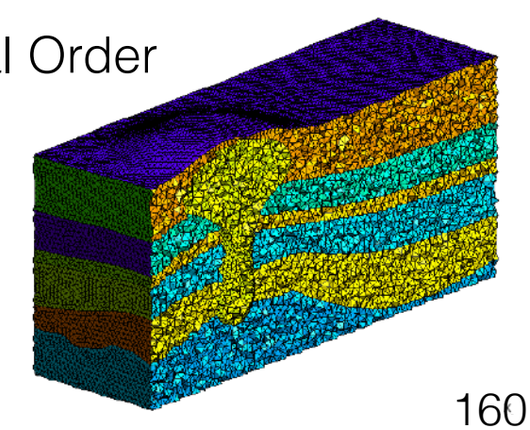

OpenMP : Intel Xeon CPU E5-2640
OpenCL/CUDA : NVIDIA Tesla K10

# OCCA: apps & benchmarks

Discontinuous Galerkin for RTM



# OCCA: apps & benchmarks

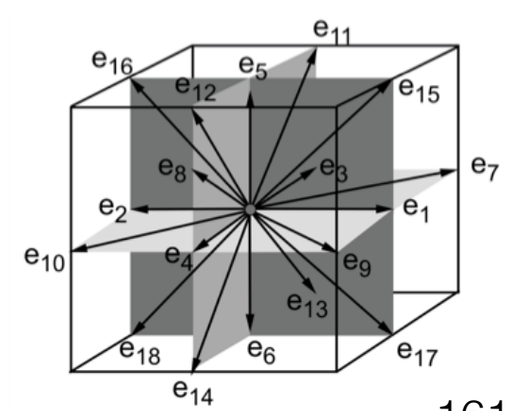
## Lattice Boltzmann Method in Core Sample Analysis

### 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 |
| OCCA | OpenCL: Intel | CPU | Intel i7-5960X | 11.18 | 22 | x 115 |
| OCCA | OpenCL: AMD | GPU | AMD 7990 | 1.39 | 176 | x 928 |
| OCCA | OpenCL: NVIDIA | GPU | GTX 980 | 1.25 | 196 | x 1032 |
| OCCA | CUDA: NVIDIA | GPU | GTX 980 | 1.20 | 205 | x 1075 |

### Comparison across platforms (Normalized with OCCA::OpenMP)

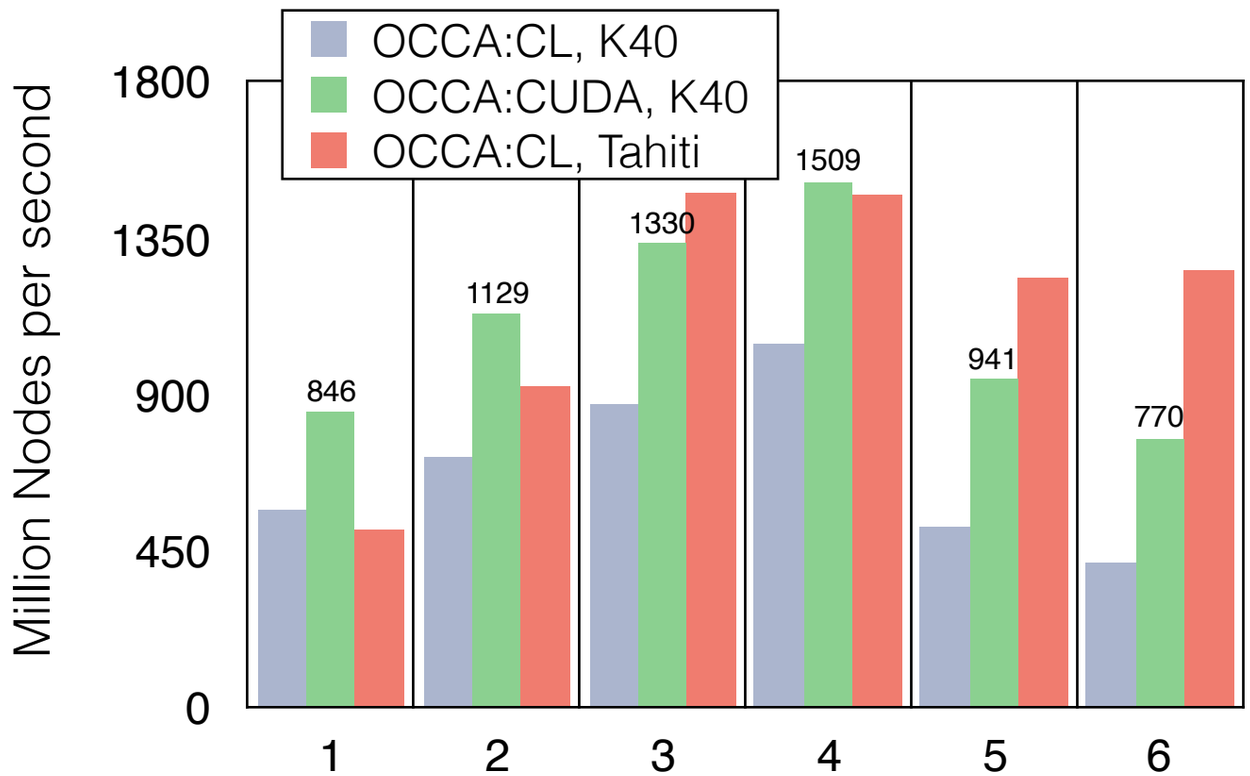
| | API Mode | Device | Model | Wall Clock | BW (GB/s) | Speedup |
|---|---|---|---|---|---|---|
| OCCA | OpenMP | CPU | Intel i7-5960X | 11.12 | 22 | x 1.0 |
| OCCA | OpenCL: Intel | CPU | Intel i7-5960X | 11.18 | 22 | x 1.0 |
| OCCA | OpenCL: AMD | GPU | AMD 7990 | 1.39 | 176 | x 8.0 |
| OCCA | OpenCL: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

