RICE

Unconventional Wisdom

Accelerators

ATPESC 2015

Tim Warburton
Computational & Applied Mathematics @ Rice University

RTI 95514115

RICE

Unconventional Wisdom

Accelerators

ATPESC 2015

Tim Warburton
Computational & Applied Mathematics @ Rice University

RTI 95514115

RIP: Blinky 08/03/15

Blinky was a MacBook Pro with discrete NVIDIA GPU and OpenMP, OpenCL, CUDA …
Latest versions come with AMD GPU and/or Intel Iris GPU ,,, no CUDA and OpenMP is not default.

Reality Check

It takes more than 3 hours to master GPUs…

Seems like 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 …

Seems like 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 …

Seems like 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.

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

Accelerators: slides & repos

Accelerators: slides & repos

Slides:

http://www.caam.rice.edu/~timwar/ATPESC15

http://bit.ly/1JNLqUx

Examples:

git clone https://github.com/tcew/ATPESC15

OCCA repo:

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

Overview

Part 0: Background on instructor.

Part 1: Processor architecture trends.

- CPU v. GPU.

Part 2: CUDA

- NVIDIA's threaded offload programming model.
- Poisson solver example.

Part 3: Interlude on CUDA optimization.

Part 4: OpenCL

- Open Computing Language

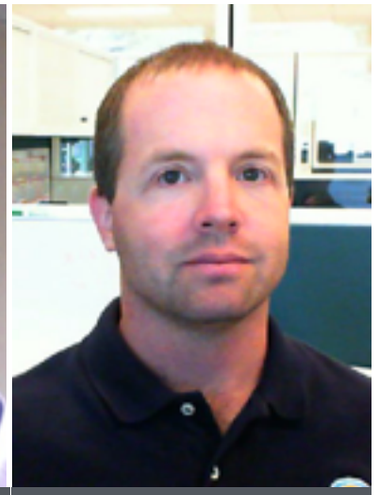
Part 5: OCCA

- Unified and extensible many-core programming model.

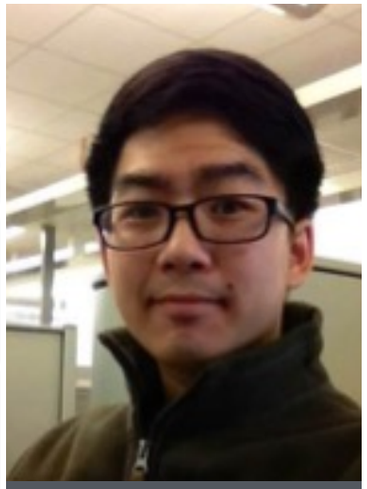
Advanced SCalable ALgorithms Team

I am fortunate to have worked with a team of excellent researchers

JF Remacle
Meshing & Numerics
Sabbatical @ Rice



Amik St-Cyr
Royal Dutch Shell
Adjunct Assoc Prof



Jesse Chan
Advanced Numerics
Post-doc

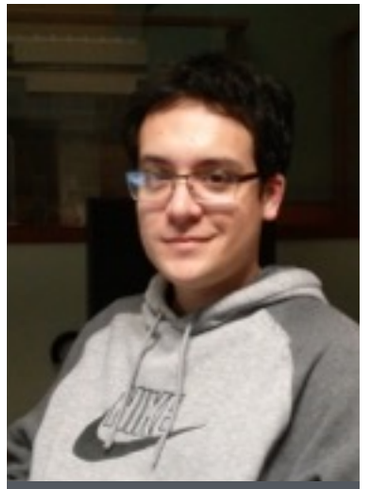
Axel Modave
Accelerate Numerics
Post-doc

Florian Kummer
Multi-phase Flows
Visiting scholar

Ali Karakus
Two Phase Flows
Visiting student

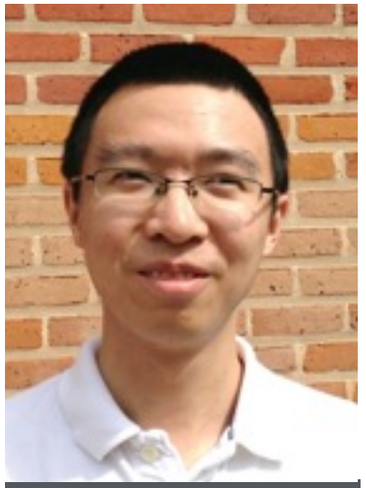
Nichole Stilwell
CFD
CAAM Grad > USAF

Rajesh Gandham
Oceans & AMG
Stoneridge Tech



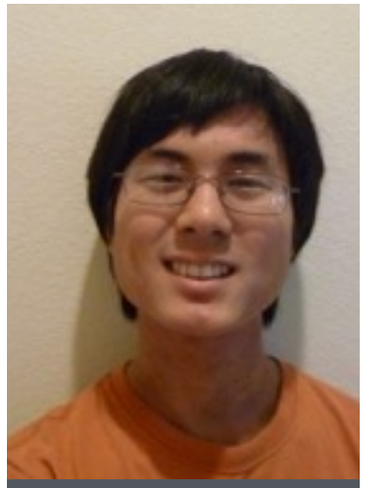
David Medina
HPC & Accelerators
*Two Sigma**

Reid Atcheson
Adaptivity/Helmholtz
Open Geophysical

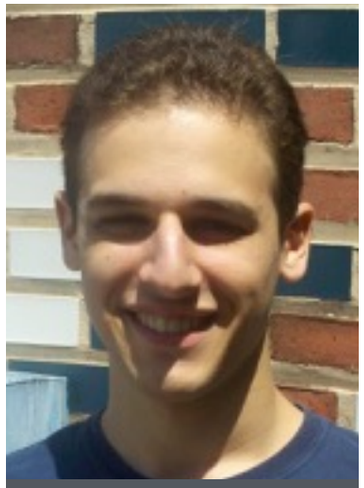


Zheng (Frank) Wang
Wave modeling
CAAM Grad

Arturo Vargas
Hermite+DG
CAAM Grad



Tim Moon
Bioheat modeling
Ugrad => Stanford

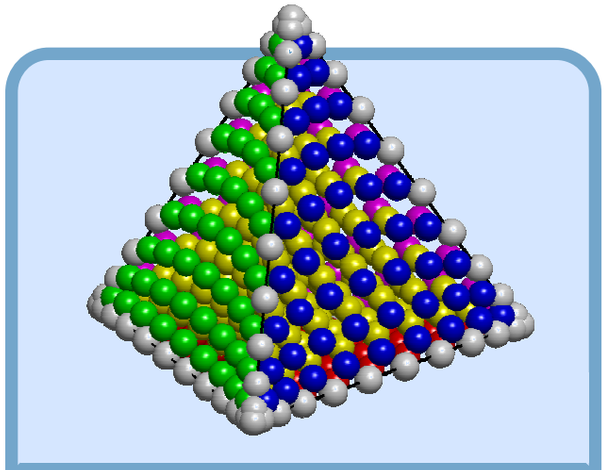


Michael Frano
Bioheat modeling
Ugrad => BP

Industrial internships, projects & fellowships:
Shell, BP, Halliburton, Hess, Stoneridge Technology, Hypercomp, Z-terra, ExxonMobil

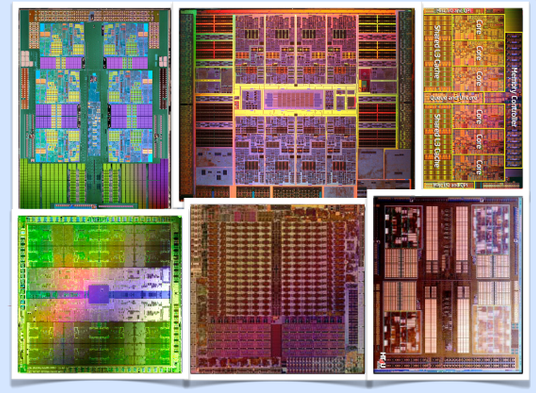
Advanced SCALable Simulations

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



Methods

Numerical Analysis

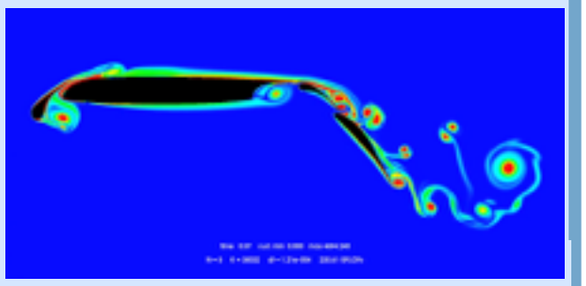


Accelerate

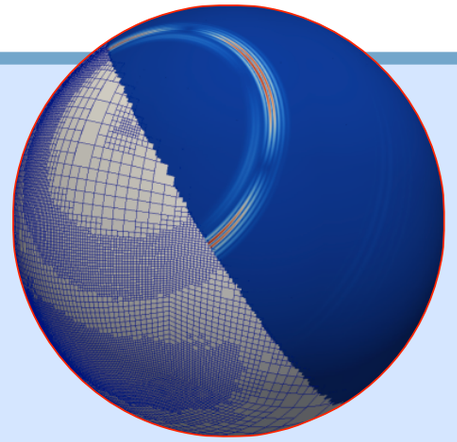
Scale

Electromagnetics

Shallow water



Fluid & gas dynamics



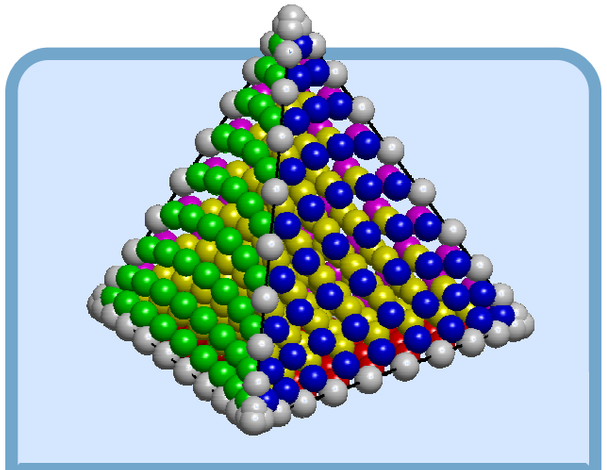
Seismic

All Multi-GPU

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

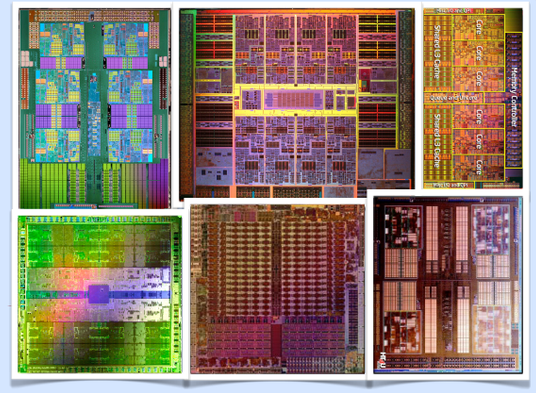
Advanced SCALable Simulations

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



Methods

Numerical Analysis

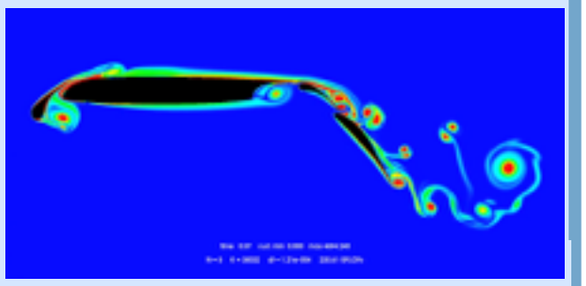


Accelerate

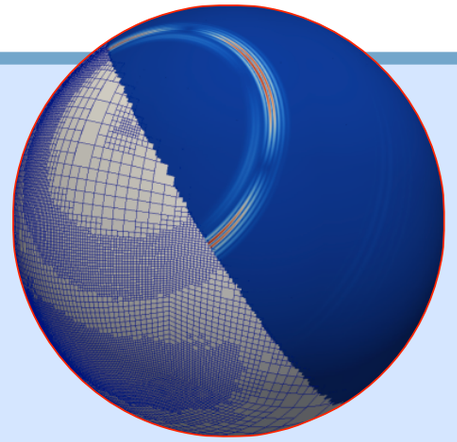
Scale

Electromagnetics

Shallow water



Fluid & gas dynamics



Seismic

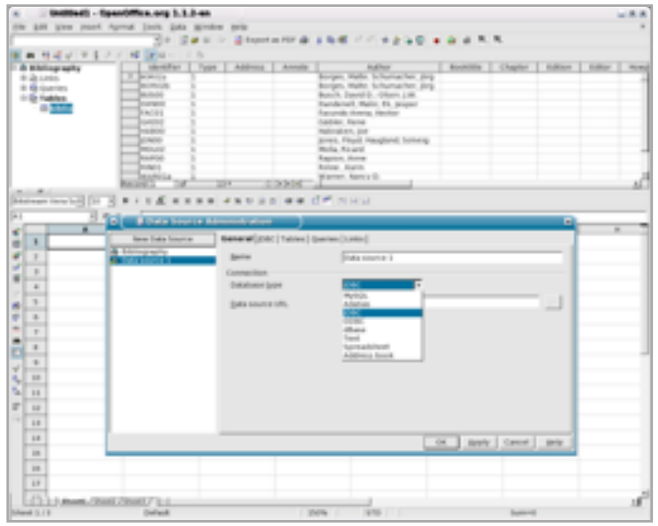
All Multi-GPU

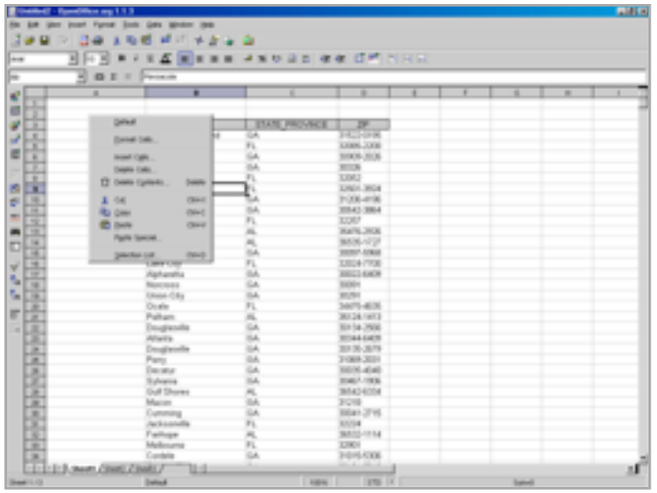
High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.
GPU programming tools & applications. Industrial collaboration.

Part 1: Processor Trends

CPU v. GPU

CPU: architecture follows purpose



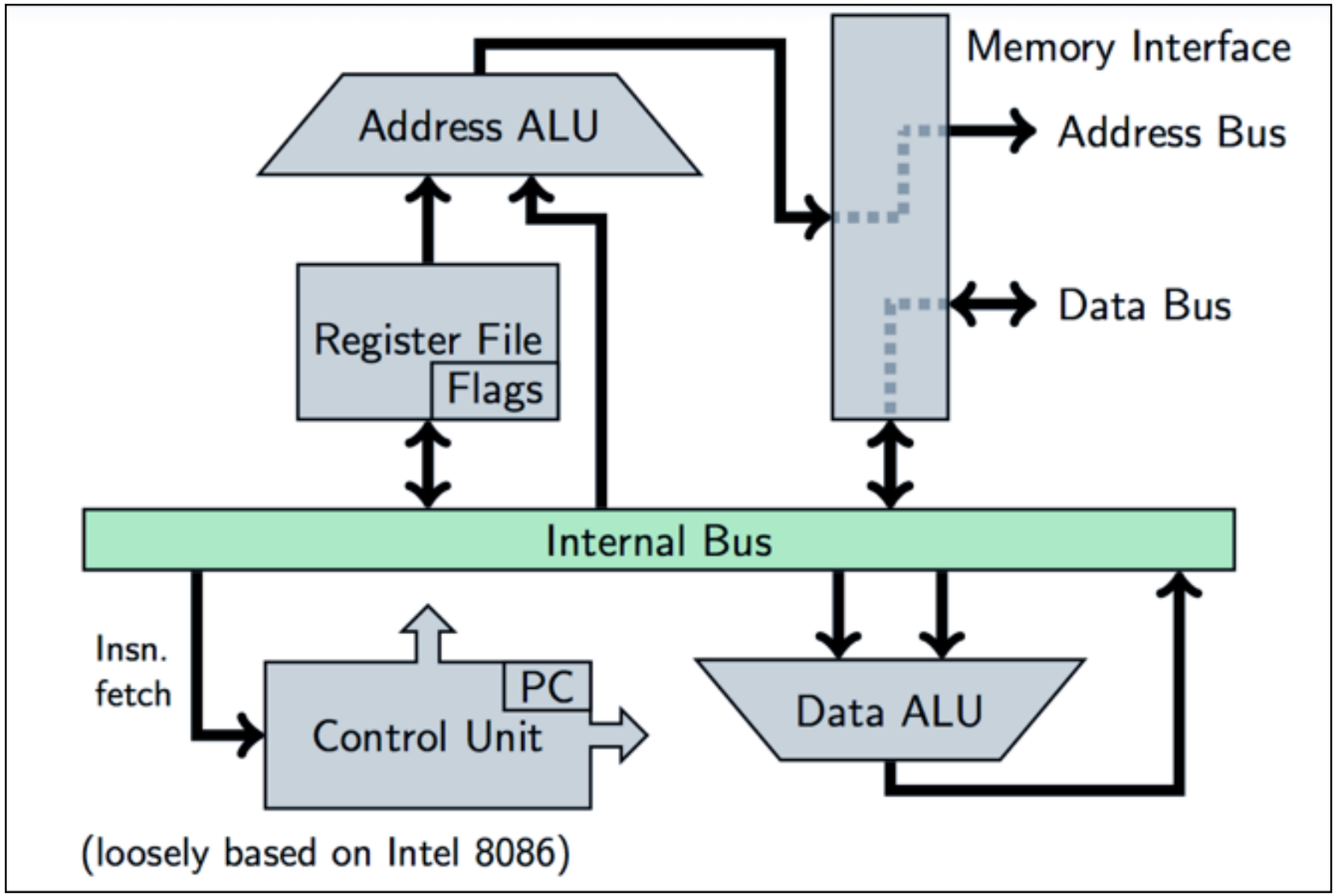


Design goals for CPUs:

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

CPU: simplified core architecture

A loose diagramatic representation of a basic processing core.



From: NYU Lecture notes by Klöckner

Core: minimally a control unit with an independent instruction stream, arithmetic logic units, register, memory interface.

CPU: simplified core architecture

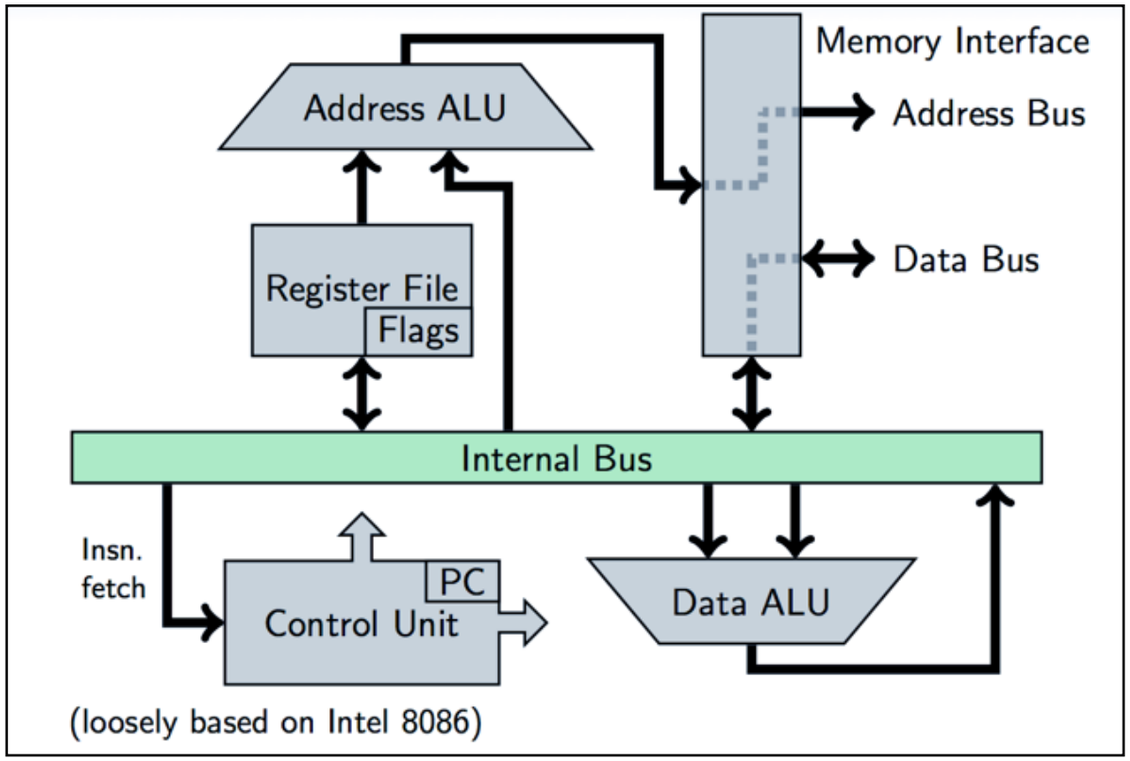
Everything synchronizes to the *Clock*.

Control Unit ("CU"): The brains of the operation. Everything connects to it.

CU controls gates, tells other units about 'what' and 'how':

- What operation?
- Which register?
  - Which addressing mode?

Bus entries/exits are gated and (potentially) buffered.



From: NYU Lecture notes by Klöckner

CPU: arithmetic logic unit (ALU)

ALUs are the basic compute units on a core.

One or two operands A, B

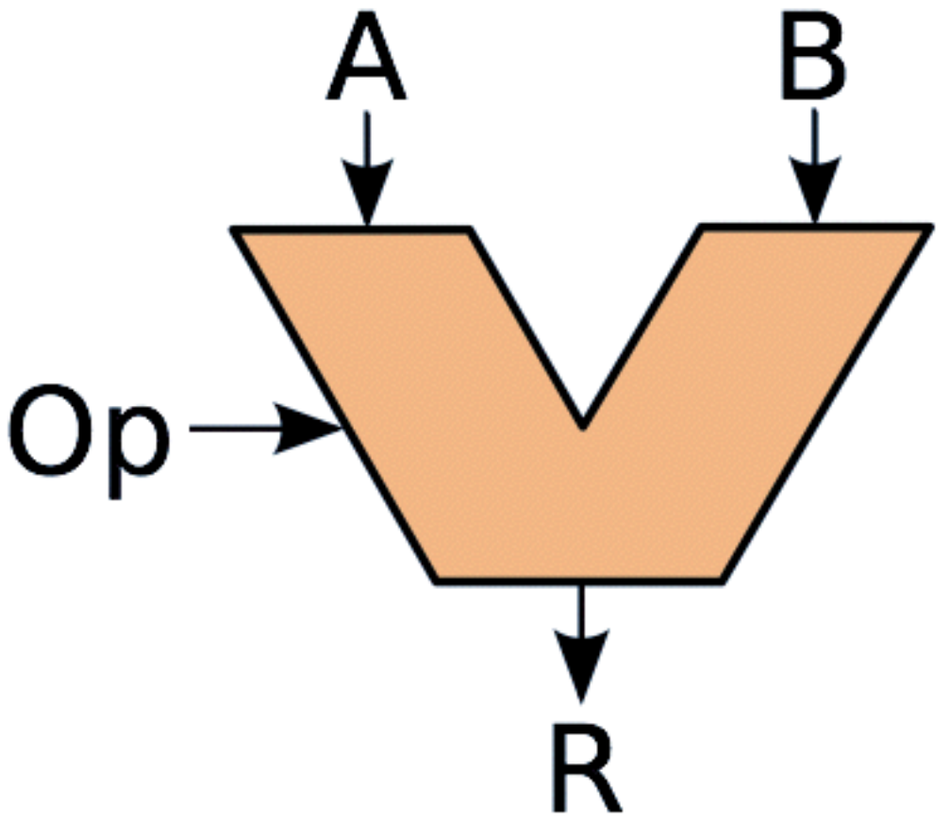
Operation selector (Op):

- (Integer) Addition, Subtraction.
- (Logical) And, Or, Not.
- (Bitwise) Shifts.
- (Integer) Multiplication, Division.

Specialized ALUs:

- Floating Point Unit (FPU).
- Address ALU.

Operates on binary representations of numbers.

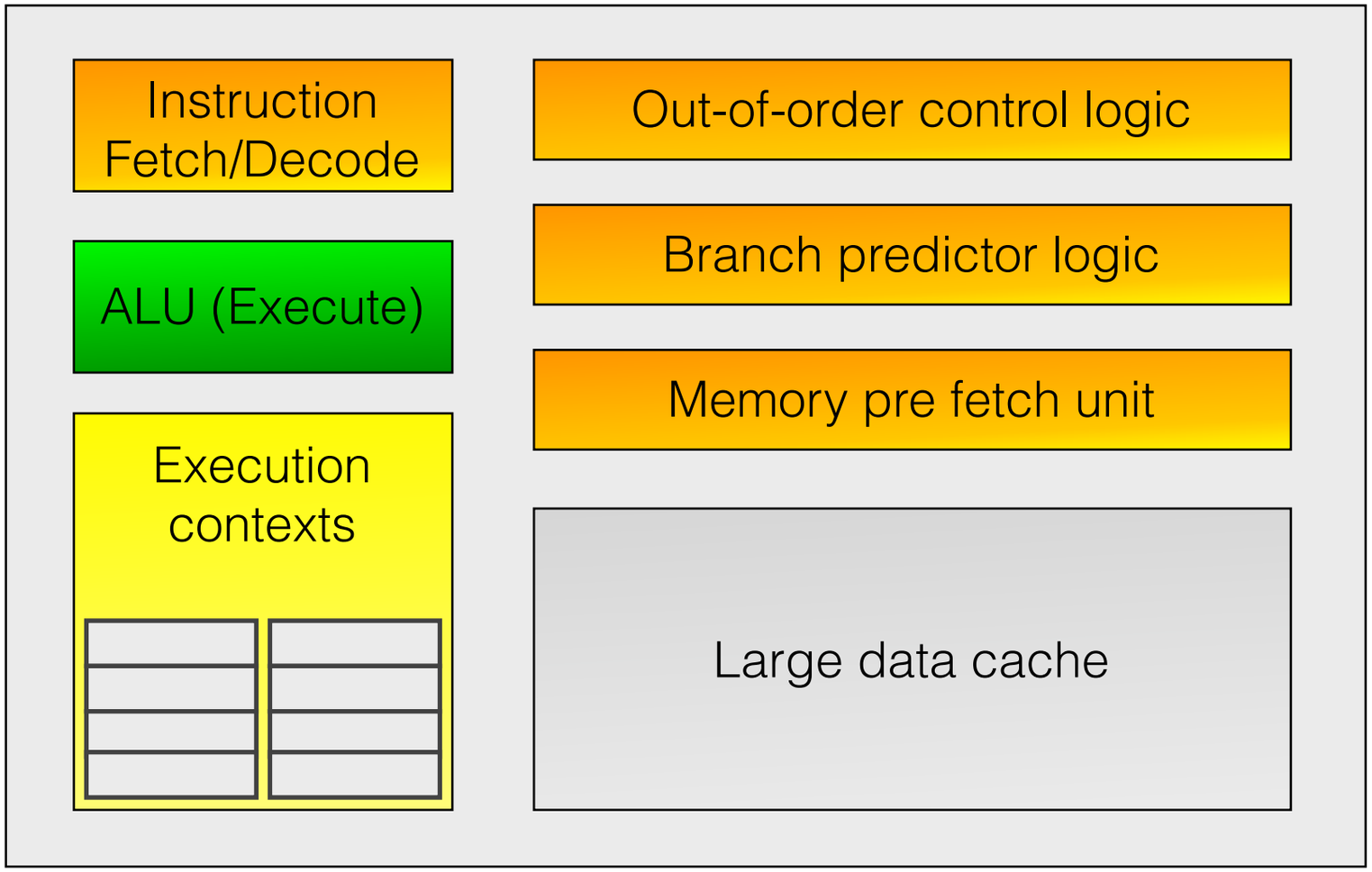


From: NYU Lecture notes by Klöckner

An ALU is not a core.

CPU: abstract modern architecture

Modern "CPU-Style" core design emphasizes individual thread performance.

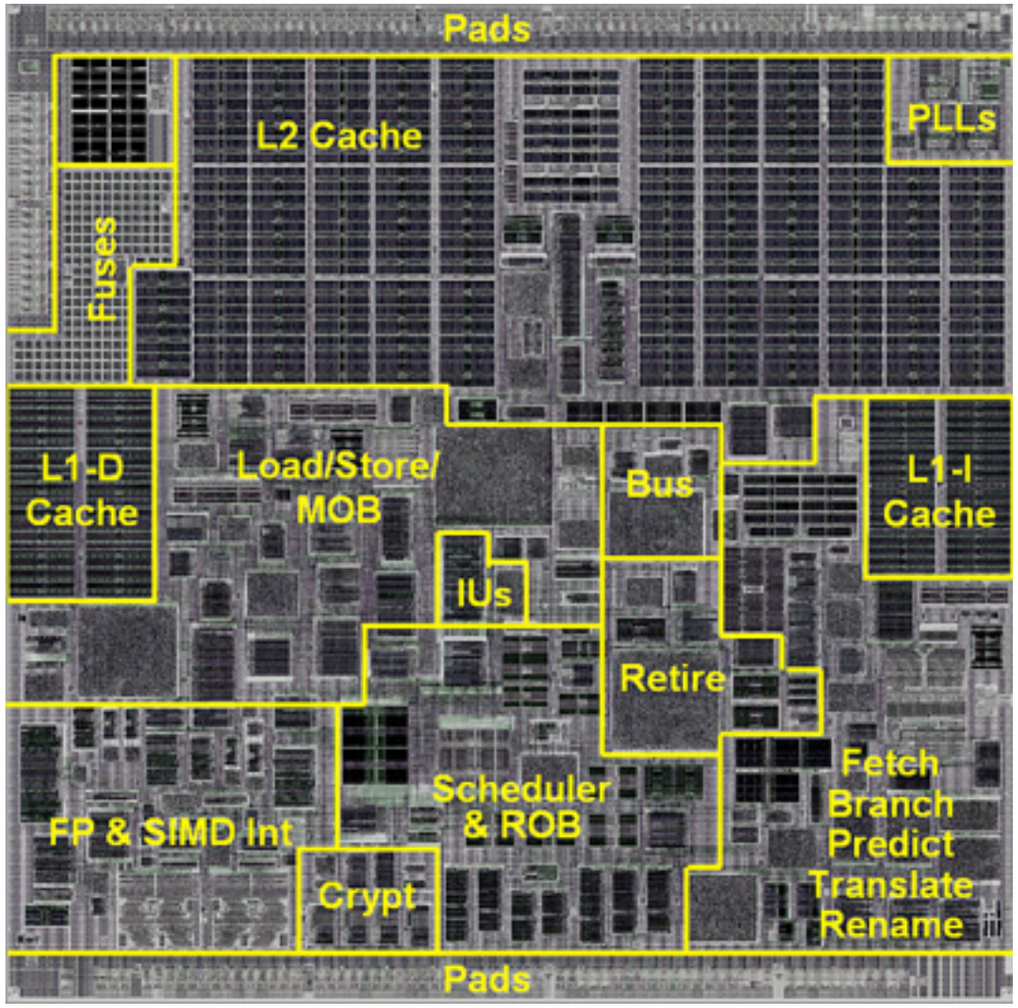


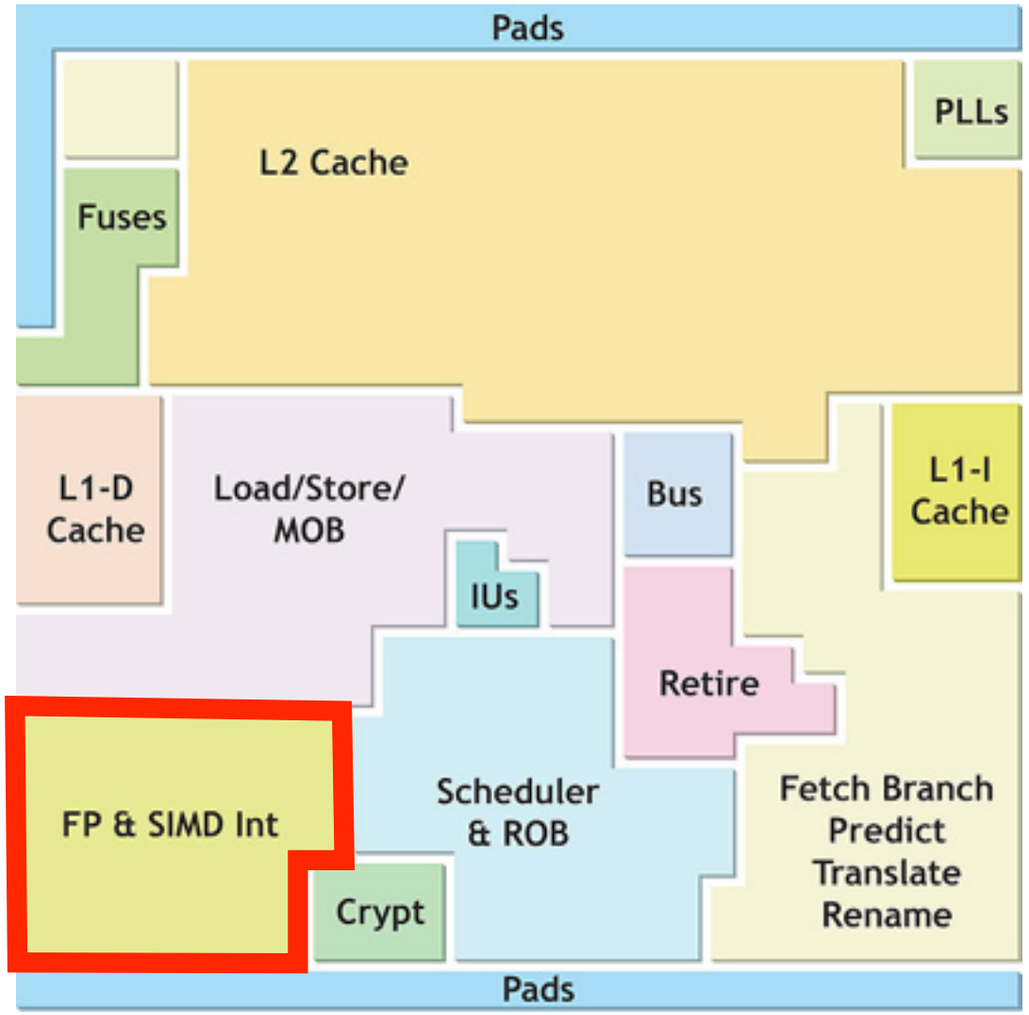
Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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

CPU: example die partition

Older single core processor example.



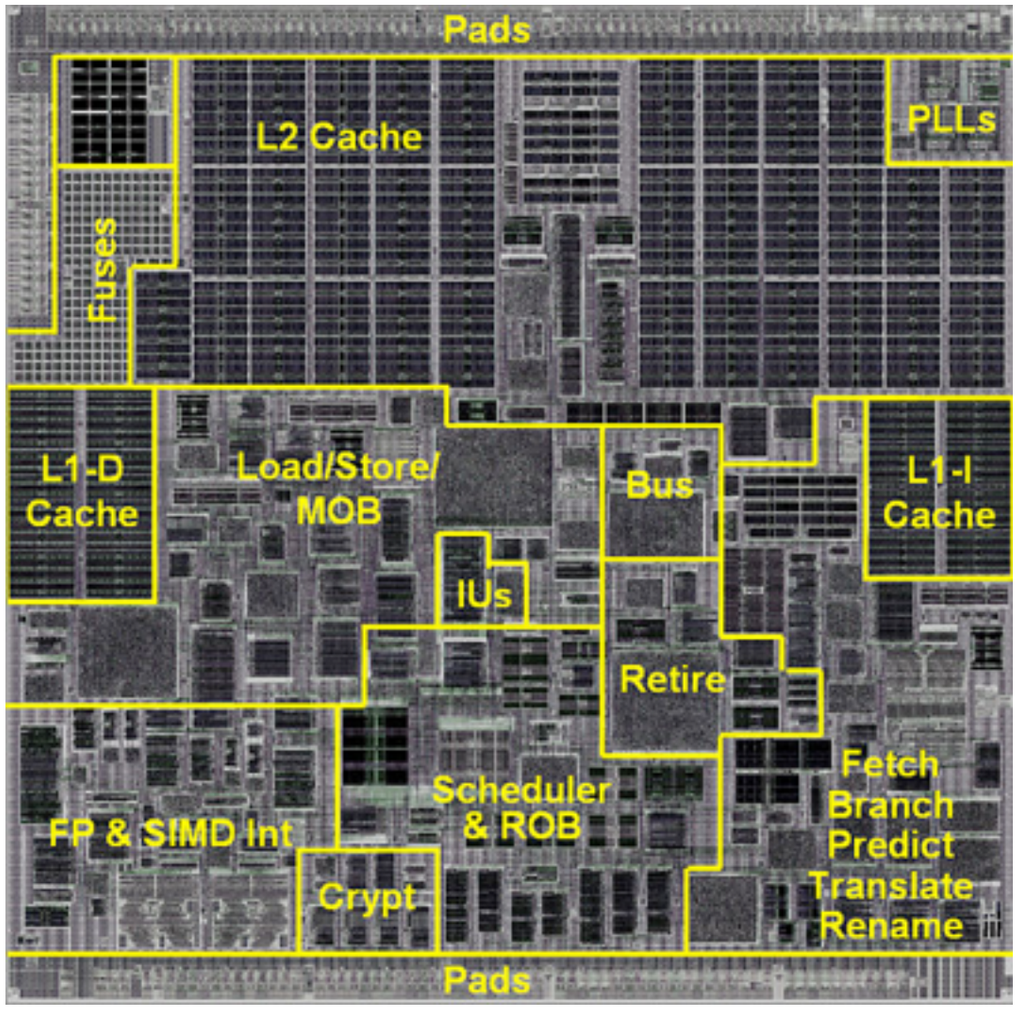


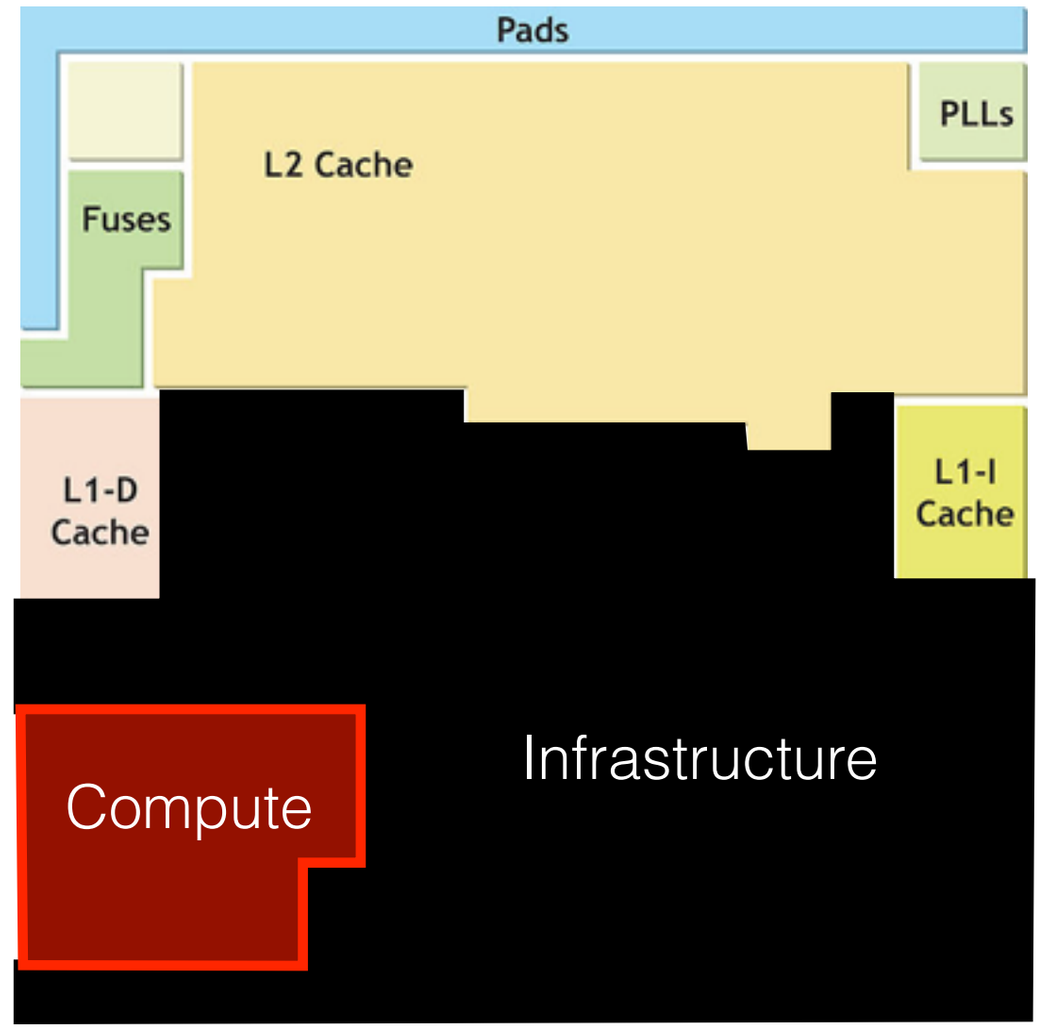
wiki source

Die floorplan: VIA Isaiah (2008). 65 nm, 4 SP ops at a time, 1 MiB L2.

CPU: top heavy management

Older single core processor example.





wiki source

Only a small portion of the chip footprint is devoted to computation.
Most of the silicon is devoted to managing the flow of instructions.

GPU: massively parallel compute

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 compute

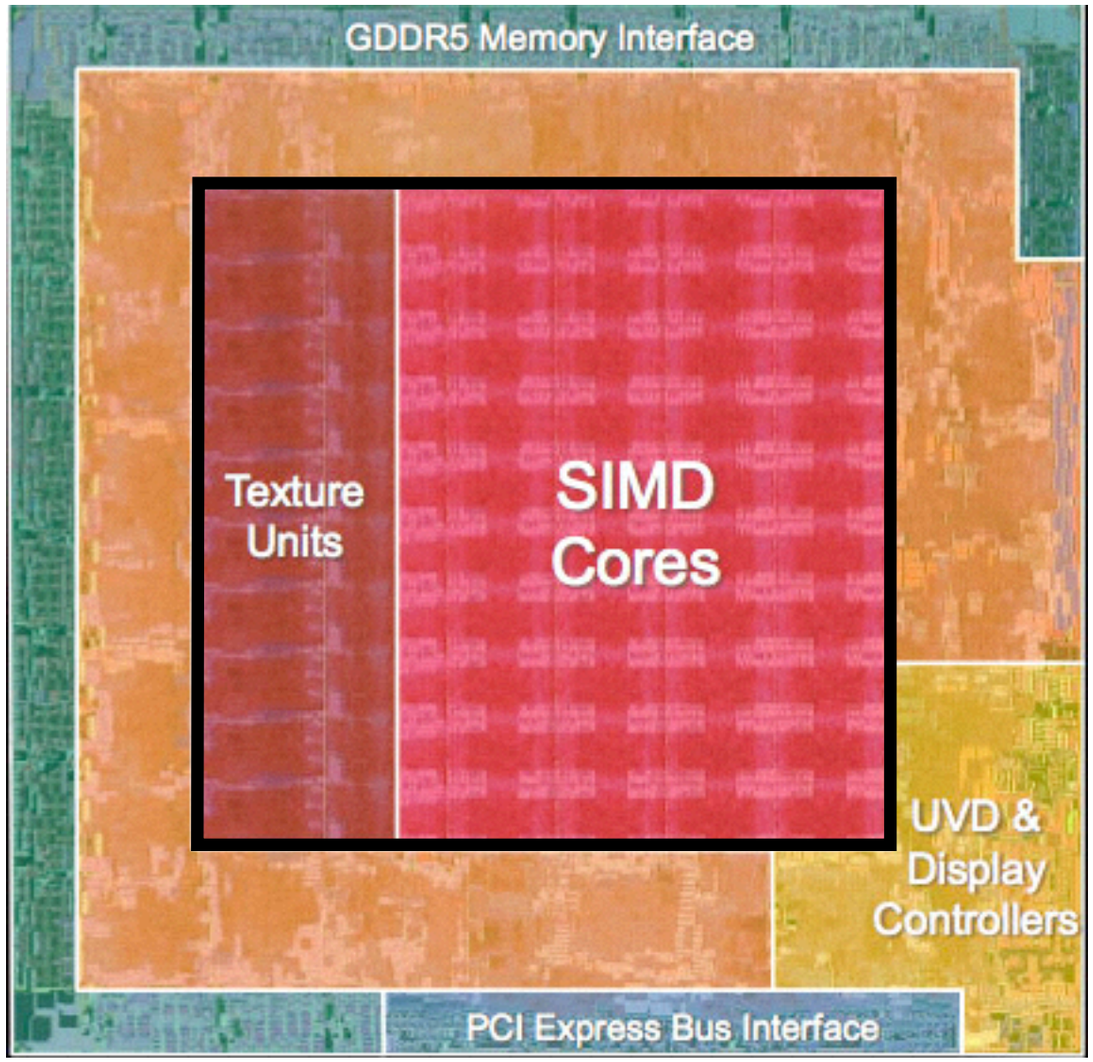
Design goals for GPUs:

- Throughput matters and single threads do not.
- Hide memory latency through parallelism.
- Let programmer deal with “raw” storage hierarchy.
- Avoid high frequency clock speed.

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

GPU: early example

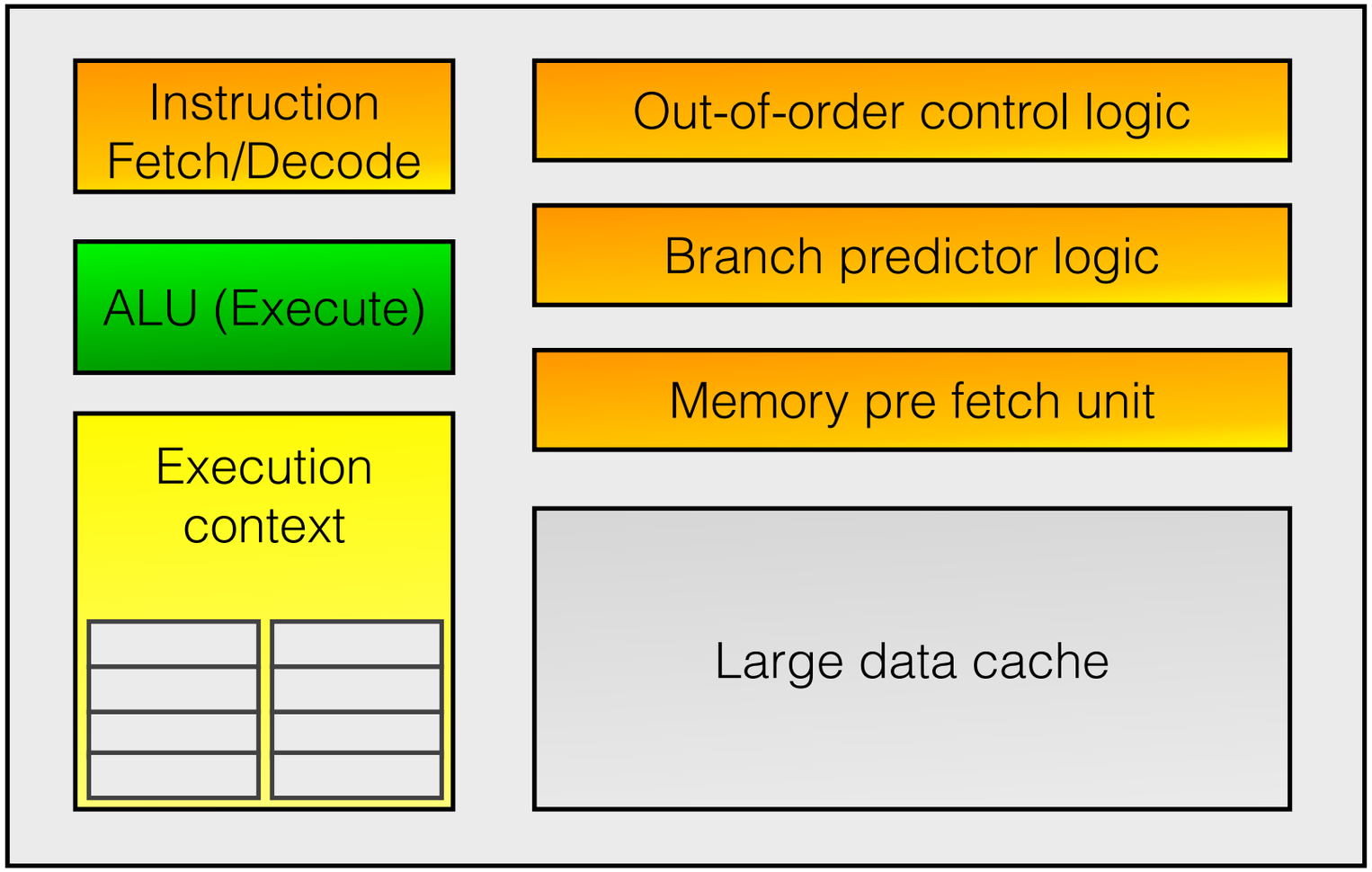
Die floorplan: AMD RV770 (2008) 55 nm, 800 SP ops at a time
The majority of the silicon is devoted to computation



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.