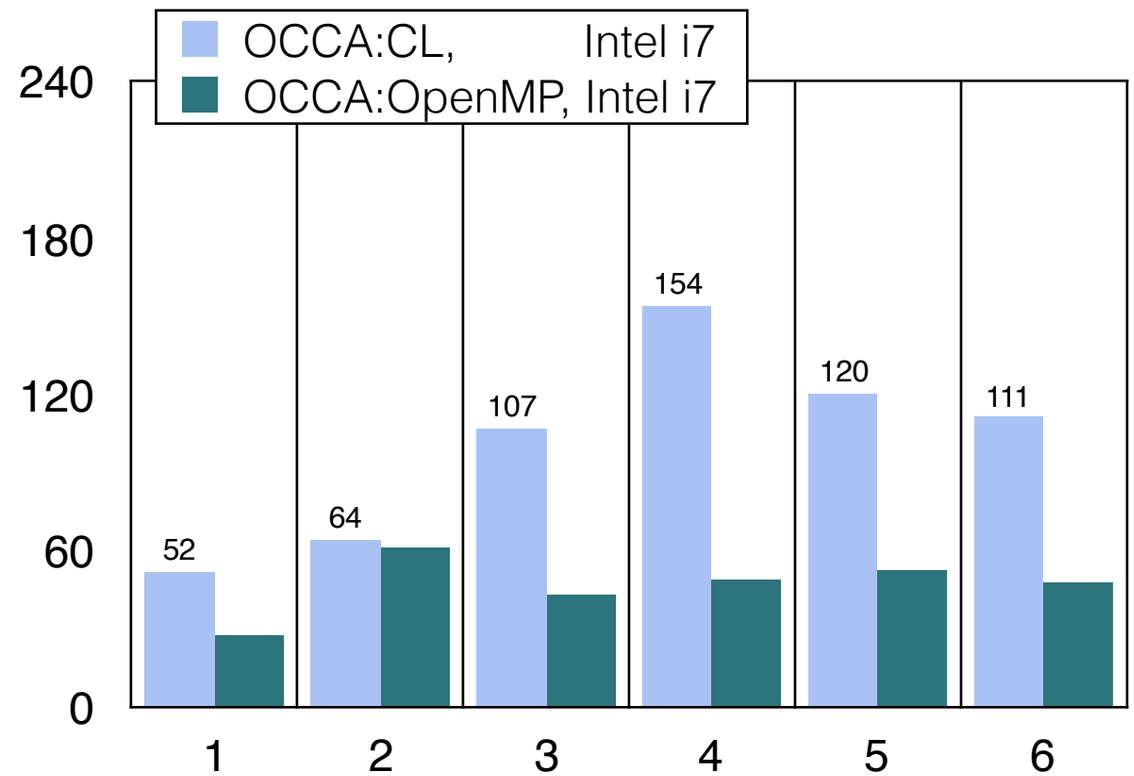


D3Q19

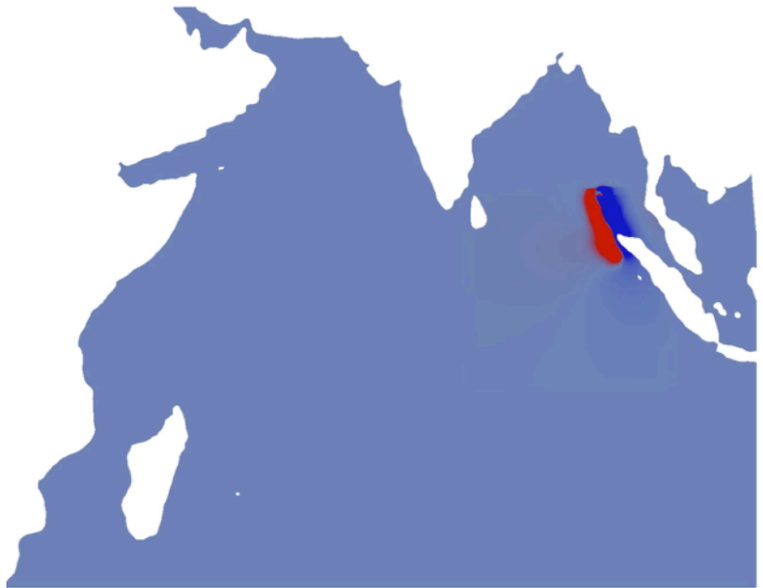
# Applications

## Discontinuous