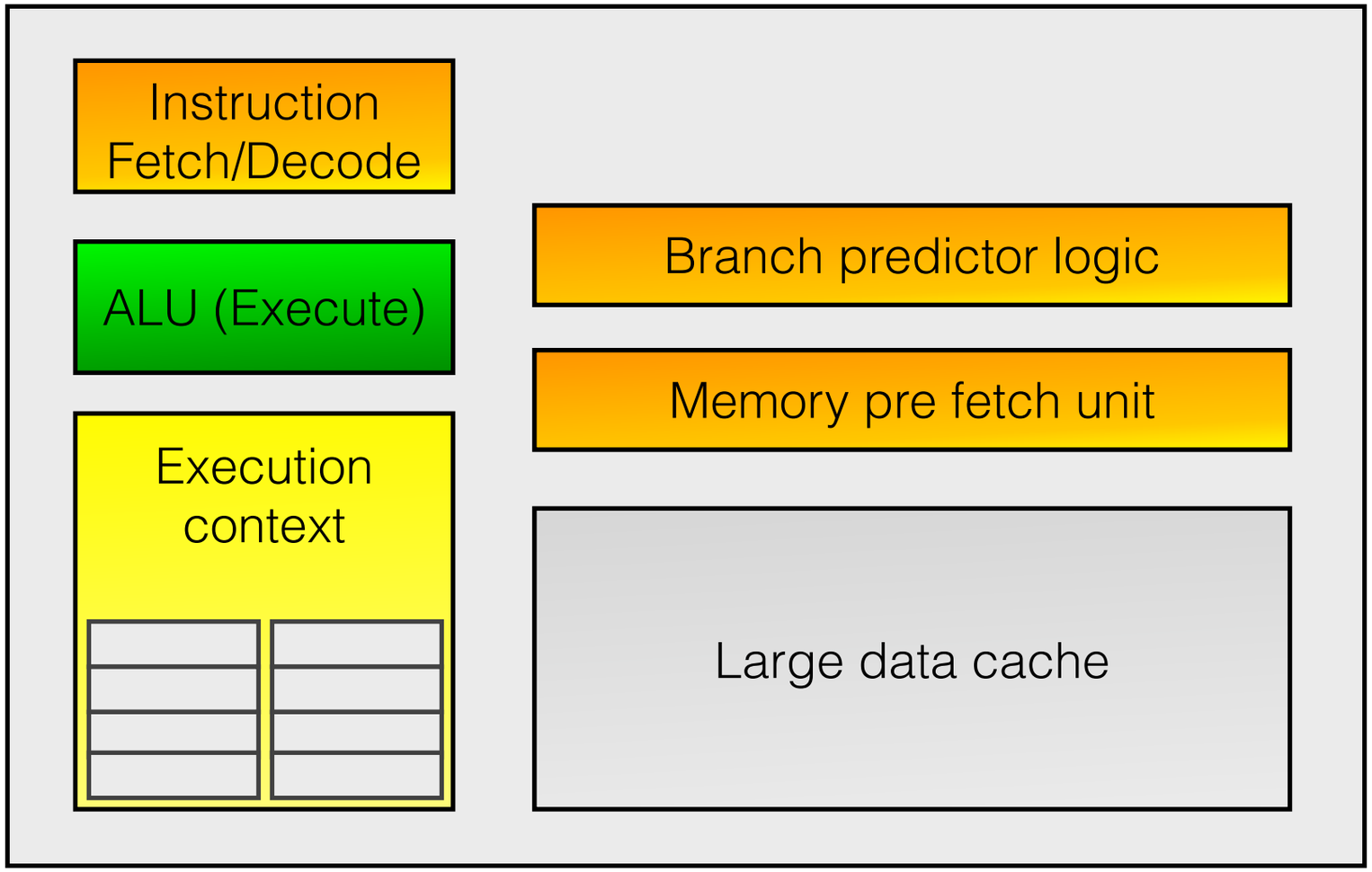


Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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.

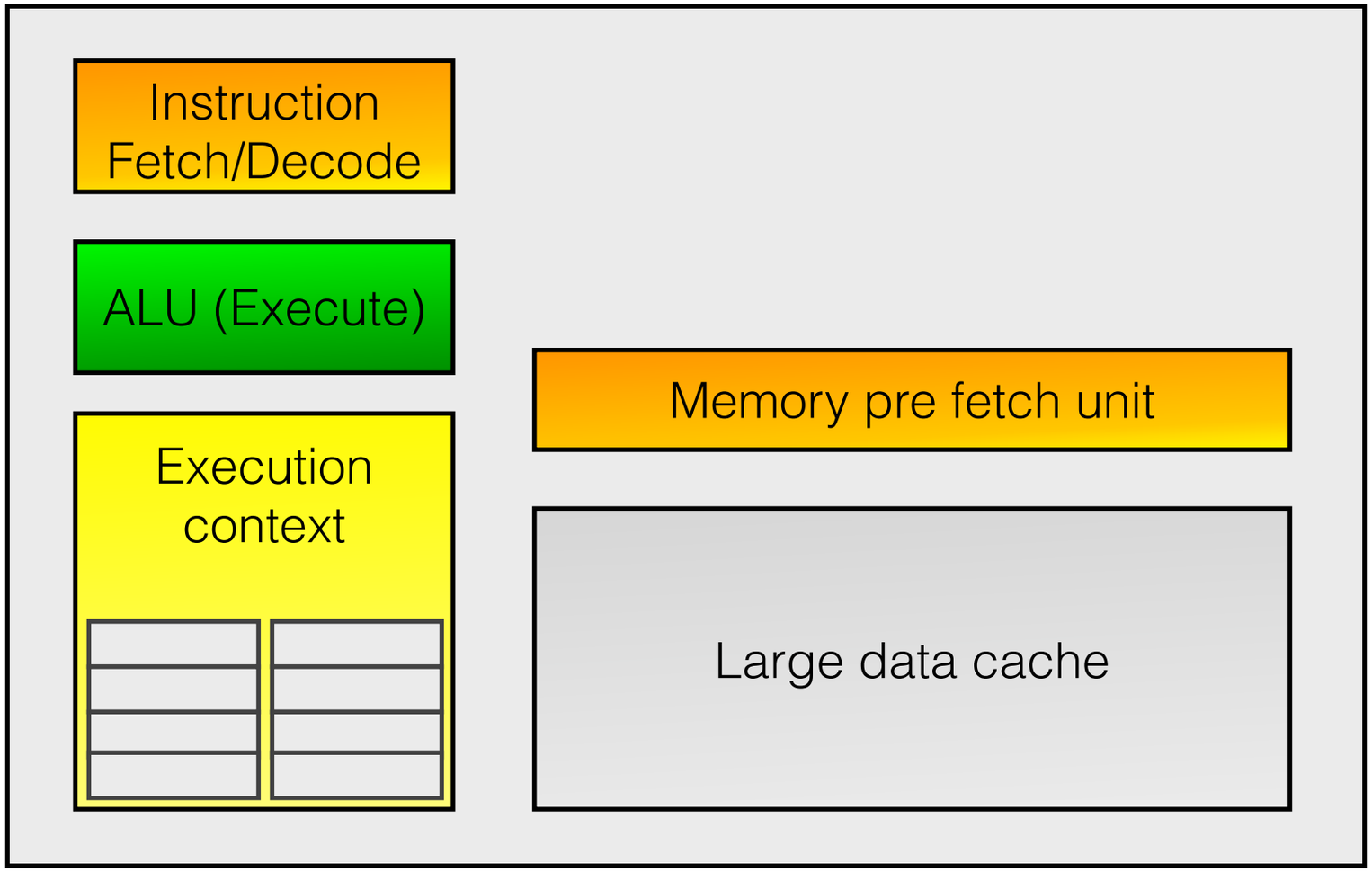


Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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.

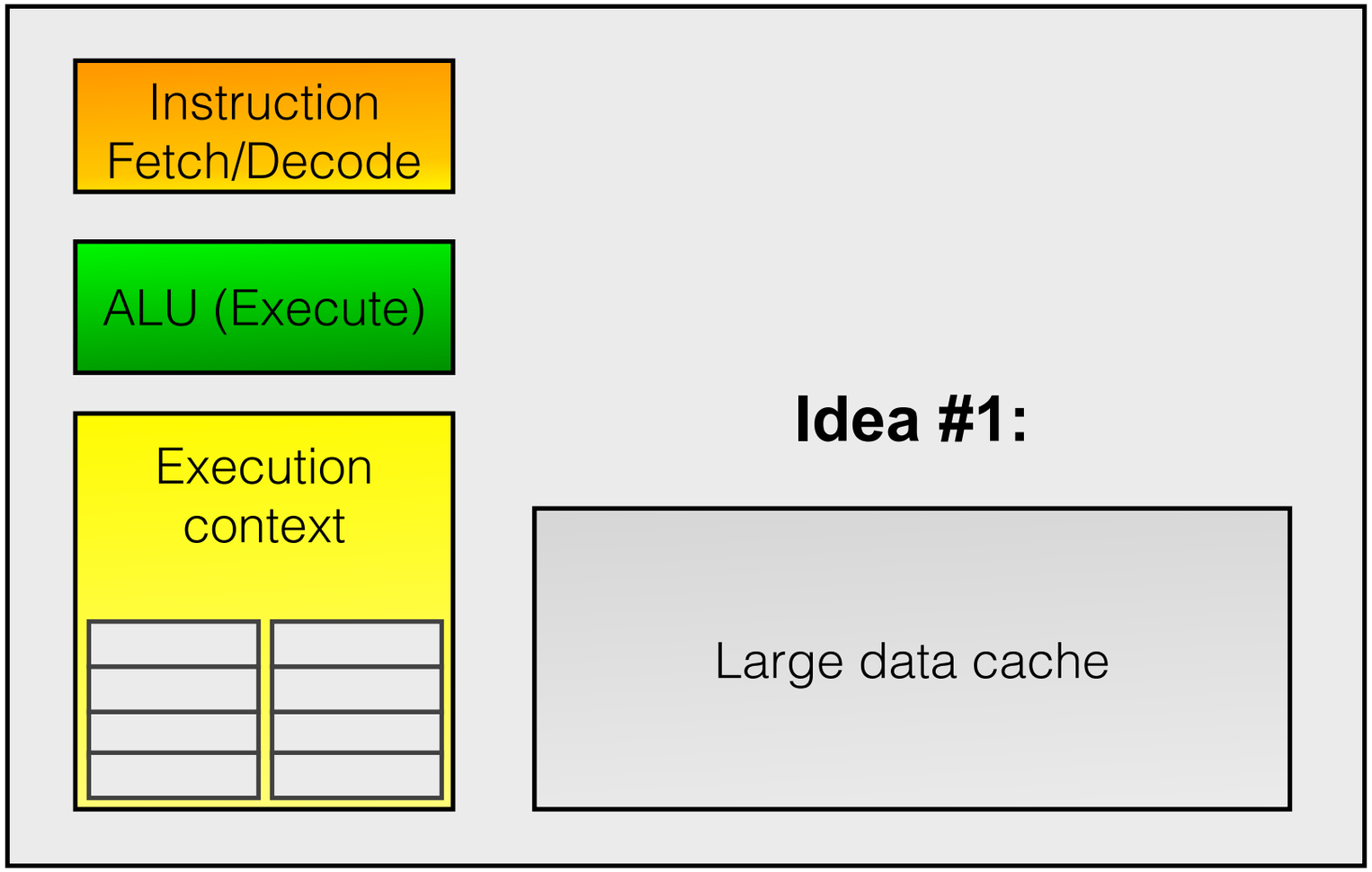


Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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.

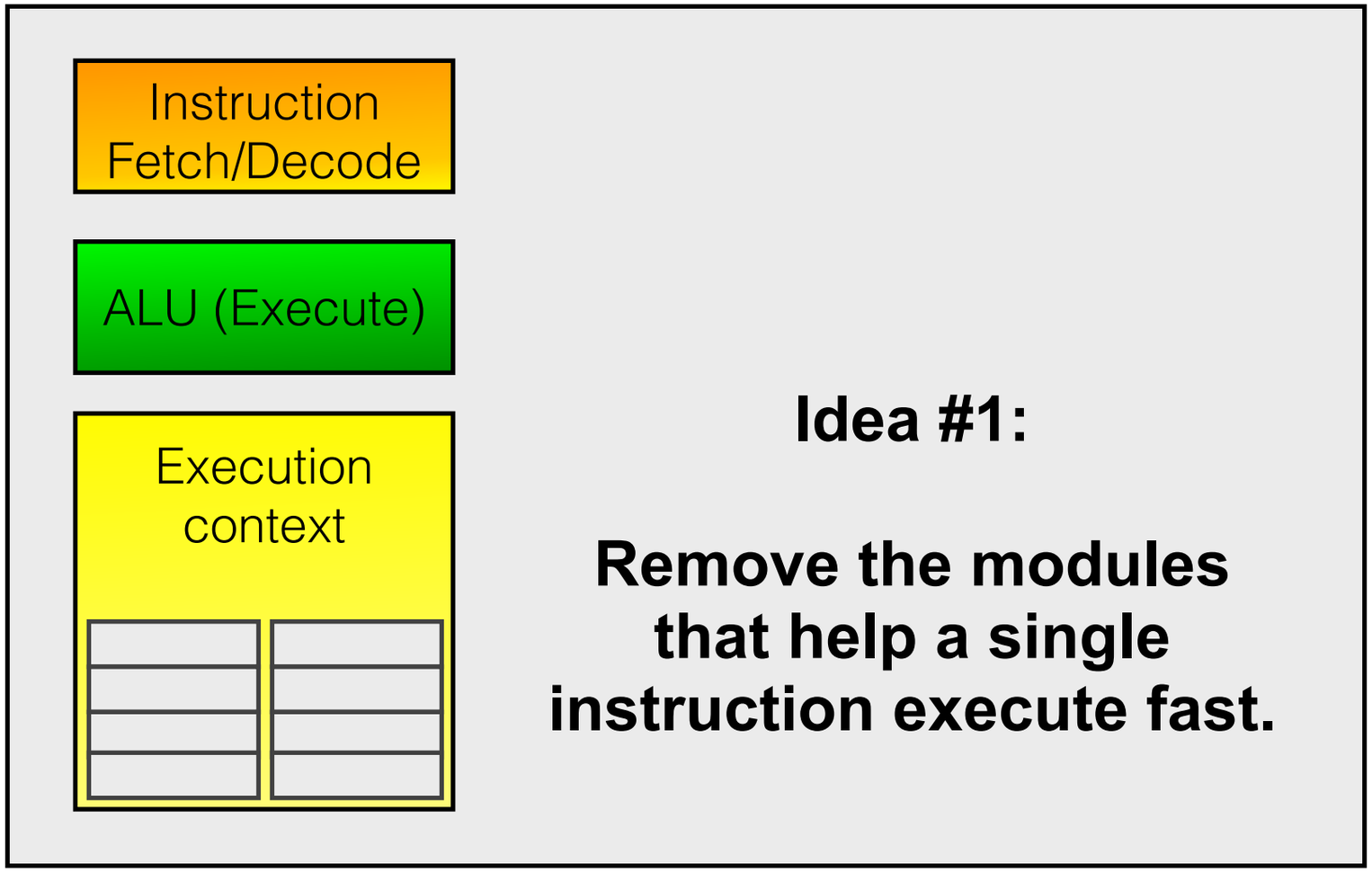


Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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.

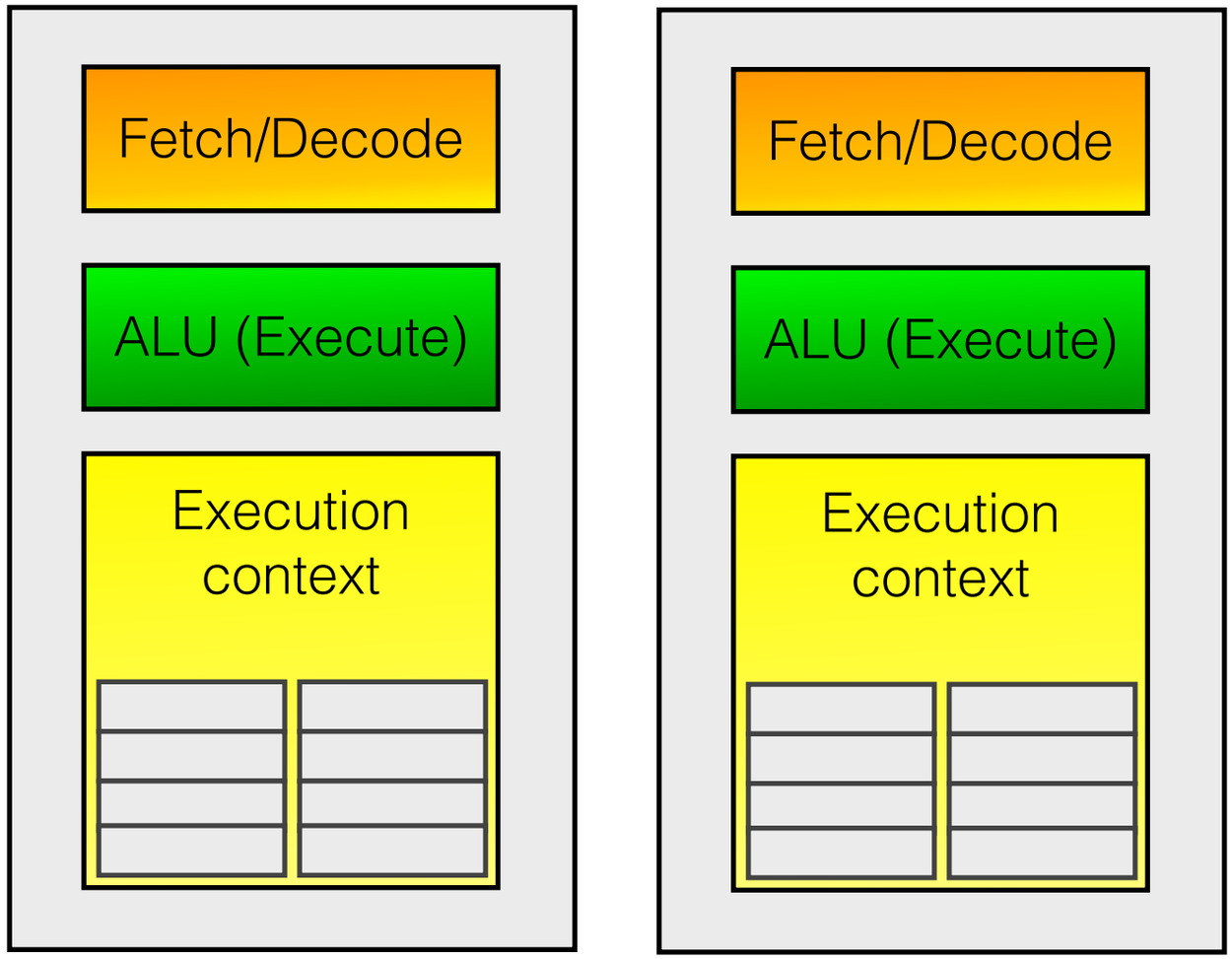


Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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

GPU: replicate cores

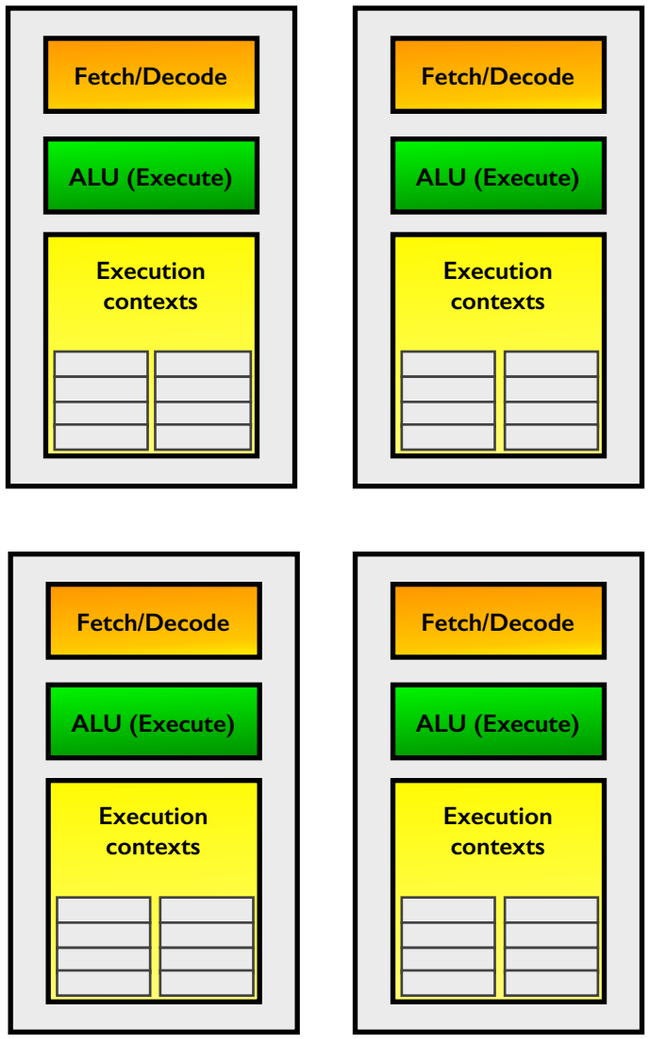
Each simplified core takes less space so double up



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)

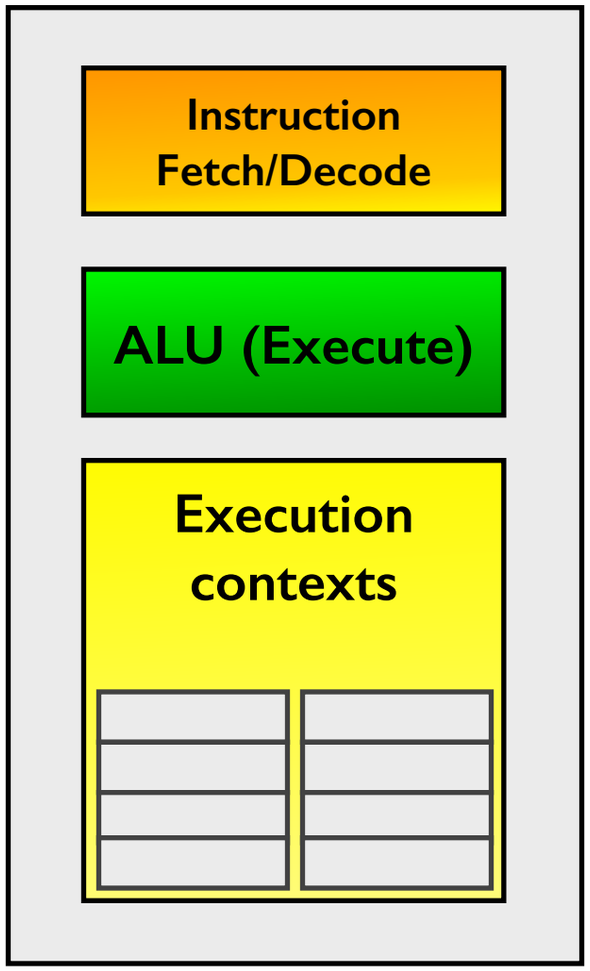
... and again to yield 16 independent instruction streams ...

But: GPU rendering instruction streams are typically very similar.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

GPU: reduce # of instruction streams

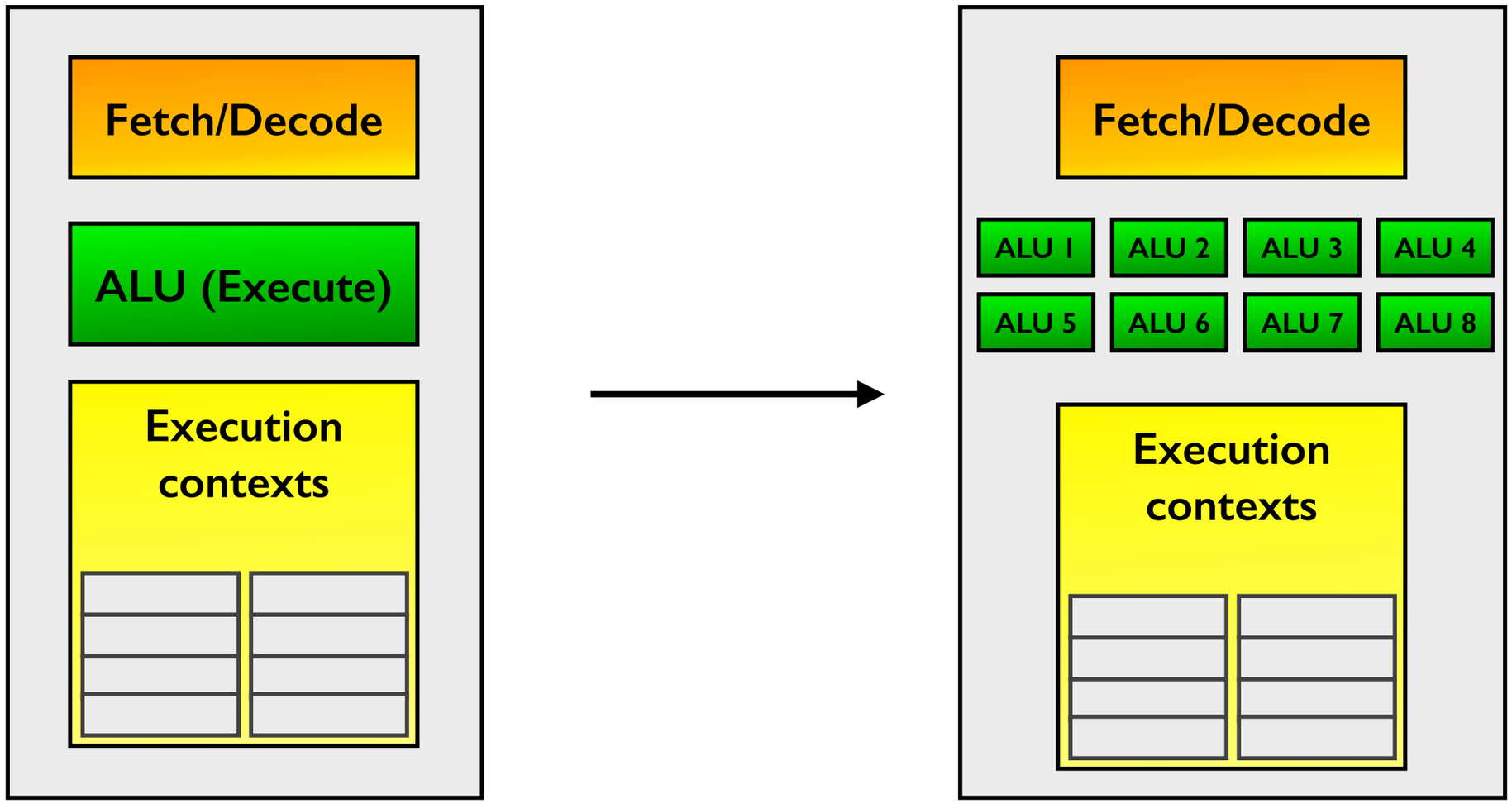
The Fetch/Decode units in those 16 clone cores are likely fetching and decoding the same instruction streams for rendering graphics...



Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

GPU: SIMD groups

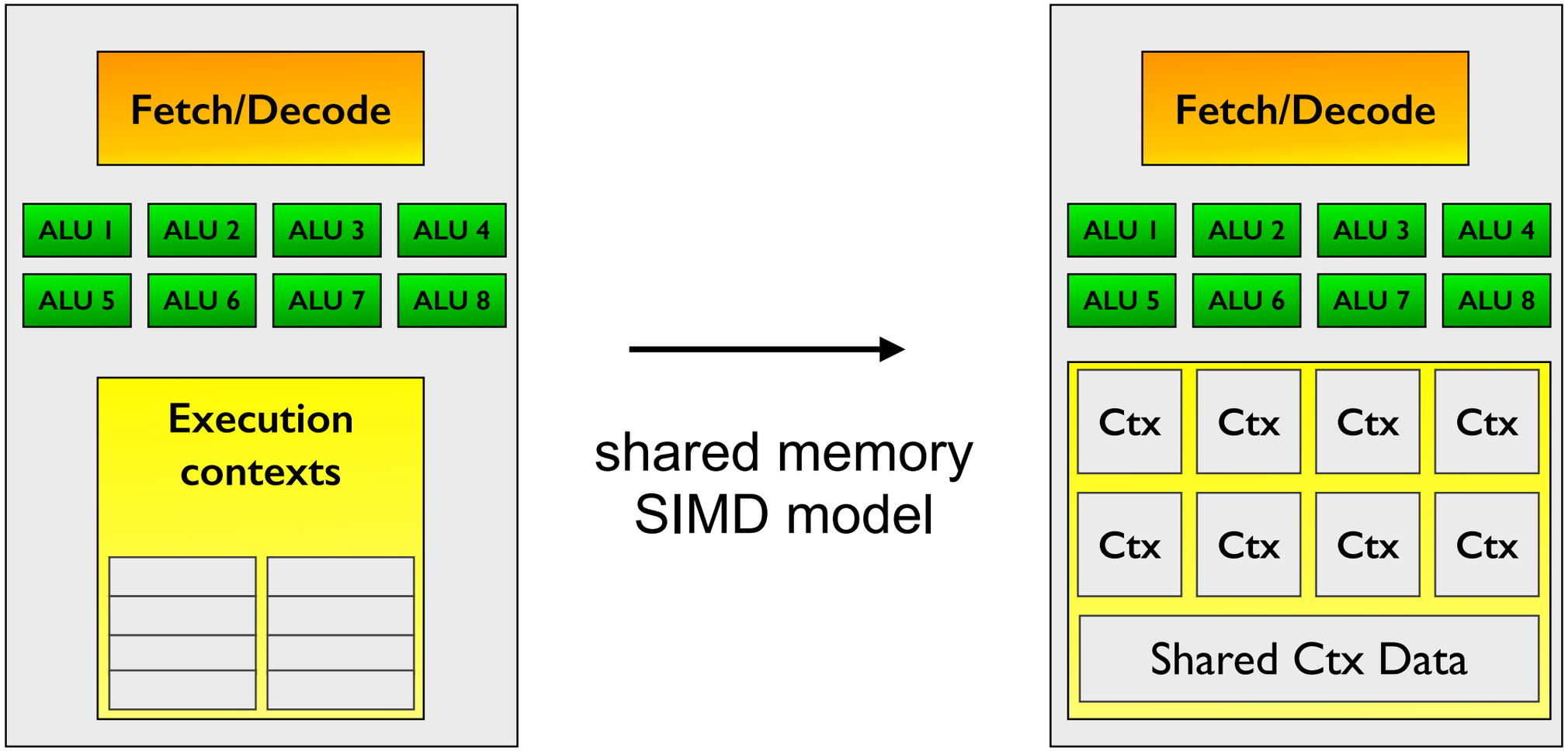
"single instruction multiple data" SIMD model:
share the cost of the instruction stream across many ALUs



Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

GPU: shared memory for SIMD group

Provision each SIMD group with shared memory



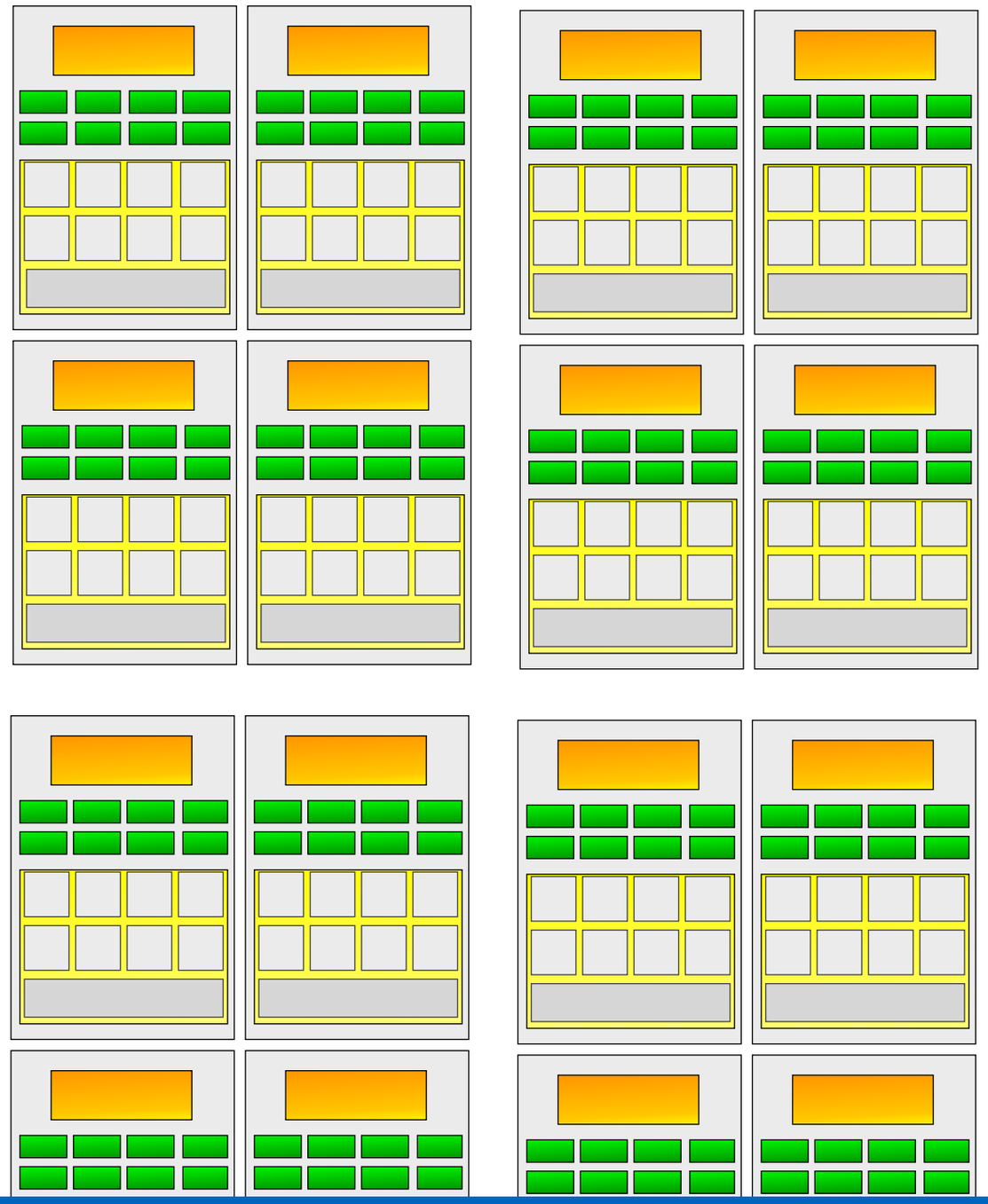
Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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 groups, each with 8 synchronized streams

The cores are cloned into a massively parallel processor.

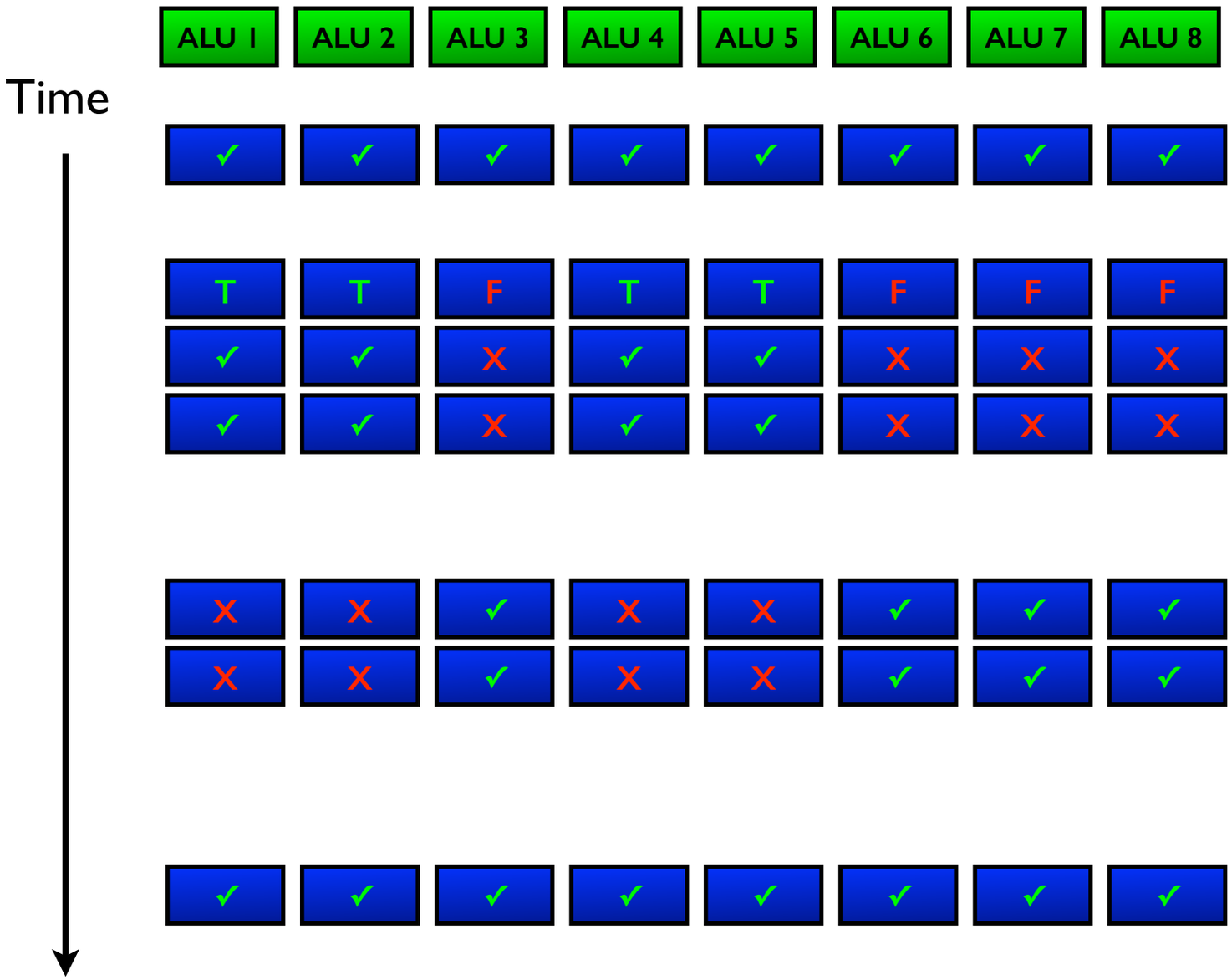


Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

This is an abstract GPU [memory system not shown]

GPU: branching with SIMD

One instruction unit per core impacts instruction branching.



Non branching code;

```
if(flag > 0){ /* branch */
  x = exp(y);
  y = 2.3*x;
}
else{
  x = sin(y);
  y = 2.1*x;
}
```

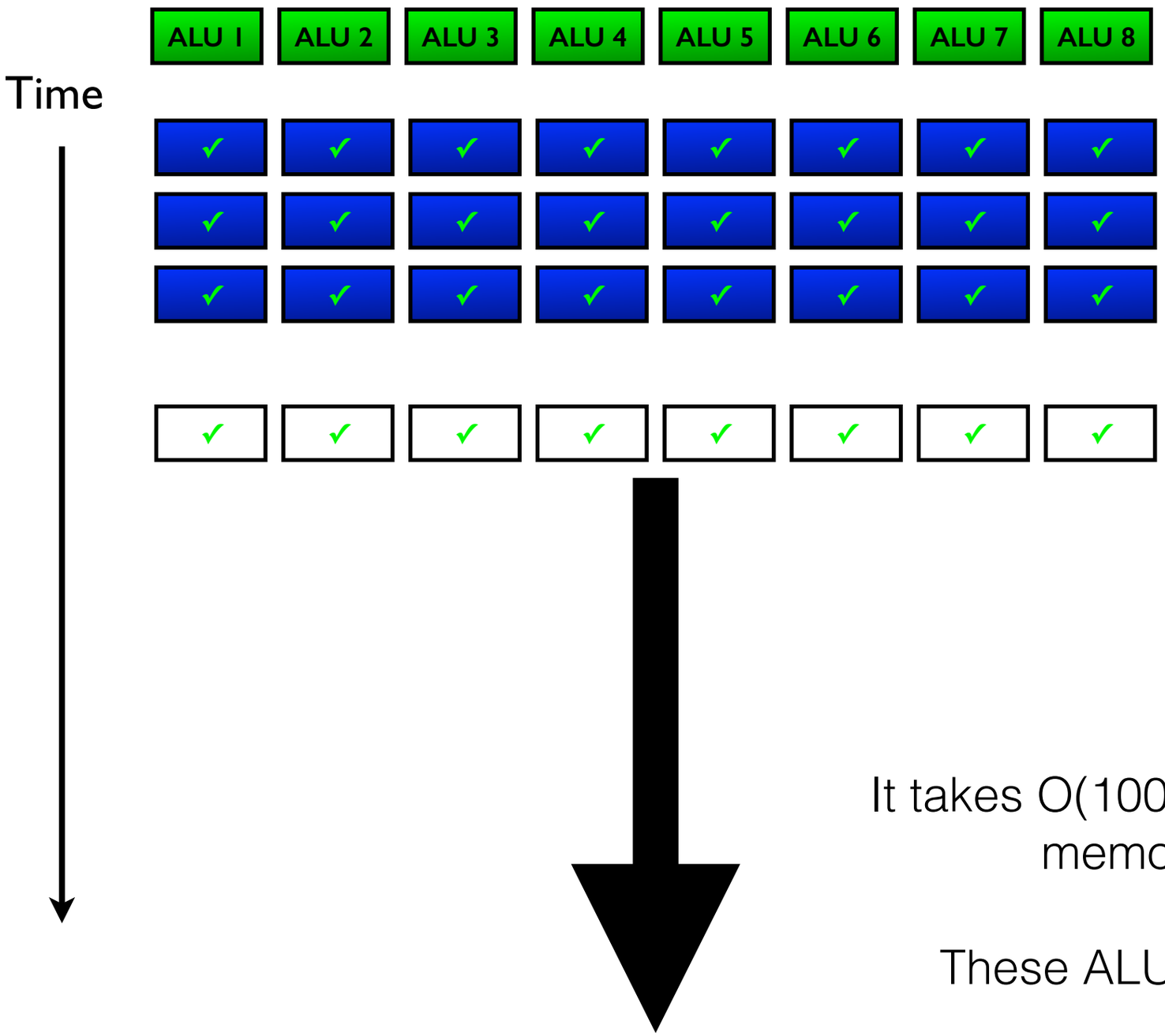
Non branching code;

Idle ALUs execute NOPs when code branches.
In this example we could see 1/8 of maximum performance.

Slides and graphics based on presentations from Andreas Klöckner and Kayvon Fatahalian

GPU: high latency memory fetches

Accessing off-chip GPU memory takes a long time (partly latency)



work on registers;
work on registers;
work on registers;

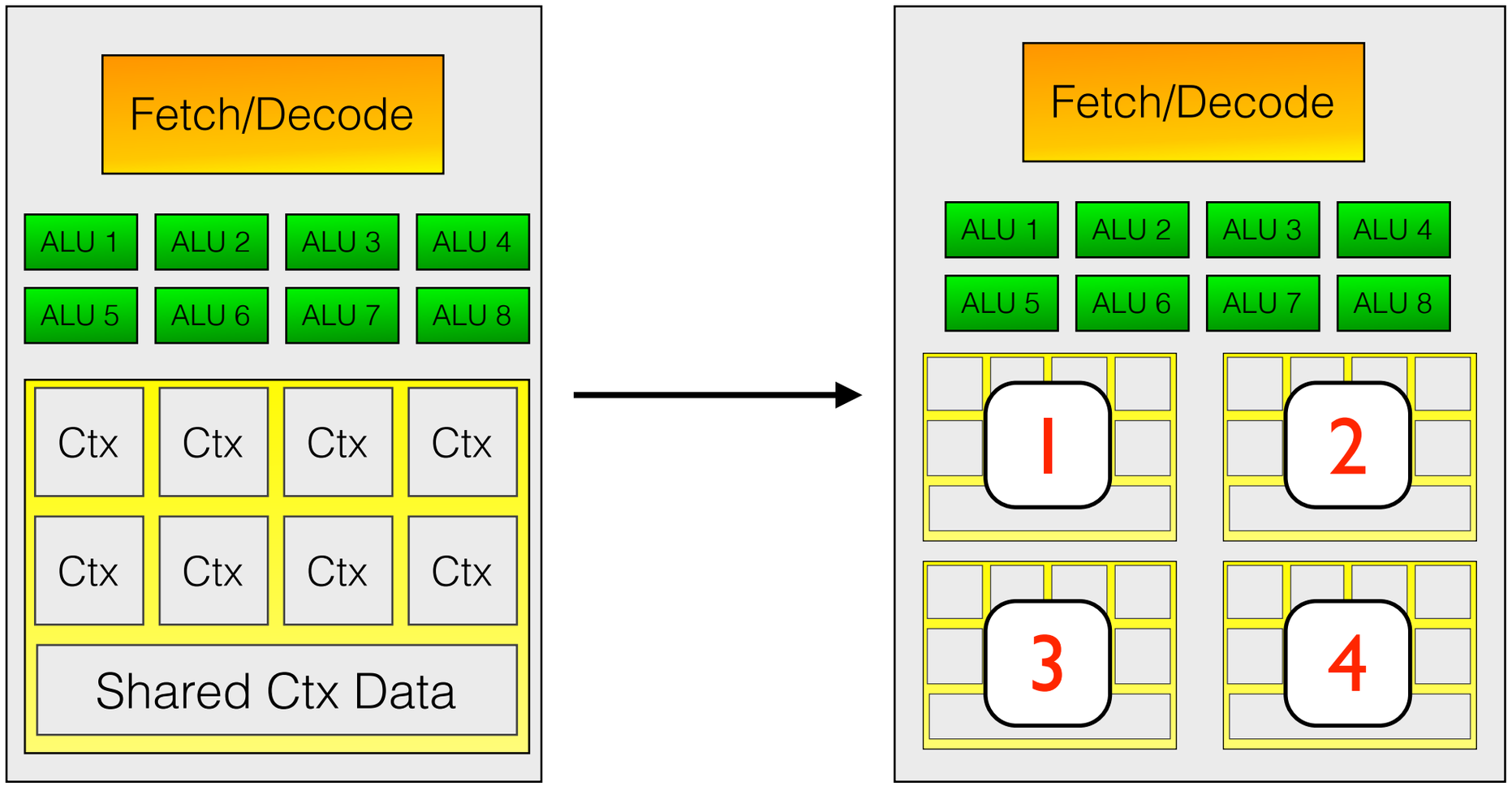
load registers from main memory;

It takes O(1000) cycles to load data from off chip memory into the SM registers file