Galerkin for shallow water equations



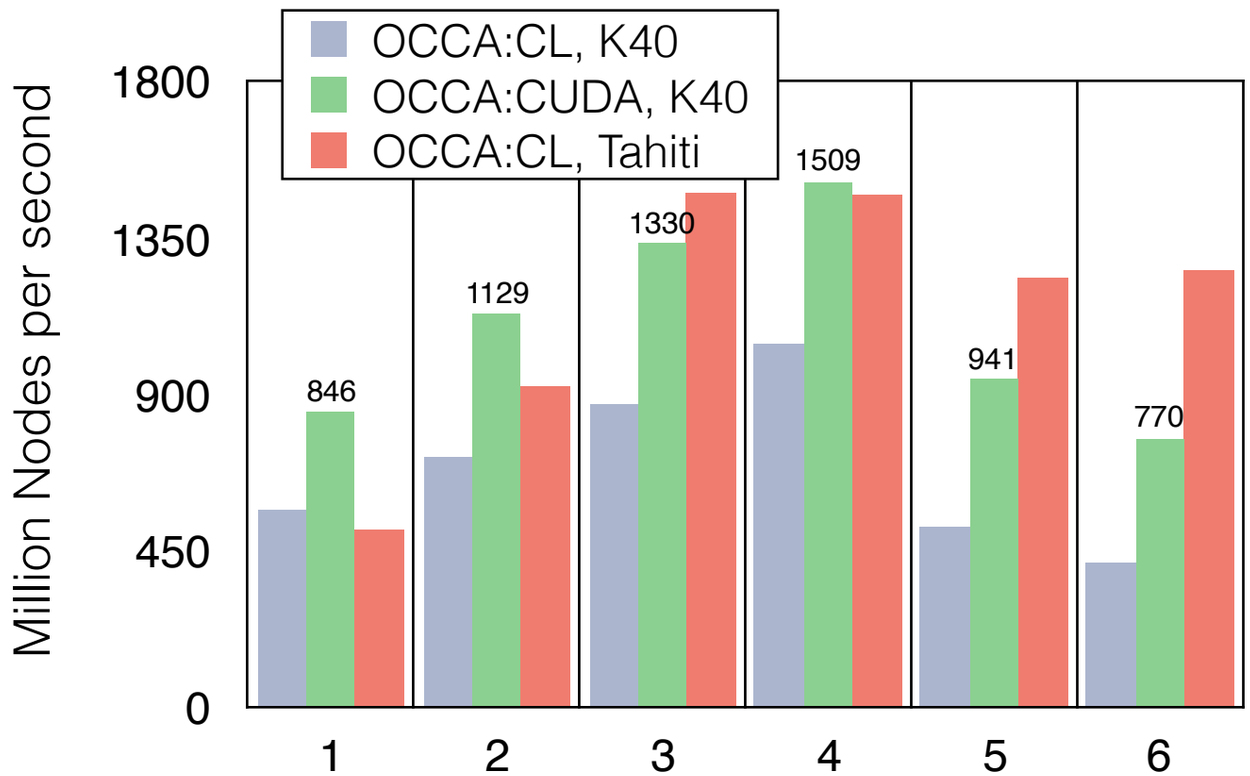



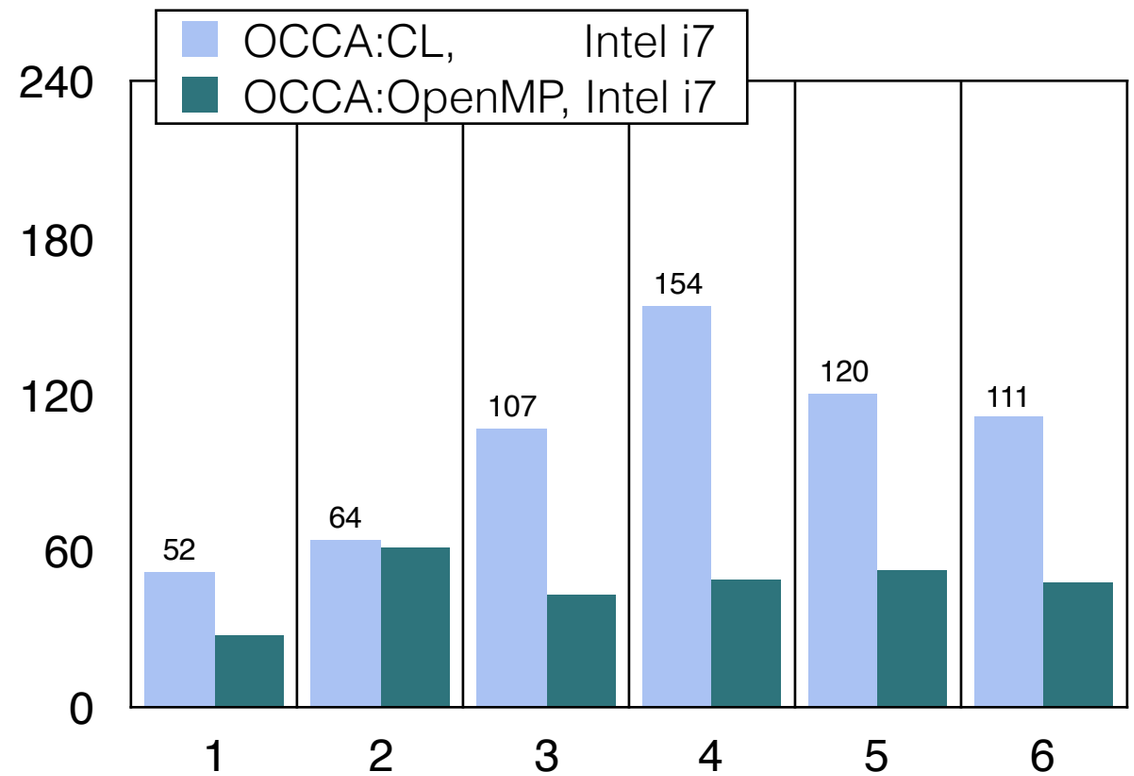

| Polynomial Order | Compute-Time vs Real-Time |
|---|---|
| 1 | x650 |
| 2 | x208 |
| 3 | x95 |
| 4 | x47 |