These ALUs are idled (stalled) after a load

GPU: multiple contexts per core

Fast hardware context switching can help hide high memory latency

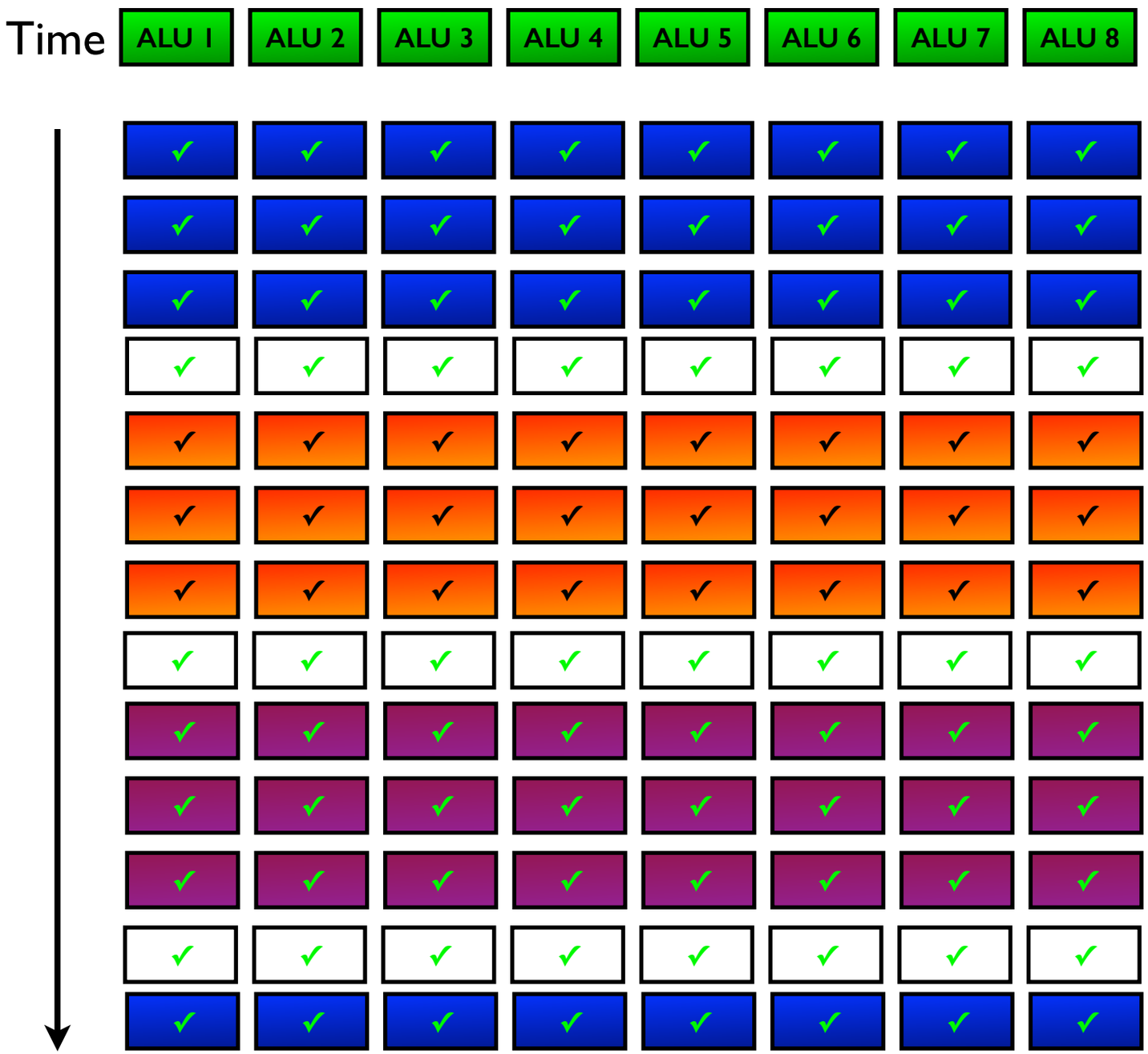


Idea: allocate enough registers and shared memory to allow for multiple contexts

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

GPU: high latency memory fetches

We can hide latency if the core can switch SIMD contexts



Ctx1: work on registers;
Ctx1: work on registers;
Ctx1: work on registers;
Ctx1: load request, switch context;

Ctx3: work on registers;
Ctx3: work on registers;
Ctx3: work on registers;
Ctx3: load request, switch context;

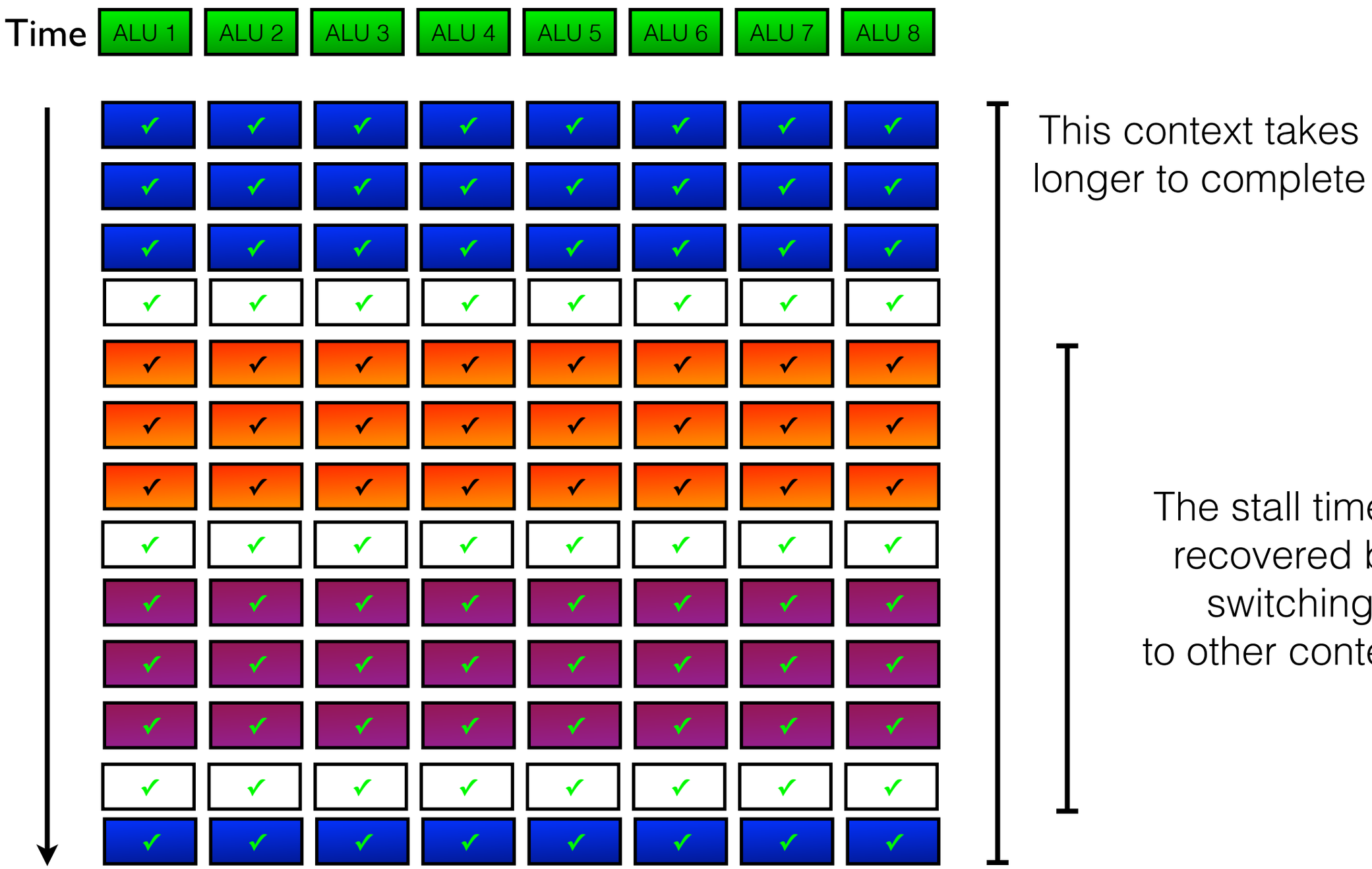
Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: load request, switch context;

Ctx1: load done so continue

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

GPU: slow threads ok

By design it is ok if a threads stall as long as we can switch to other threads



Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

The amount of latency hiding possible depends on how many contexts can be simultaneously resident on the core.

GPU: summary of architecture

Summary of multi-level GPU parallel architecture

- Multiple cores.
- Each core has one (or more) wide SIMD vector units.
  - Each wide SIMD vector unit executes one instruction stream.
- Each core has a pool of shared memory.
- Each core hardware can switch between multiple contexts to hide memory latency.
- Branching code involves partial serialization.

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

Part 2: NVIDIA GPUs & CUDA

NVIDIA's Compute Unified Device Architecture
GPU programming model

Example GPU Card

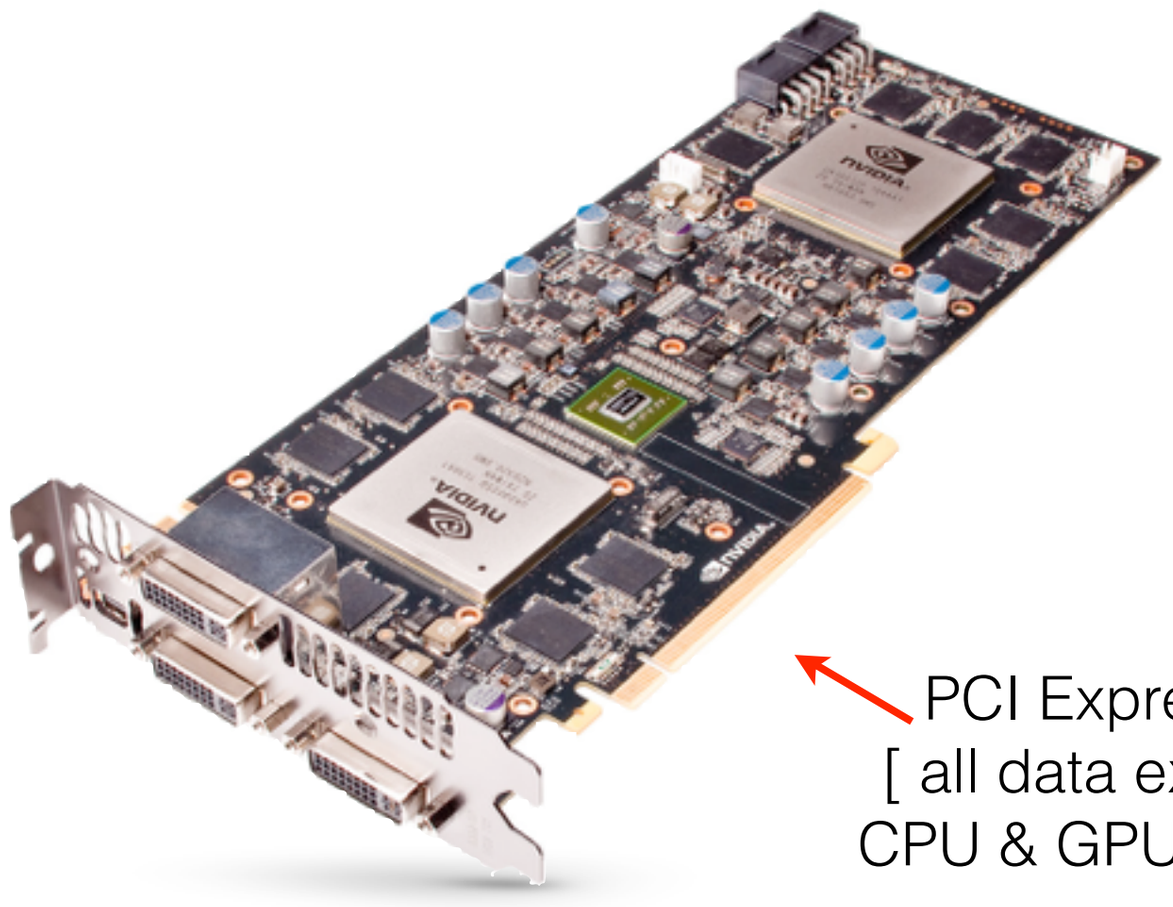
Slightly vintage dual GPU consumer card

GTX 590 GPU

Each discrete GPU is a nearly self-contained daughter card (aka sidecar or accelerator).

GPU: PCI link between host & device

The GPU “device” communicates with the “host” through the PCI slot.



GTX 590 GPU

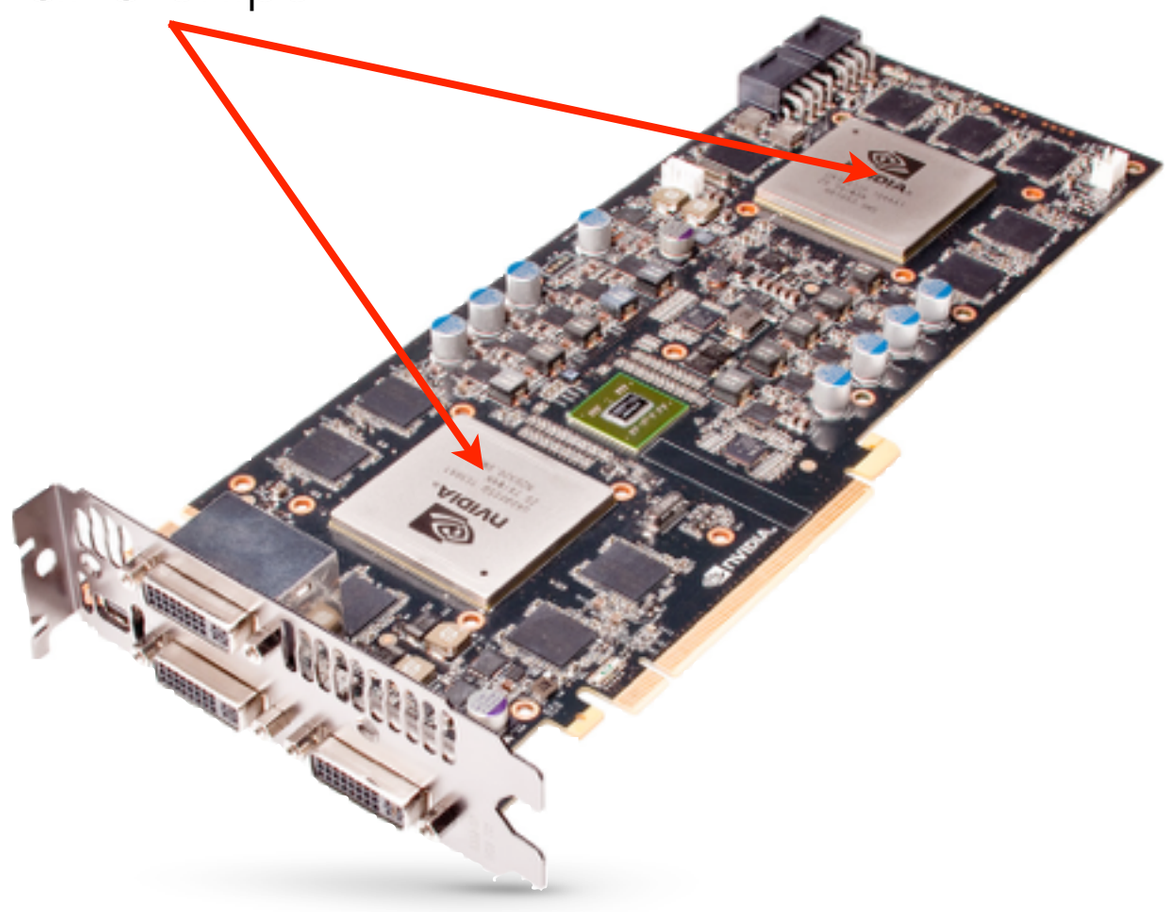
PCI Express 2.0 interface
[all data exchange between
CPU & GPU crosses this bus]

Data has to be off loaded from the HOST to the DEVICE.
Latest GPUs use the PCI Express 3.0 interface standard.

Example GPU Card

Slightly vintage dual GPU consumer card

Two Fermi (GF110)
GPU chips

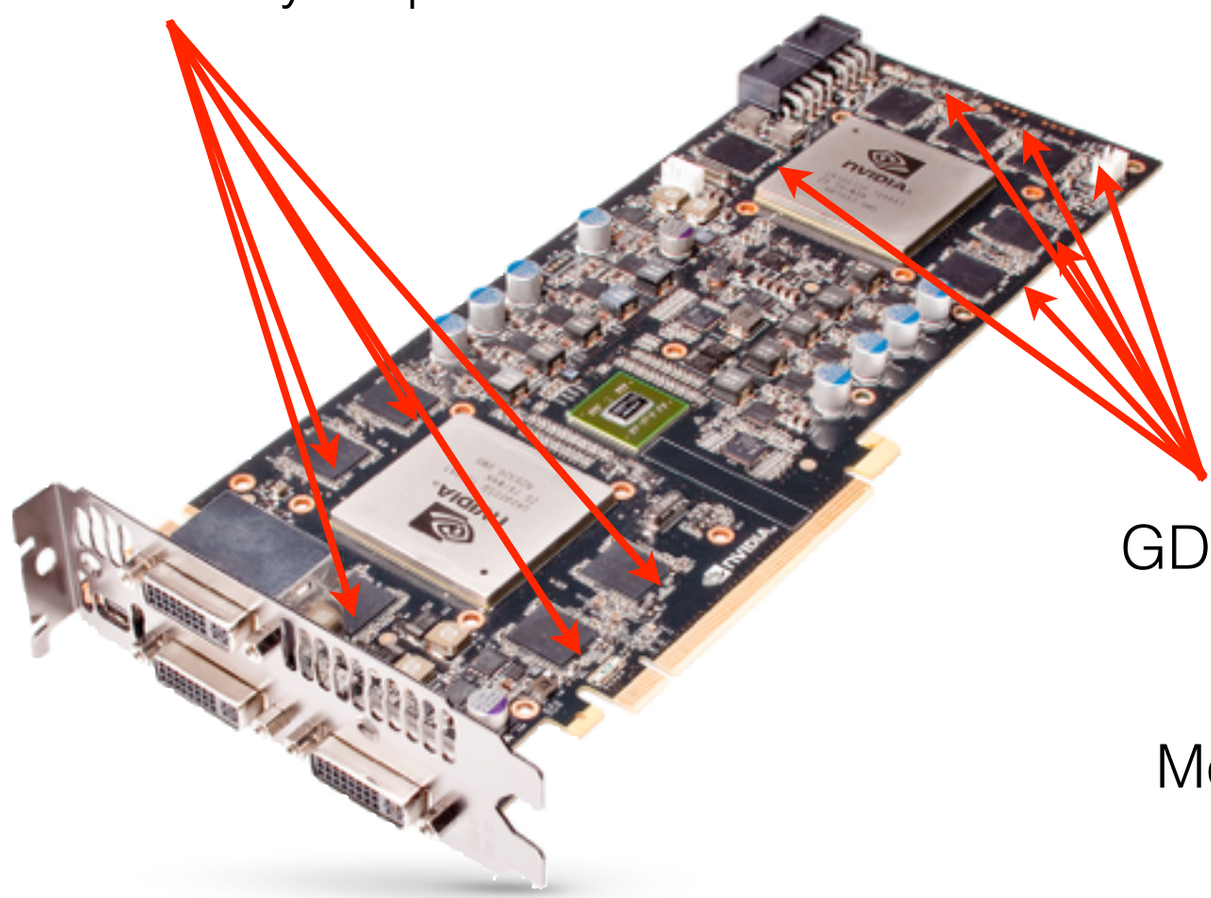


Discrete GPU cards have one or two graphics processing units.
Latest cards from NVIDIA have Kepler GPU chip(s).

GPU: Example Card

Slightly vintage dual GPU consumer card

GDDR5 Memory chips



GDDR5 Memory chips:
1536MB

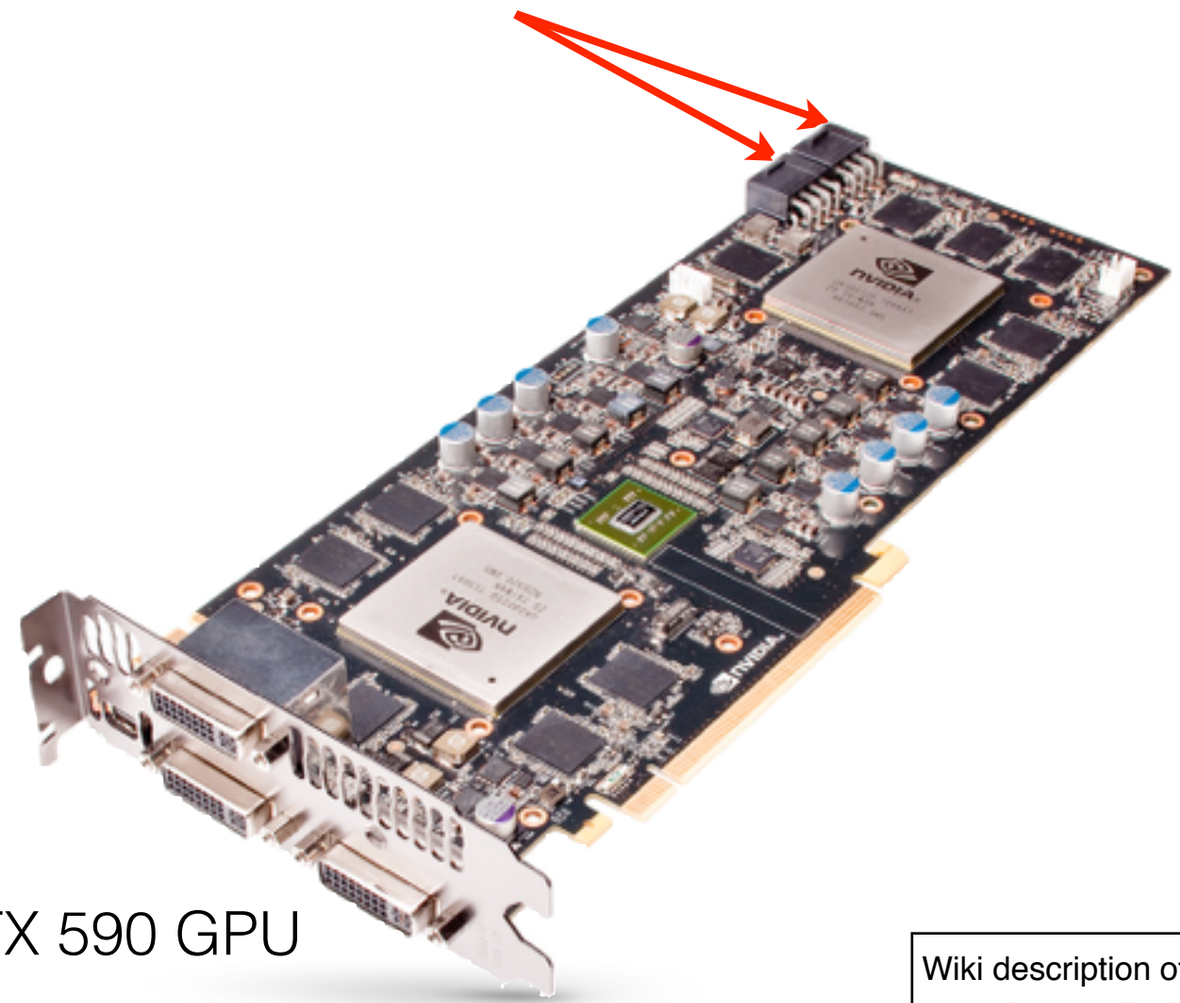
Memory bus: 384 bit

The amount of "device memory" and "device bandwidth" depends on the specific model.

GPU: example card power requirement

The GPU power requirement is typically limited by the PCI Express standard

Power inputs:
TDP* 365W



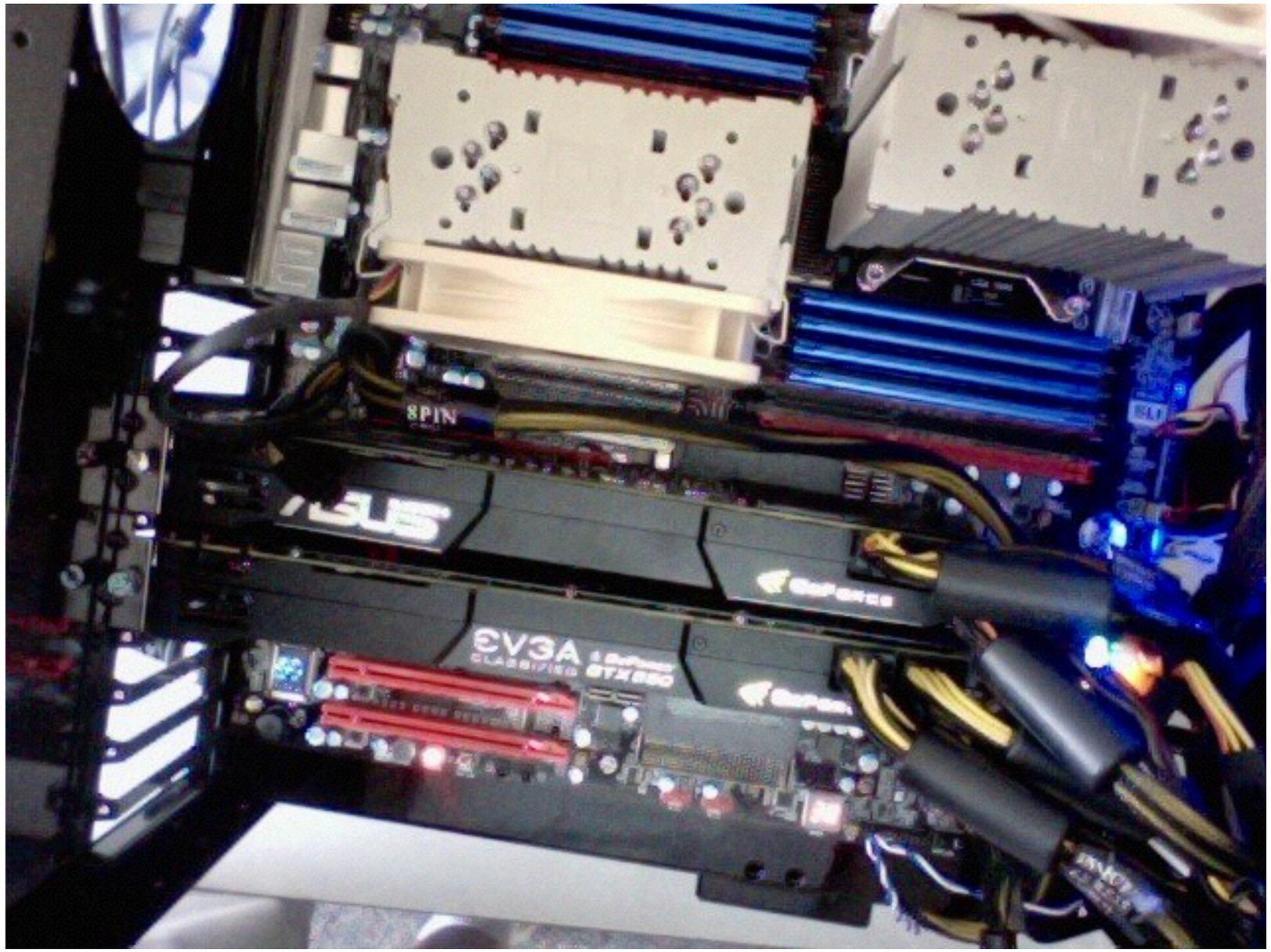
GTX 590 GPU

Wiki description of TDP: “The **thermal design power (TDP)**, sometimes called **thermal design point**, refers to the maximum amount of power the cooling system in a computer is required to dissipate.”

A high end GPU might draw 250W.
See: NVIDIA GPU wiki for detailed specs.

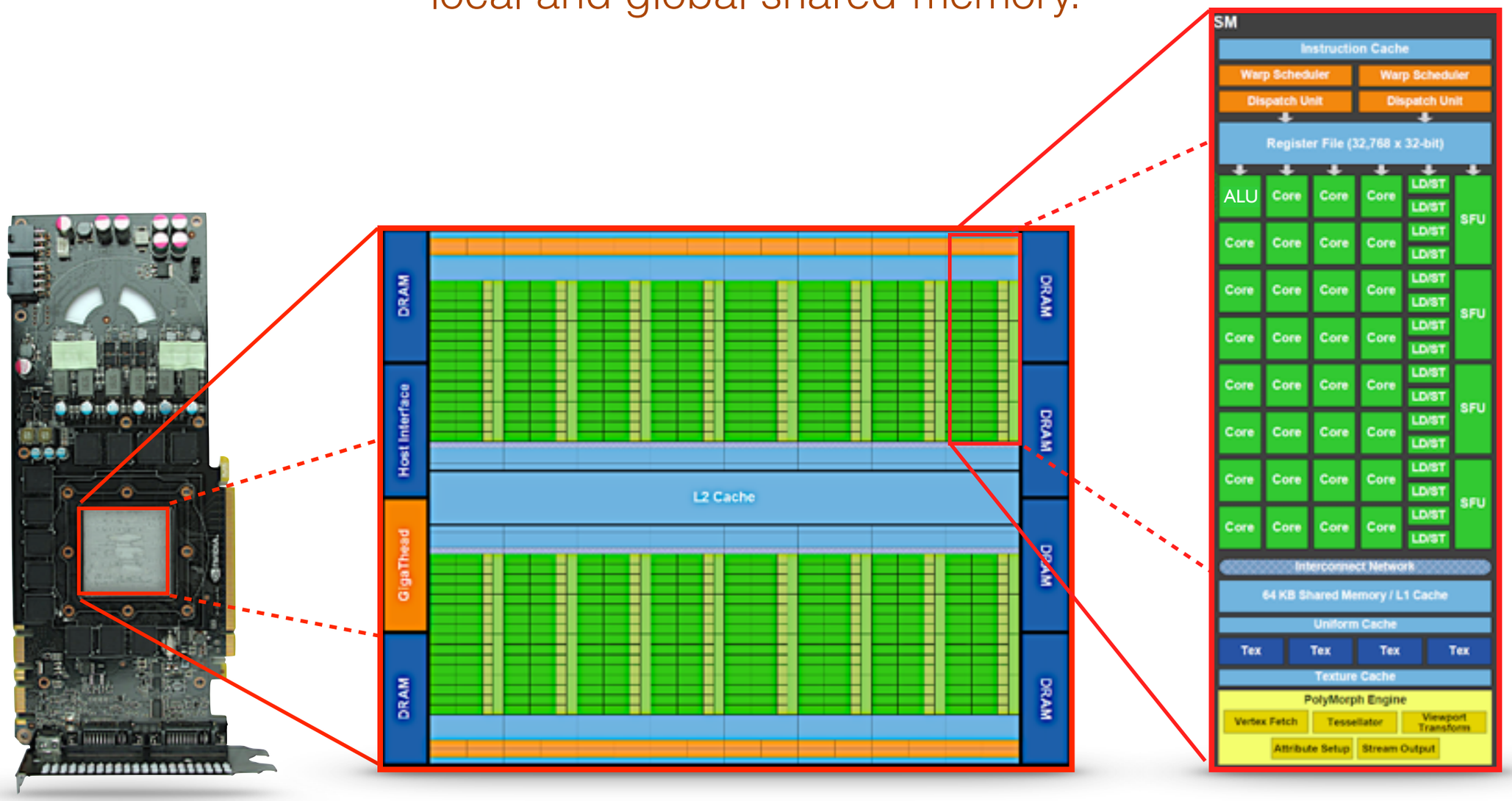
GPU: HOST & DEVICE

Two GPUs (DEVICEs) plugged into HOST motherboard.



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

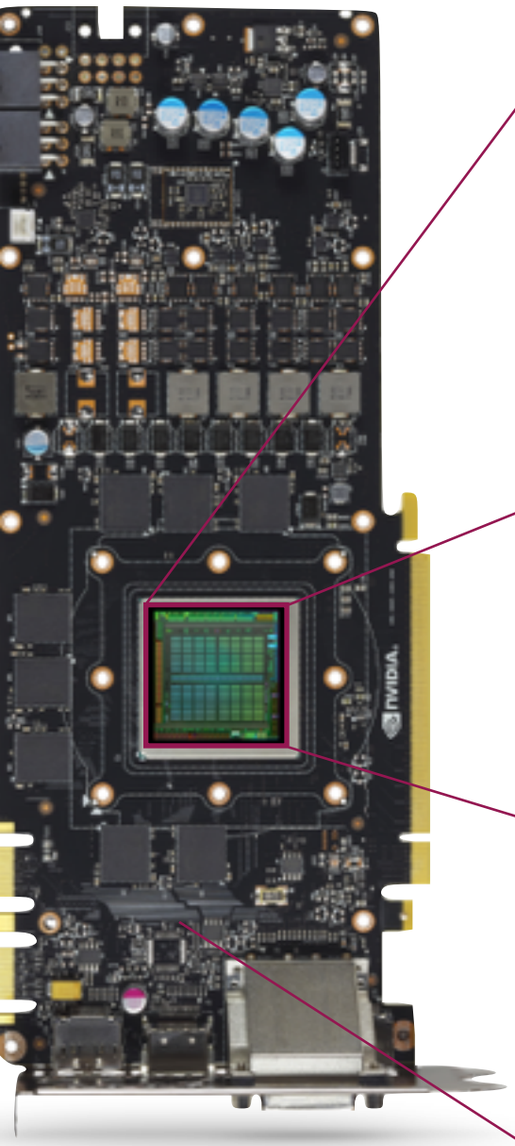
GK110: 15 cores that cluster 192 FPU each.

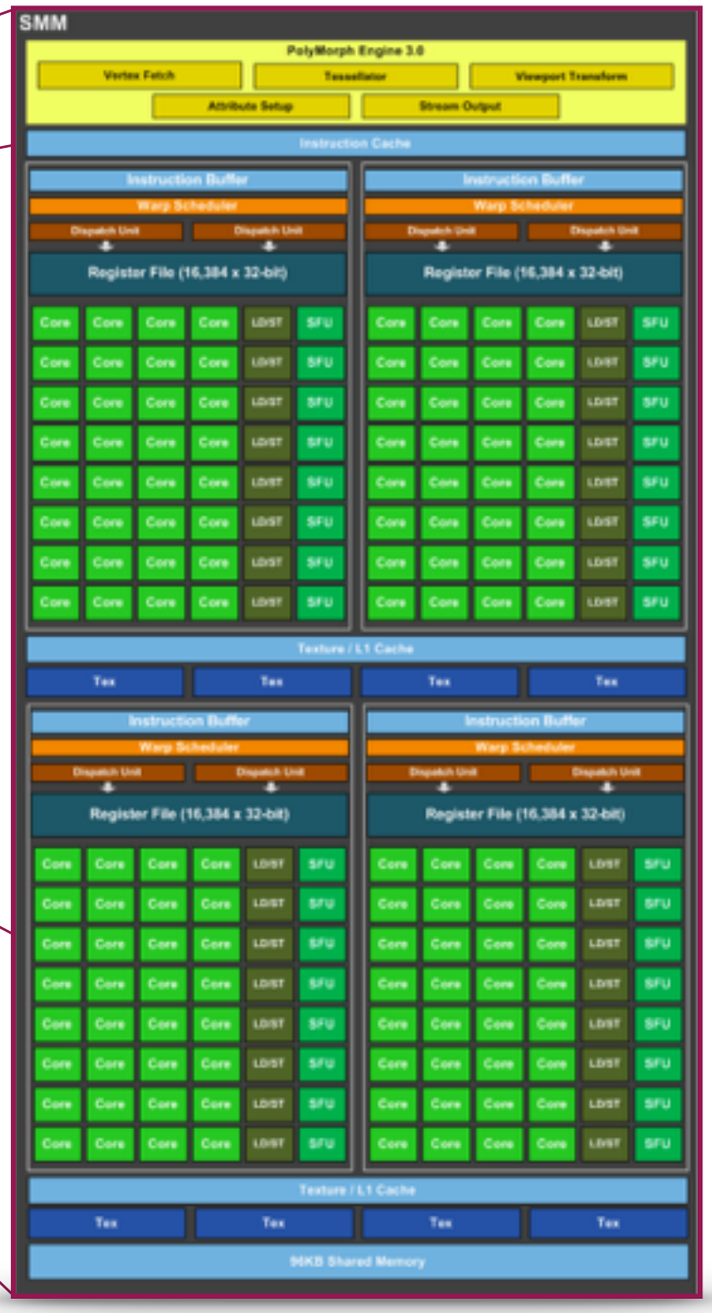
Each Kepler core (SMX) has six SIMD clusters of 32 ALUs
Data streams at ~70 GFLOAT/s and peak 4+ TFLOP/s (SP)

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



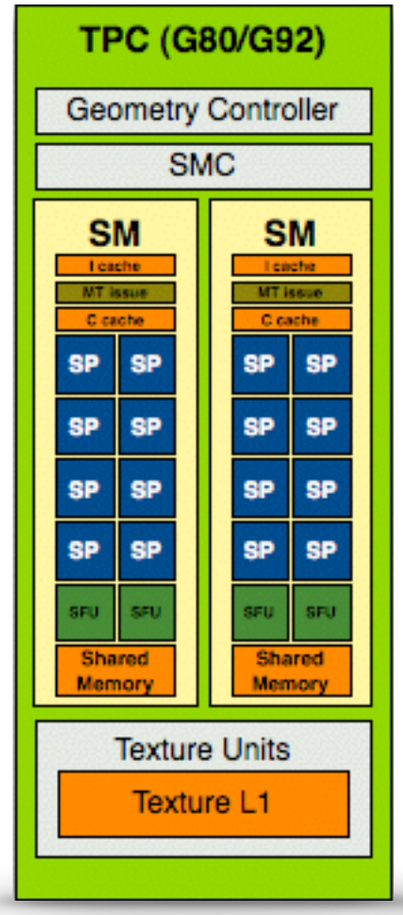


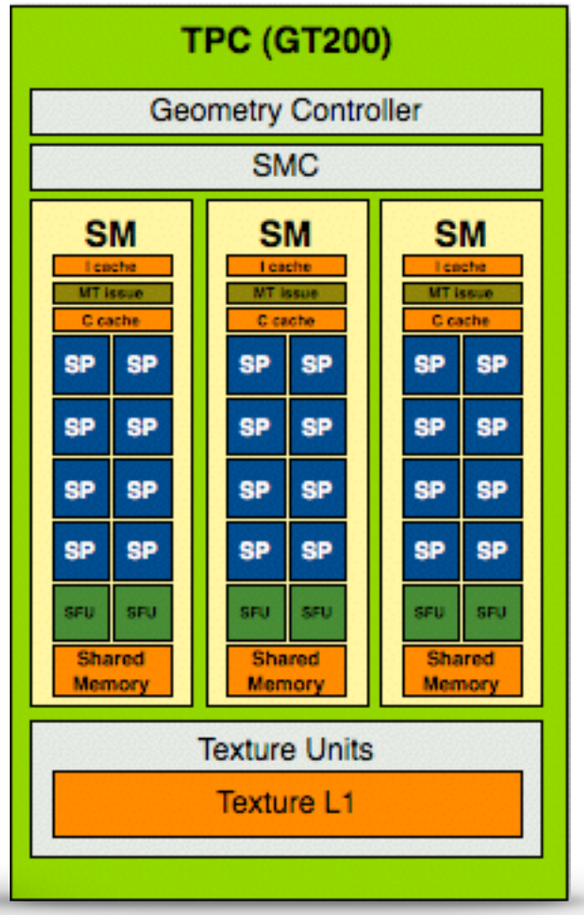
16 Maxwell cores each have four SIMD clusters with 32 ALUs.
Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)

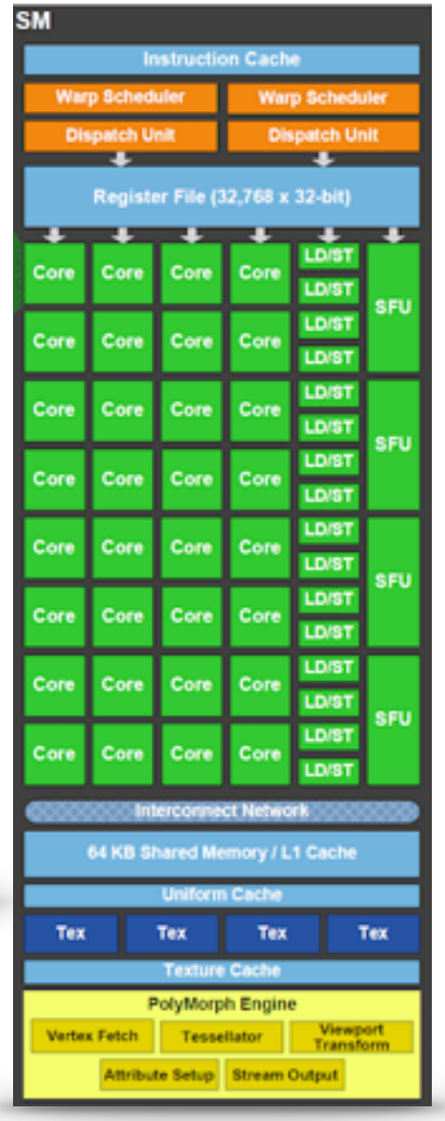
GPU: trends in FPU Clusters

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

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







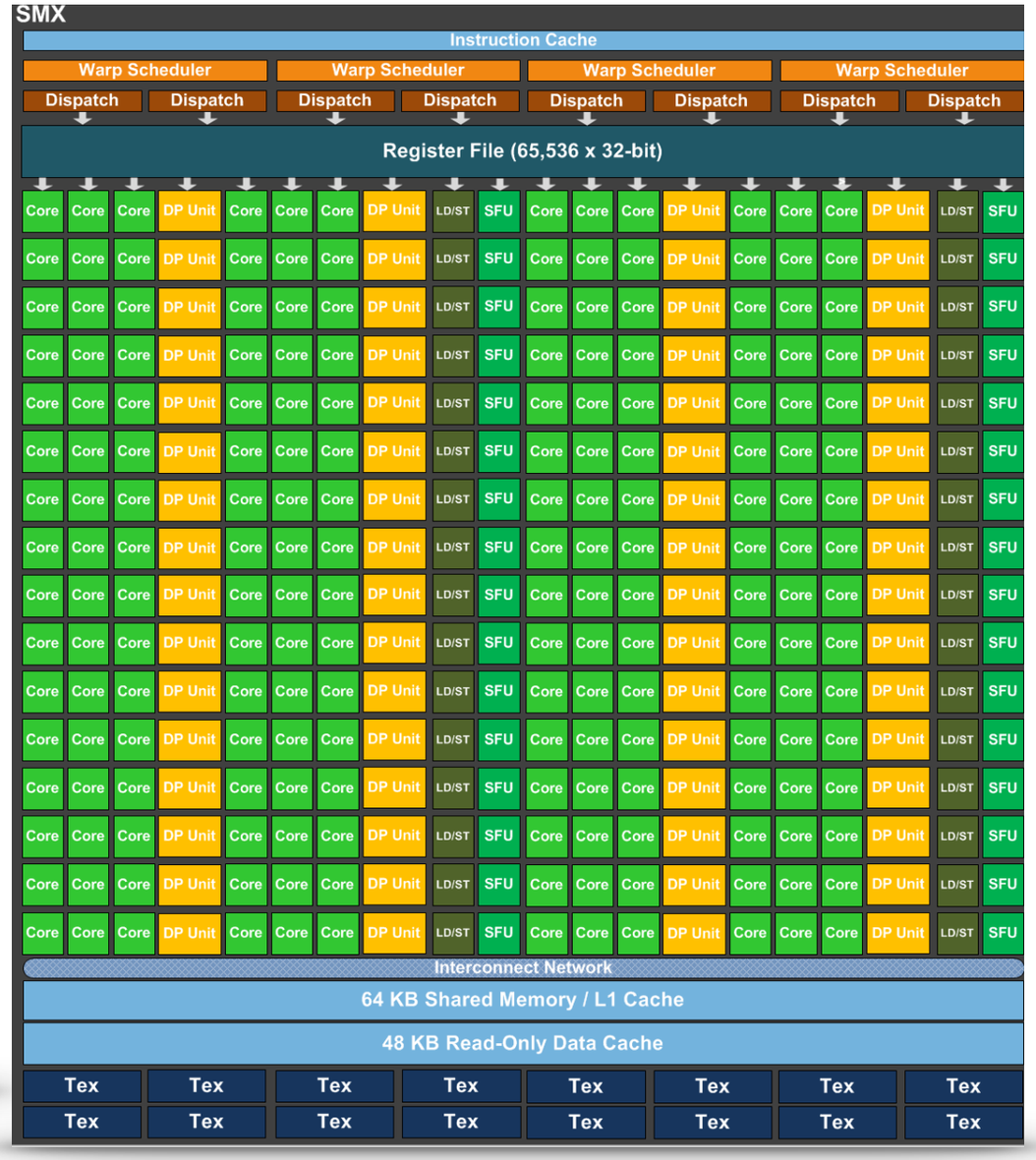


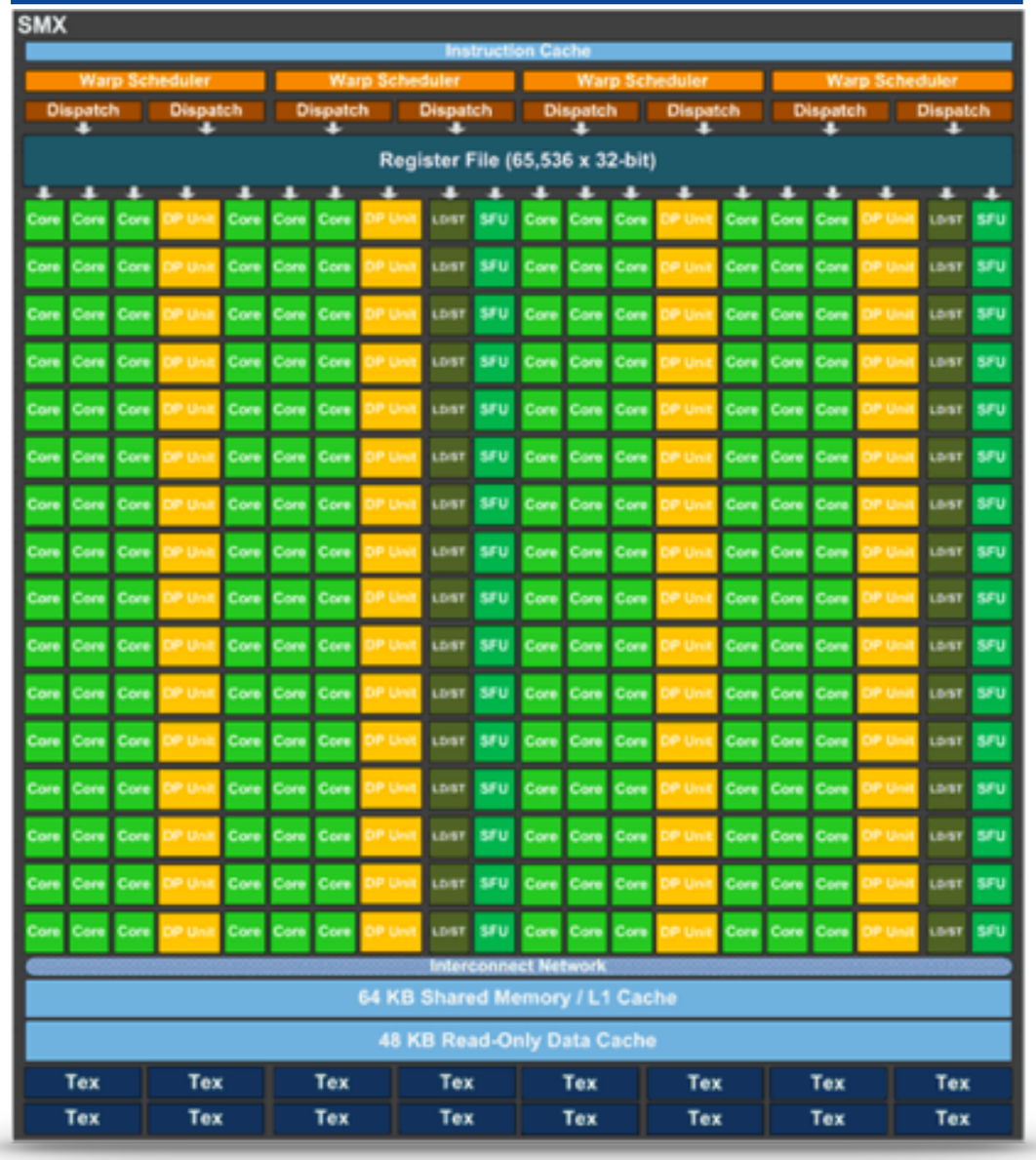
Image sources: http://forum.beyond3d.com/showthread.php?t=58668&page=140, http://www.anandtech.com/show/2549/2 , http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