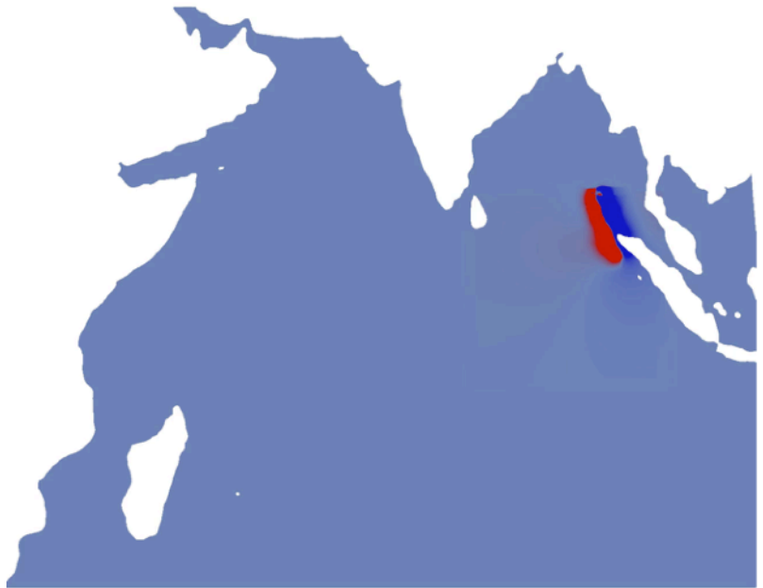
# Applications

## Discontinuous Galerkin for shallow water equations



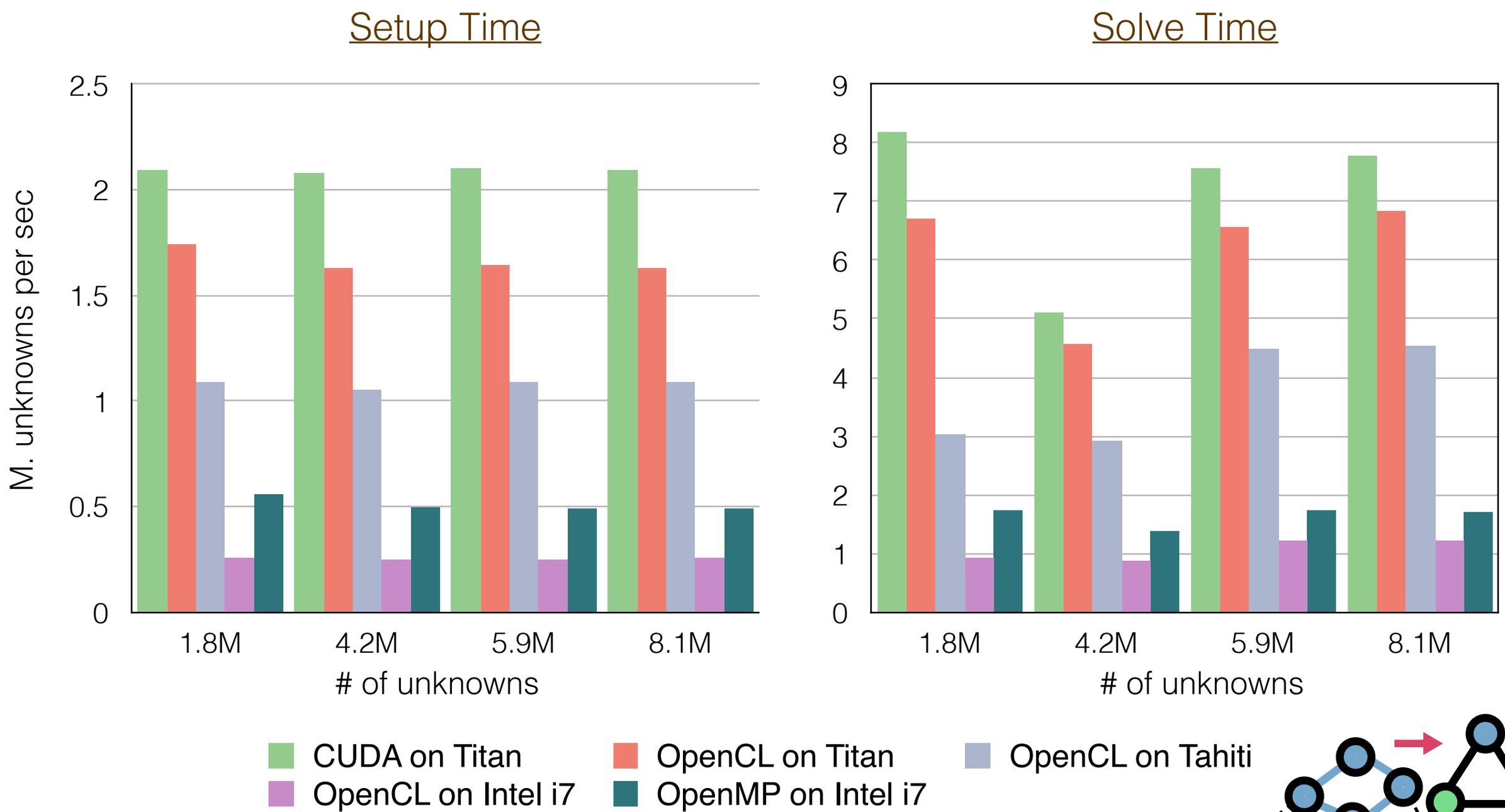




| Polynomial Order | Compute-Time vs Real-Time |
|---|---|
| 1 | x650 |
| 2 | x208 |
| 3 | x95 |
| 4 | x47 |