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

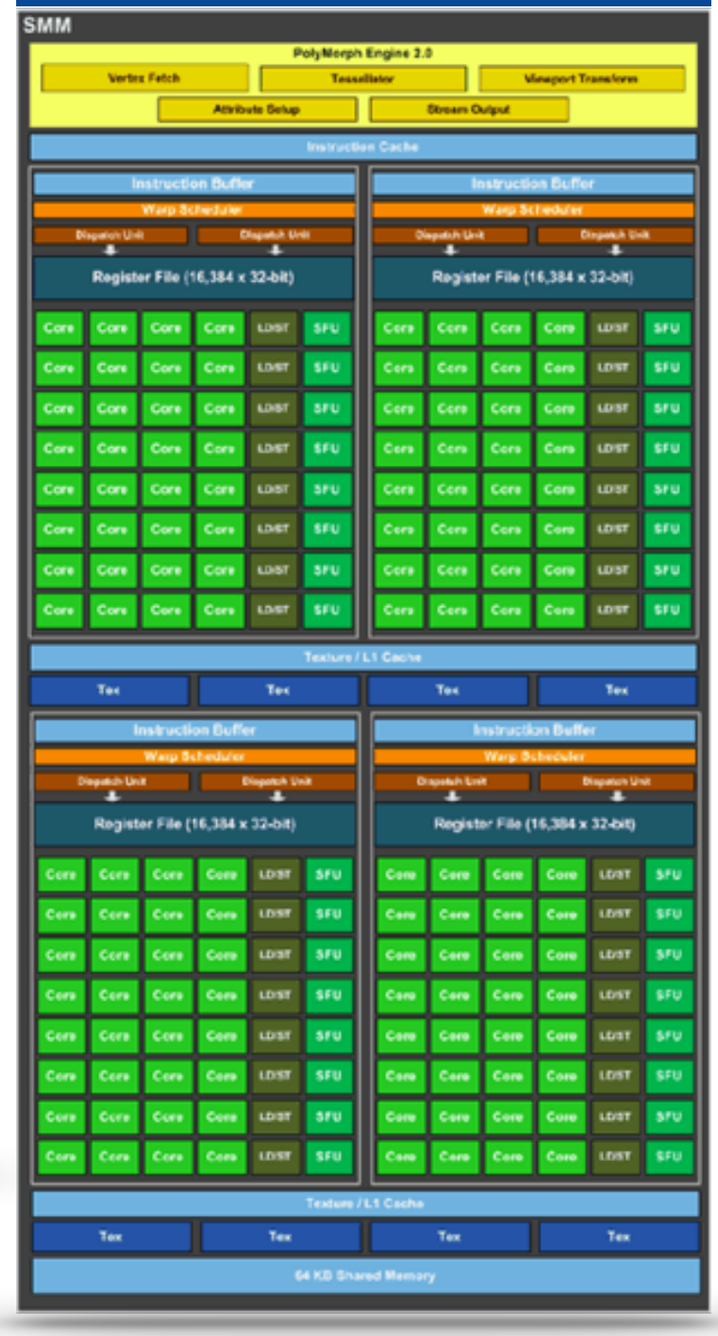
The FPU clusters ("core") in 2 latest NVIDIA processor architectures

Kepler SMX



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

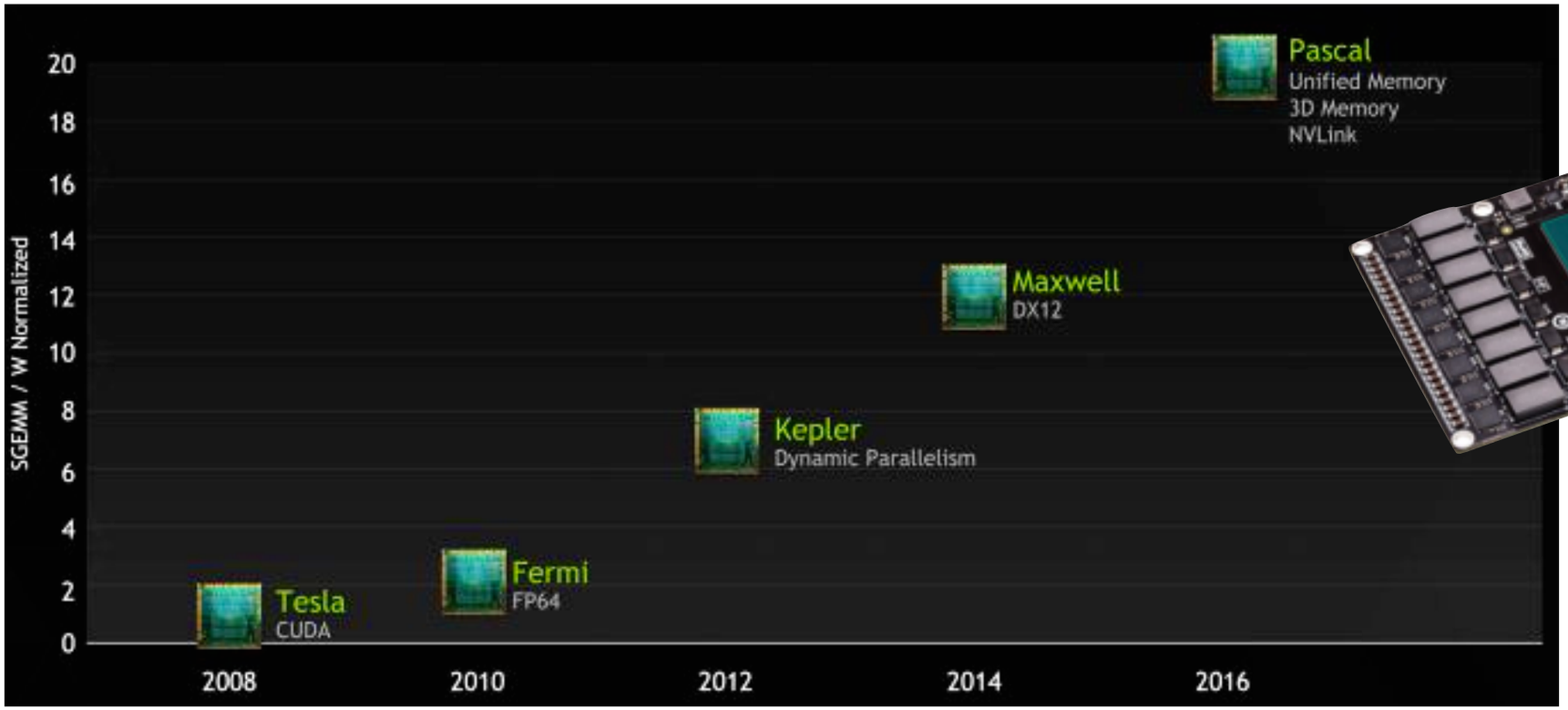
Maxwell SMM

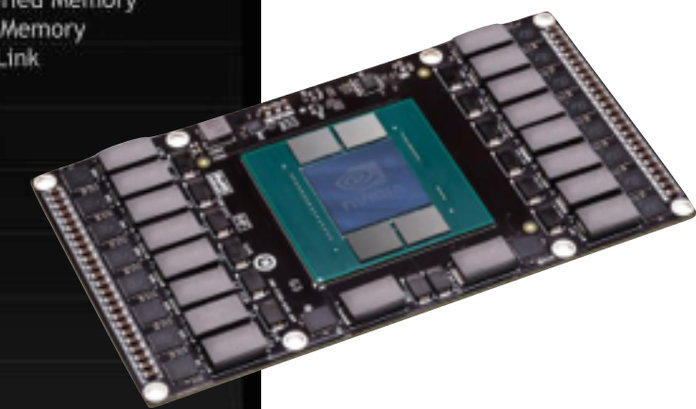


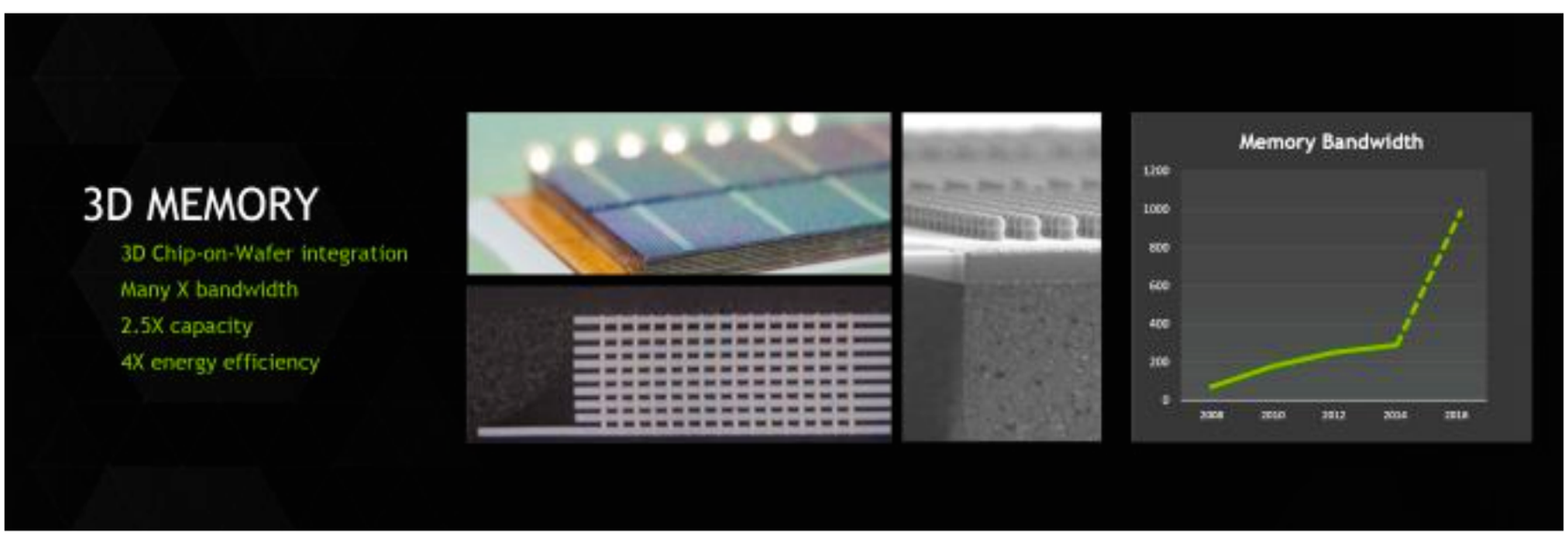
Maxwell notes: Partitioned register files, multiple instruction buffers, partitioned core, same shared memory, multiple texture/L1 caches.

GPU: NVIDIA roadmap

NVIDIA's public roadmap for continued scaling.

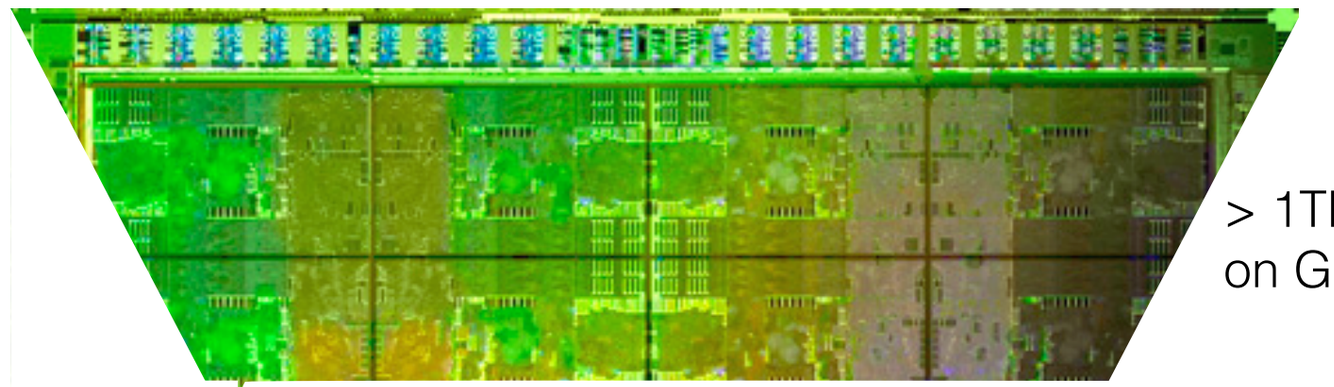




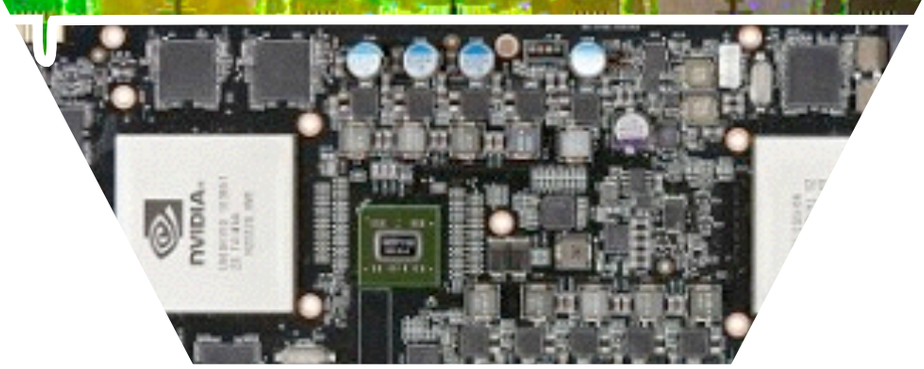


Major bump in performance Maxwell - Pascal transition is contingent on stacked memory.
[http://www.anandtech.com/show/7900/nvidia-updates-gpu-roadmap-unveils-pascal-architecture-for-2016]

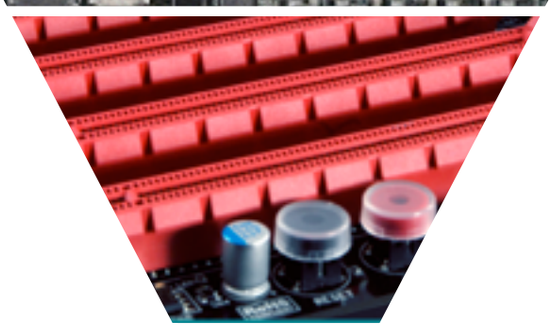
GPU: Data Transfer NUMA Bottlenecks



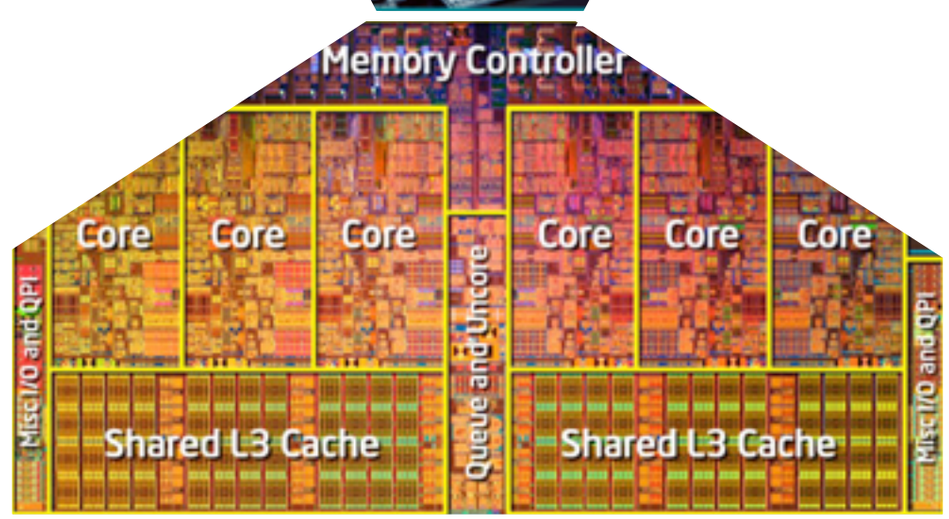
> 1TB/s
on GPU



up to 330GB/s
DEVICE <BUS> DEVICE DRAM



~16GB/s
DEVICE <PCI> HOST



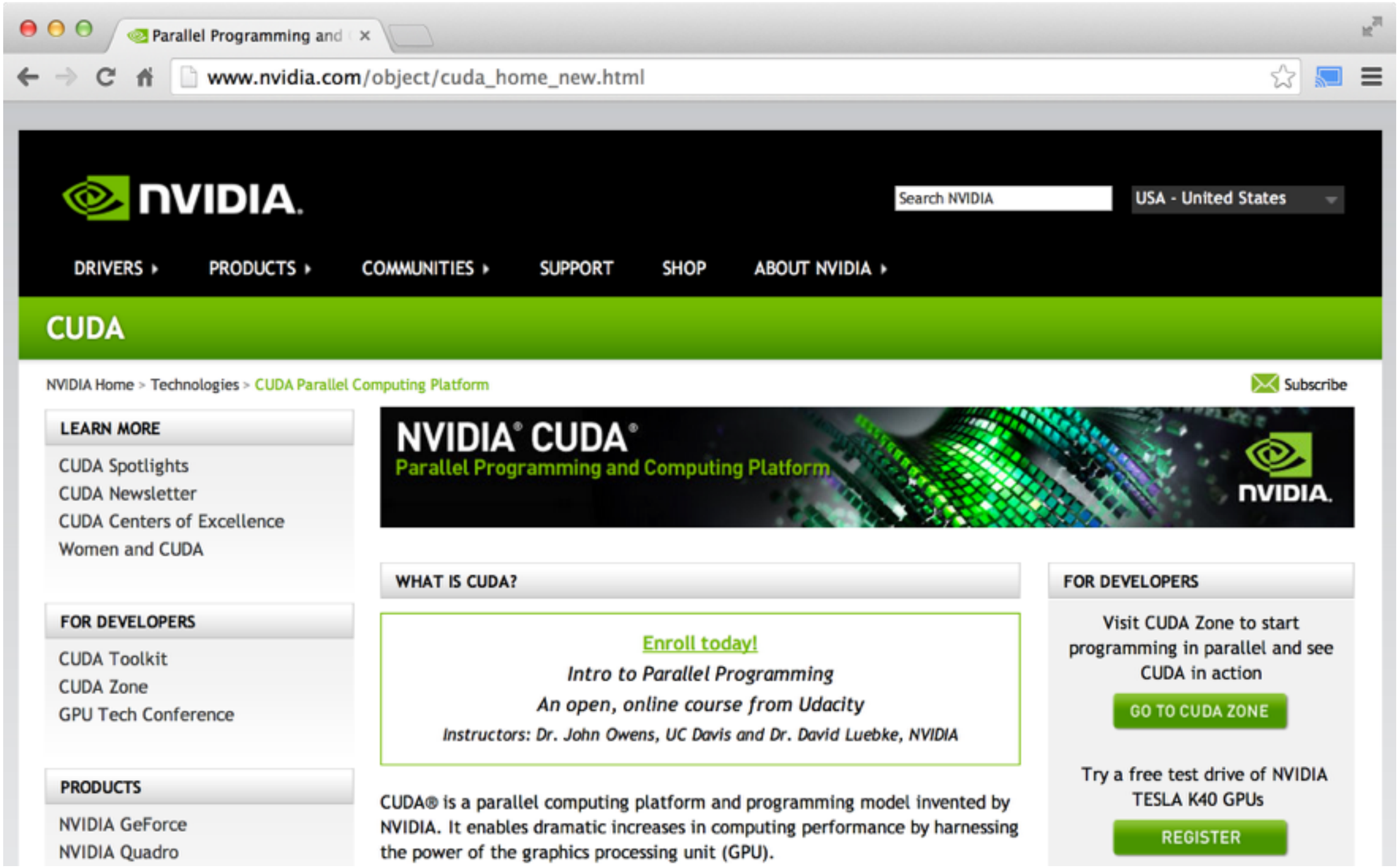
~25.6GB/s
CPU <QPI/HT> CPU RAM

Images credits: http://alienbabeltech.com/main/gtx_590_review
http://www.evga.com/articles/00537/

There is a major bottleneck between the GPU and CPU.
Data localization is critical for high-performance.

CUDA: compute unified device archicture

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

In CUDA the programmer
explicitly moves data between HOST and DEVICE

Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.

CUDA: offload model

In CUDA the programmer
explicitly moves data between HOST and DEVICE

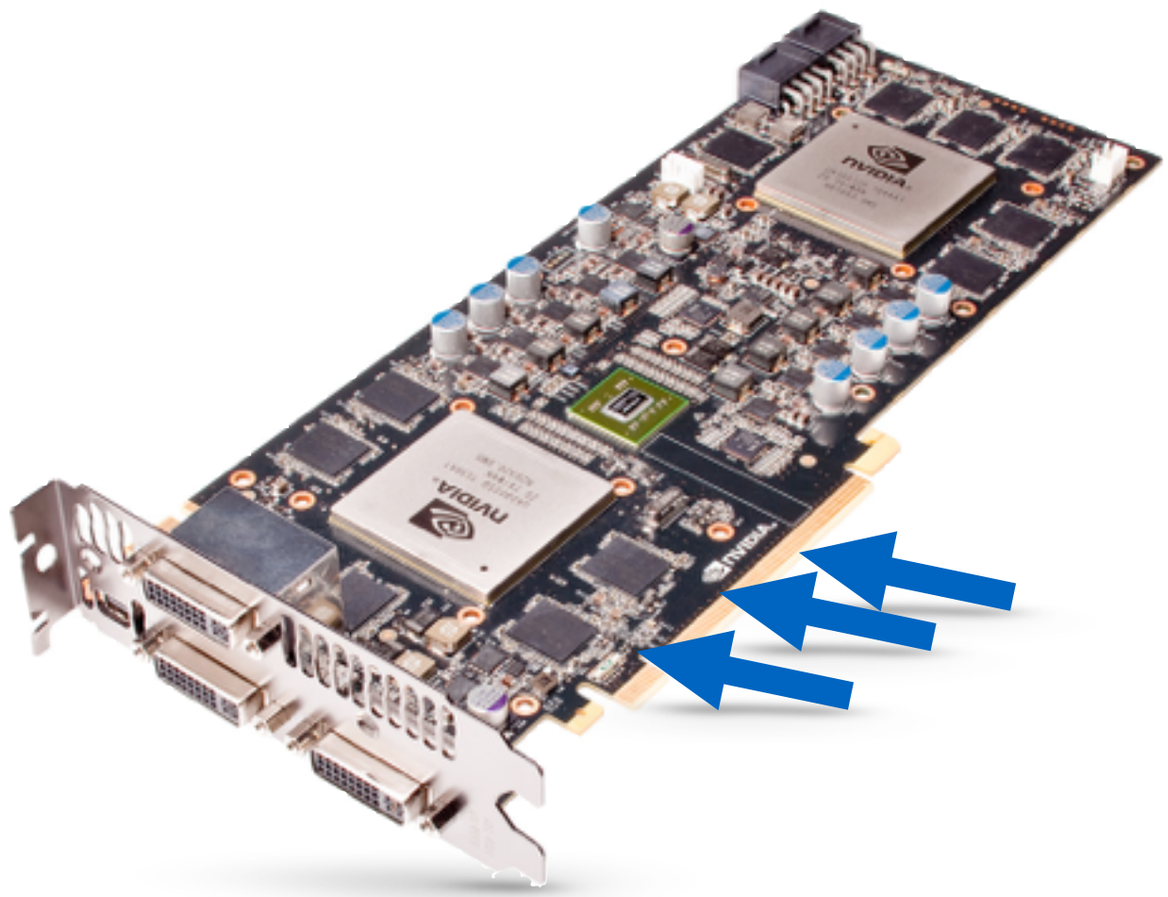
1. cudaMalloc: allocate memory for a DEVICE array

Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.

CUDA: offload model

In CUDA 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

Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.