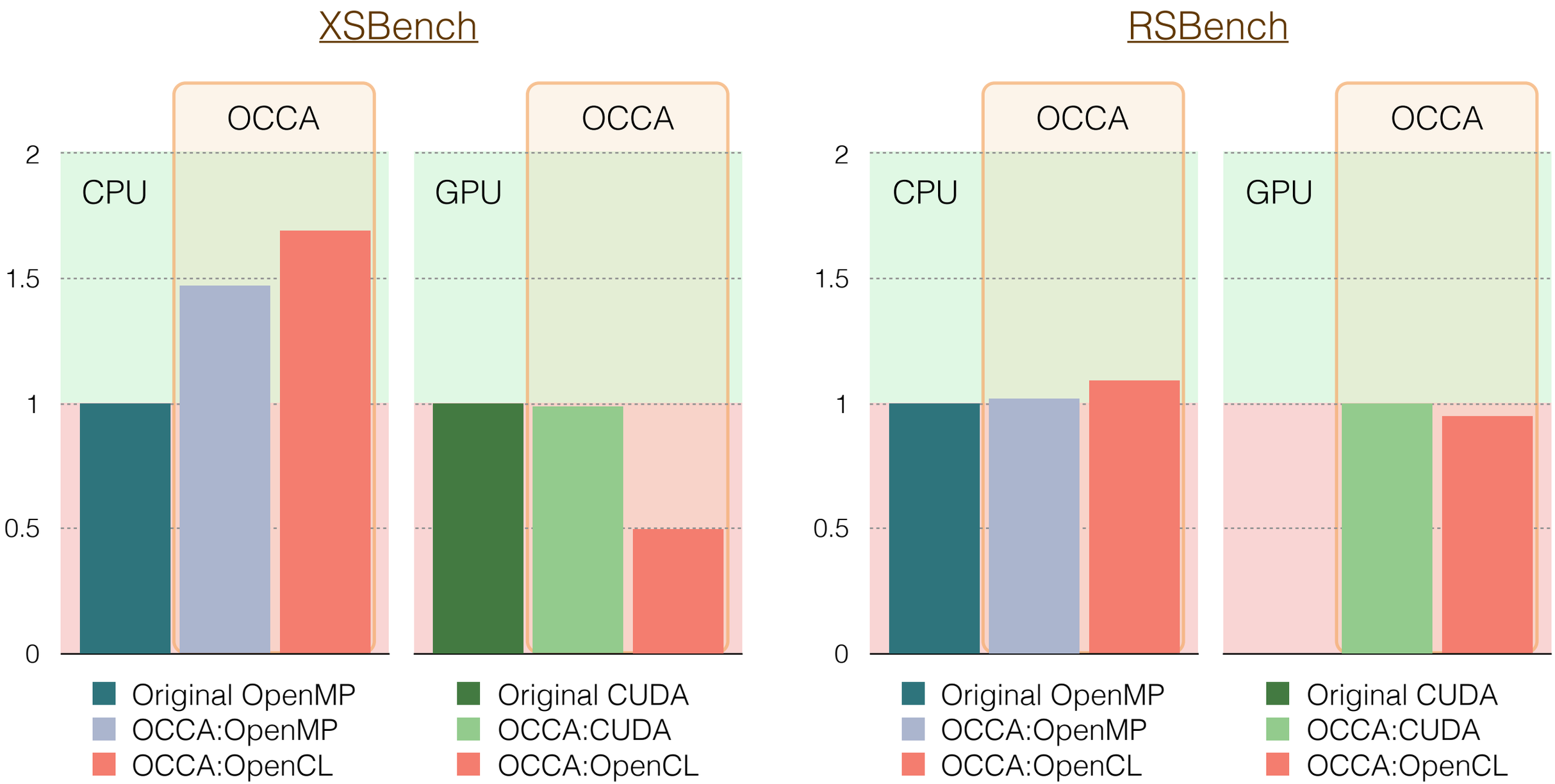
# OCCA: apps & benchmarks

Algebraic multigrid for elliptic problems



# OCCA: apps & benchmarks

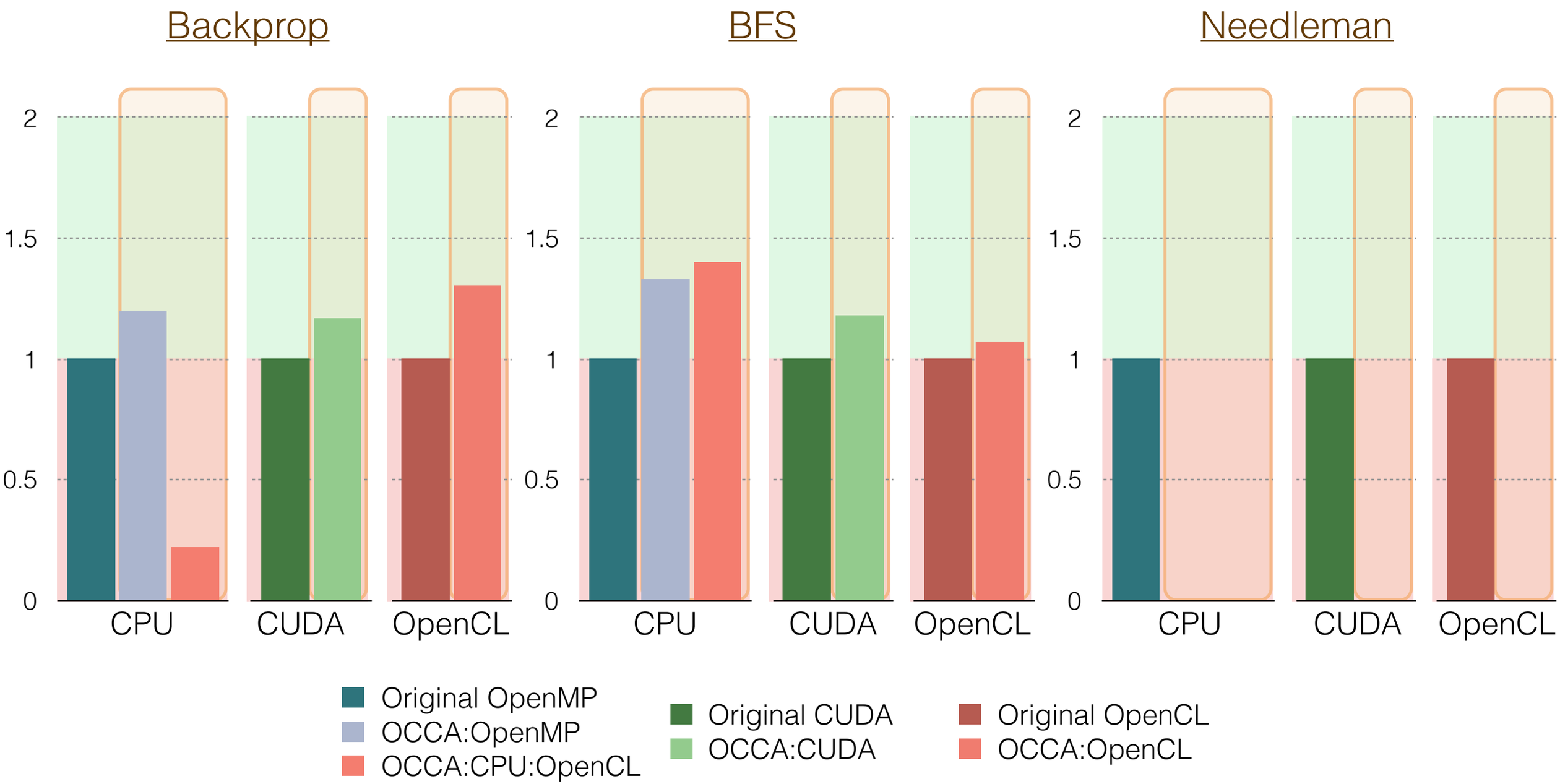
Monte Carlo for neutronics
Collaborations with Argonne National Lab




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”




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

https://github.com/dmed256/OCCA-Benchmarks

tim.warburton@vt.edu

VirginiaTech
Invent the Future®