CUDA: offload model

In CUDA 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

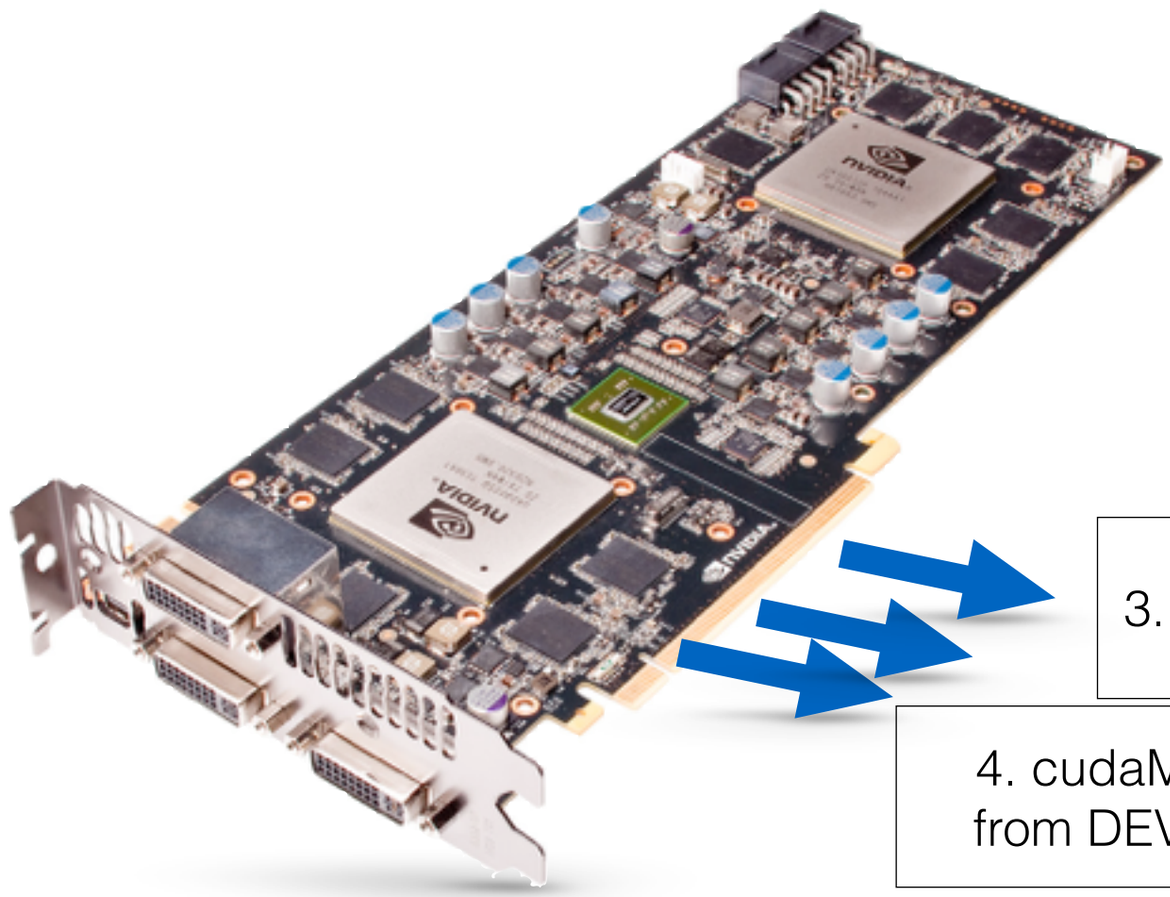
3. Queue kernel task on DEVICE

Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.

CUDA: offload model

In CUDA 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

3. Queue kernel task on DEVICE

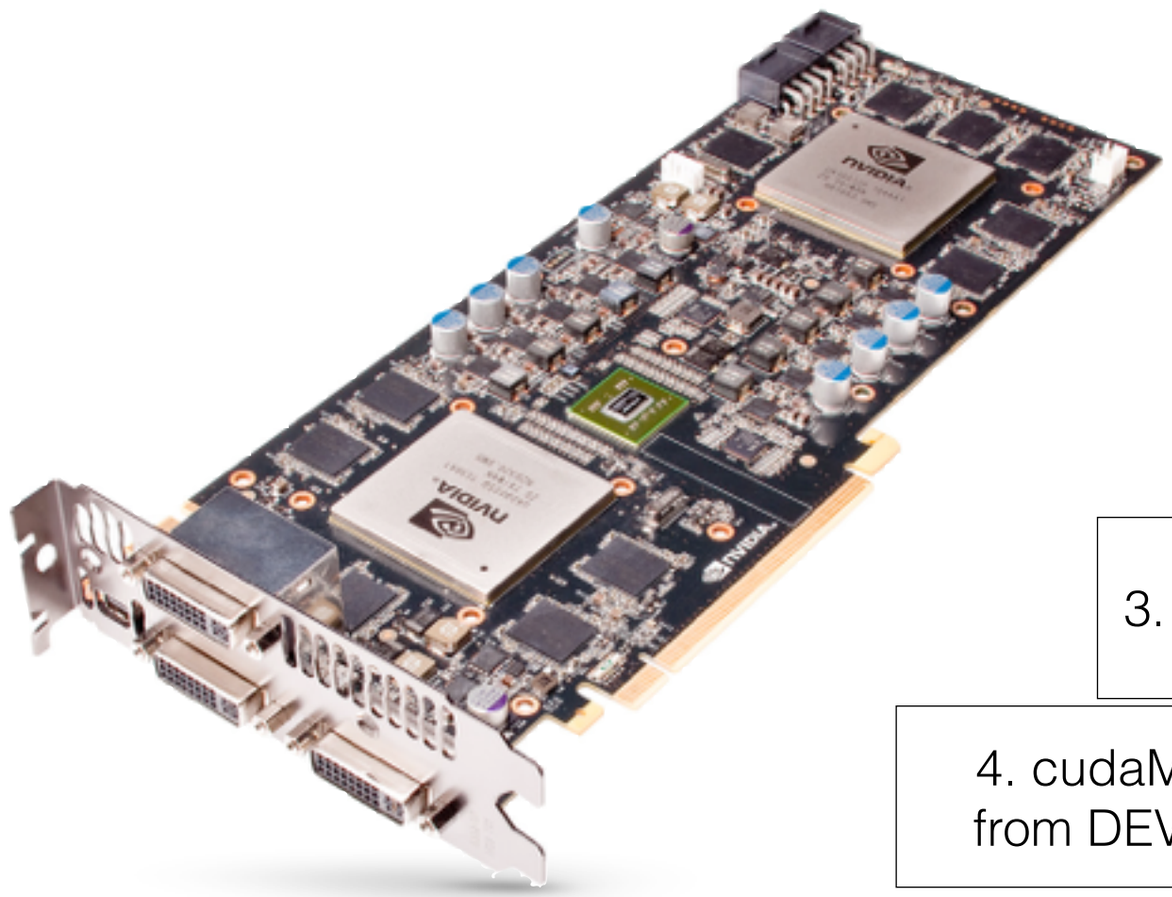
4. cudaMemcpy: copy data from DEVICE to HOST array

Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.

CUDA: offload model

In CUDA 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

3. Queue kernel task on DEVICE

4. cudaMemcpy: copy data from DEVICE to HOST array

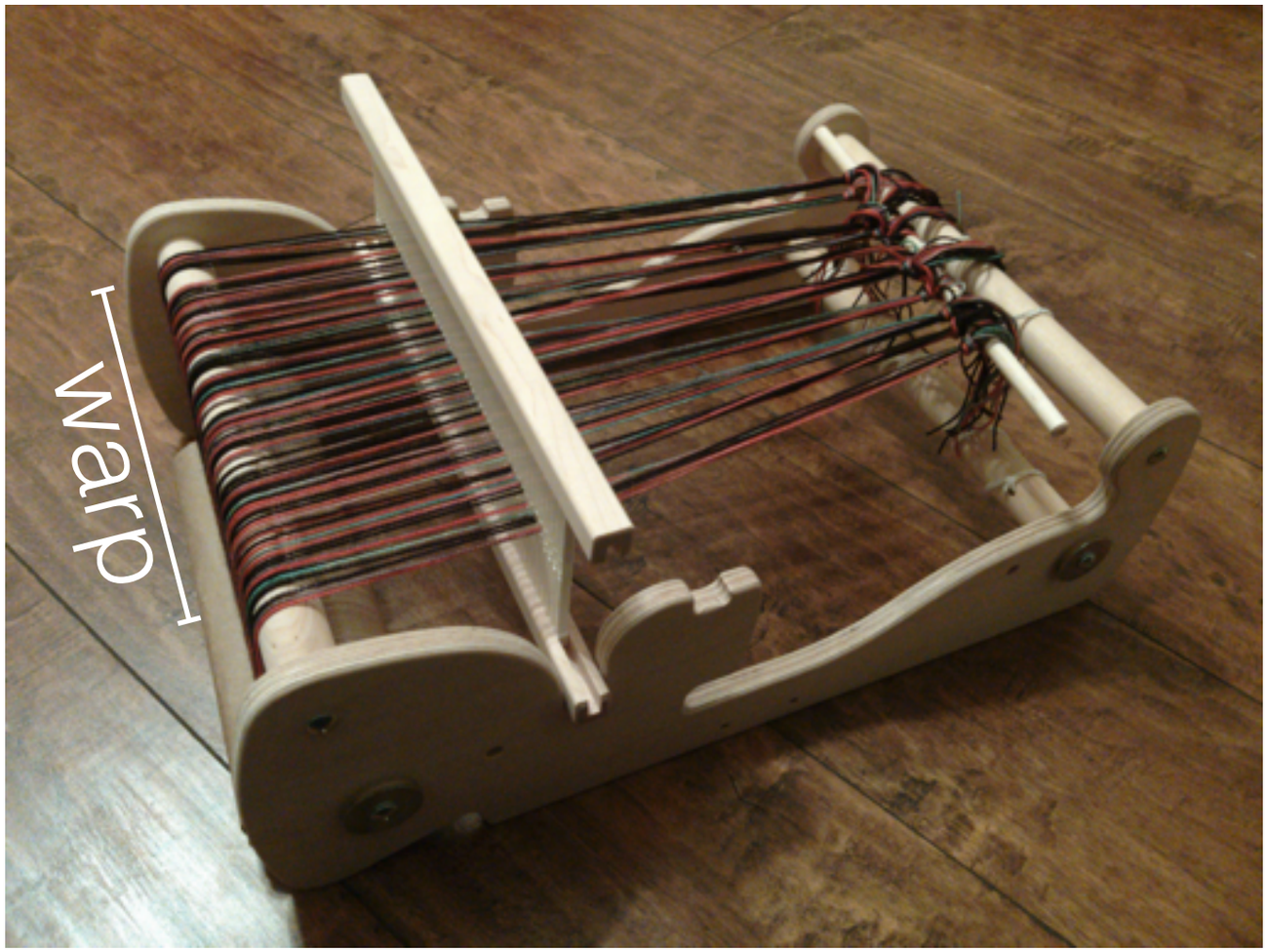
Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.

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…

GPU: natural thread model

The GPU architecture admits a natural parallel threading model

- Programmer decides how to partition a compute task 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 (ALU)
      - 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 (blocking context 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: HOST code

Overview of C-like 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 = 17;
    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

In preparation for writing a CUDA kernel we consider
first a serial function that fills an array with entries 0:N-1

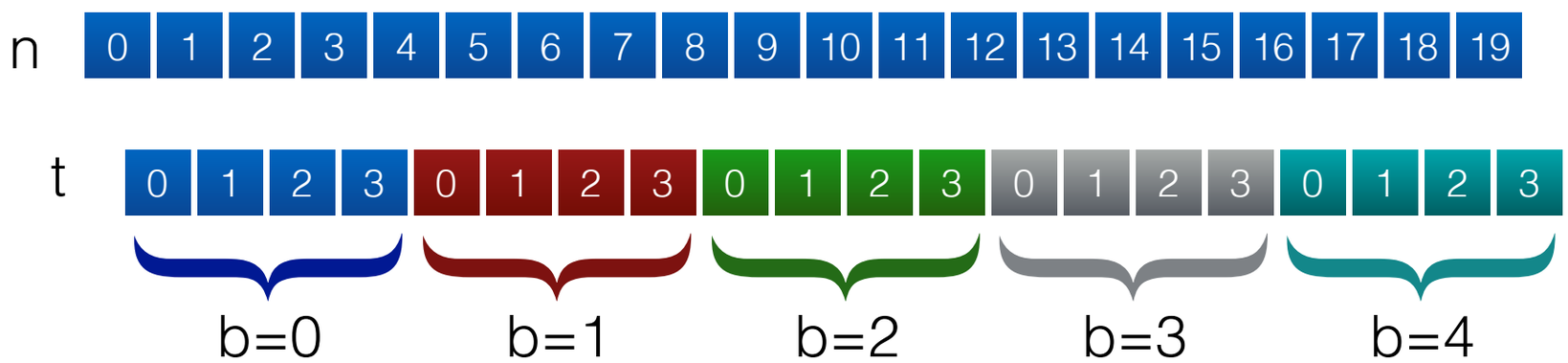
```
void serialSimpleKernel(int N, float *d_a){

  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;

  }
}
```

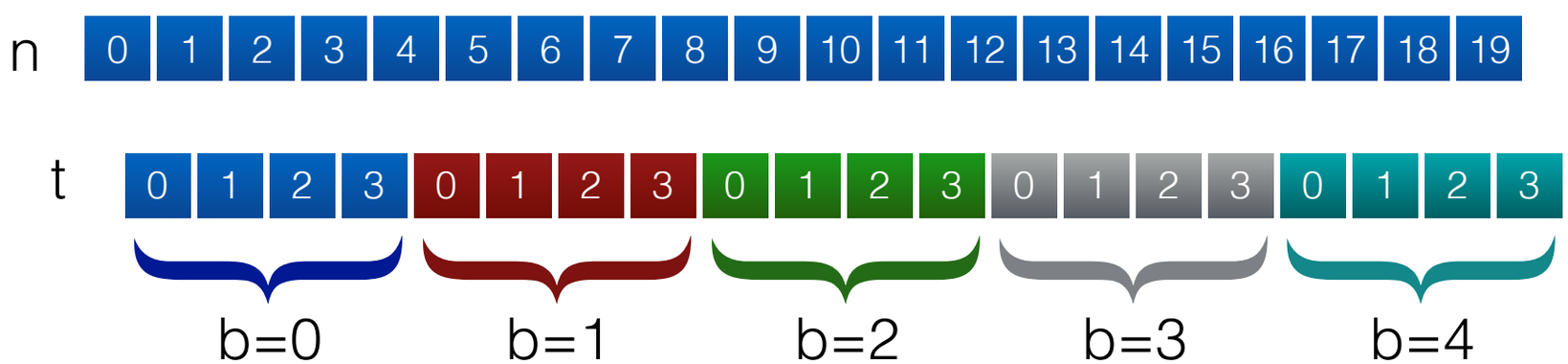


We can think of splitting the n-loop into tiles of size 4: n=t+4b.
Here: block dimension = 4 and grid dimension = 5.

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 kernel code

```
// 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 kernel code

```
// 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 kernel code

```
// 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 kernel code

```
// 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: code samples @ github

GitHub, Inc.

This repository Search

Pull requests Issues Gist

tcew / ATPESC15

Unwatch 1 Star 0 Fork 0

Description

Short description of this repository

Website

Website for this repository (optional)

Save or Cancel

5 commits 1 branch 0 releases 1 contributor

Branch: master ATPESC15 / +

tidy

Tim Warburton authored 6 hours ago latest commit a0b74b5da4

examples	tidy	6 hours ago
LICENSE	Initial commit	a day ago
README.md	Update README.md	a day ago

README.md

Code

Issues 0

Pull requests 0

Wiki

Pulse

Graphs

Settings

HTTPS clone URL

https://github.com/tcew

Make sure you can complete this exercise now !

github repo: https://github.com/tcew/ATPESC15
See: examples/cuda/simple

CUDA: compiling & running example

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

```
# find the source
cd ATPESC15/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/ATPESC15/examples/cuda/simple

CUDA: elliptic solver example

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

Elliptic Poisson problem:

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

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

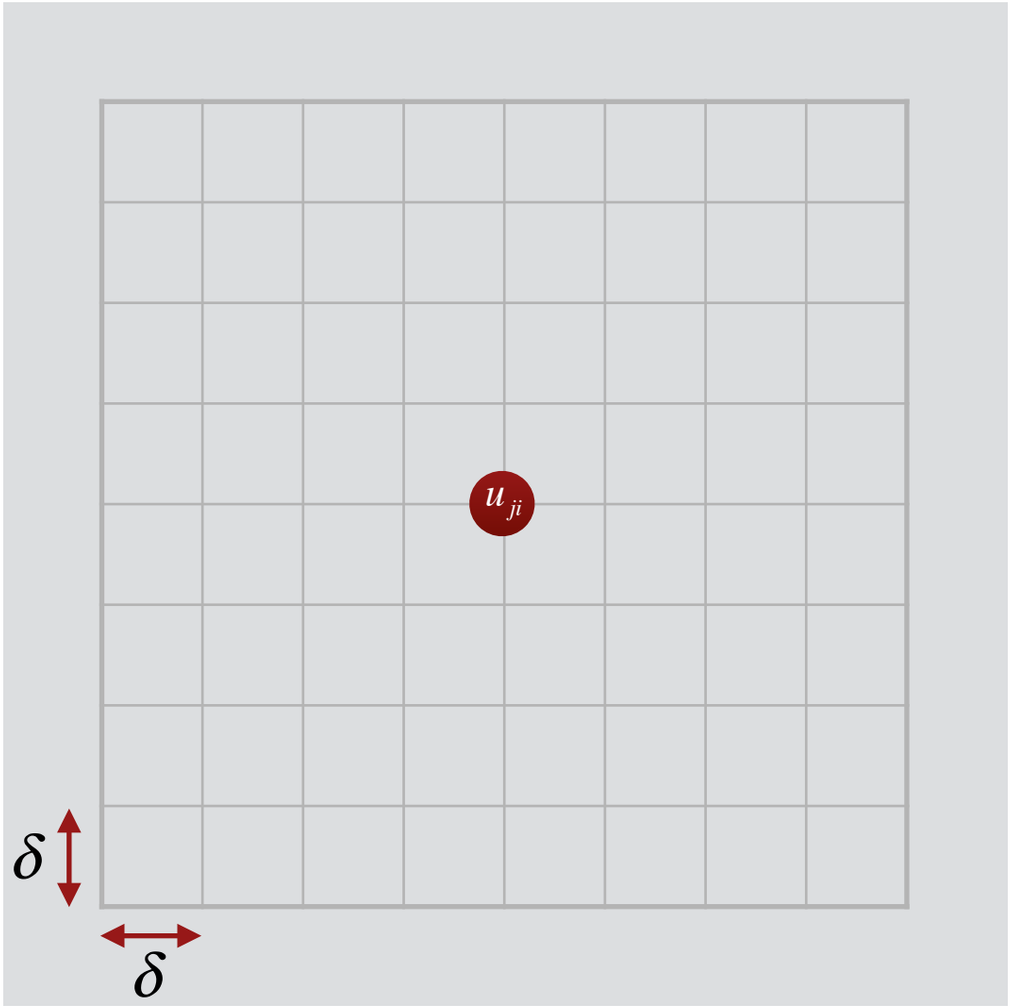
Poisson problem is an archetypal building block for many physics packages.

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.

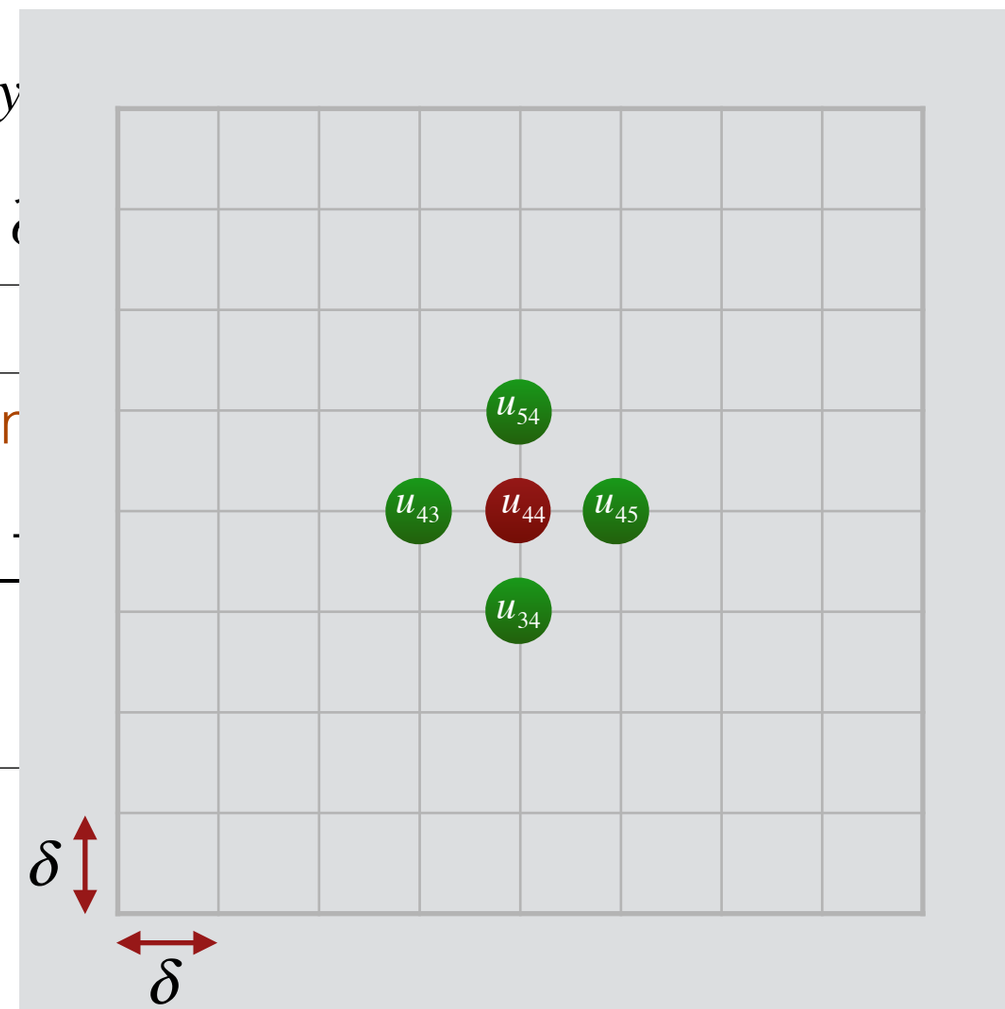
Elliptic Poisson problem:

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

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

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/ATPESC15/examples/cuda/simple

CUDA: parallelism for solver example

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

Iterate:

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

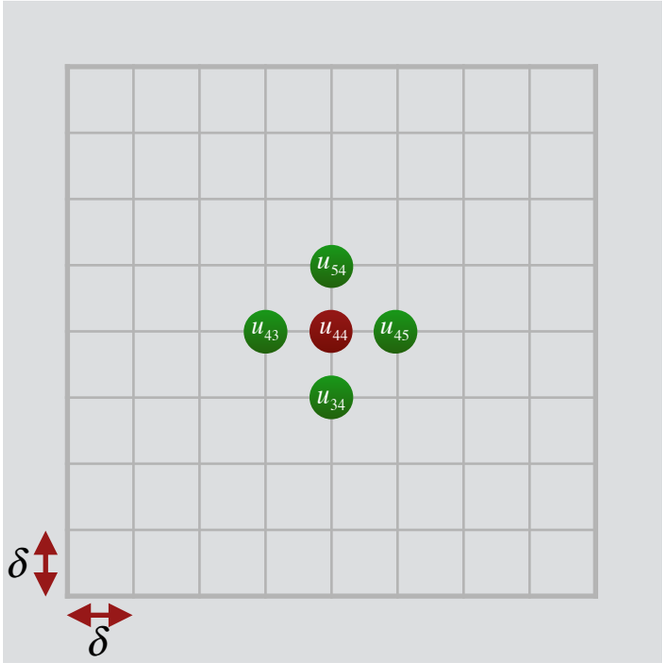
The GPU works best when every thread is doing the same thing.

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:

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]);
    }
  }
}
```

Iterate:

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

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*

CUDA: parallel reduction

Second step: reduce solution array to a single scalar.

Check if:

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

Simplify to reduction of a linear vector:

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

Ah - this looks like a very serial sum.

CUDA: parallelism for solver example

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

Reduction:

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

Block reduction (B blocks)

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

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

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 box;
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 thread-block.
All threads in the thread-block have to enter the __syncthreads() before any of them can return.

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;
    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 thread-block.
All threads in the thread-block have to enter the __syncthreads() before any of them can return.

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;
    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 thread-block.
All threads in the thread-block have to enter the __syncthreads() before any of them can return.

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;
    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 thread-block.
All threads in the thread-block have to enter the __syncthreads() before any of them can return.

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

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

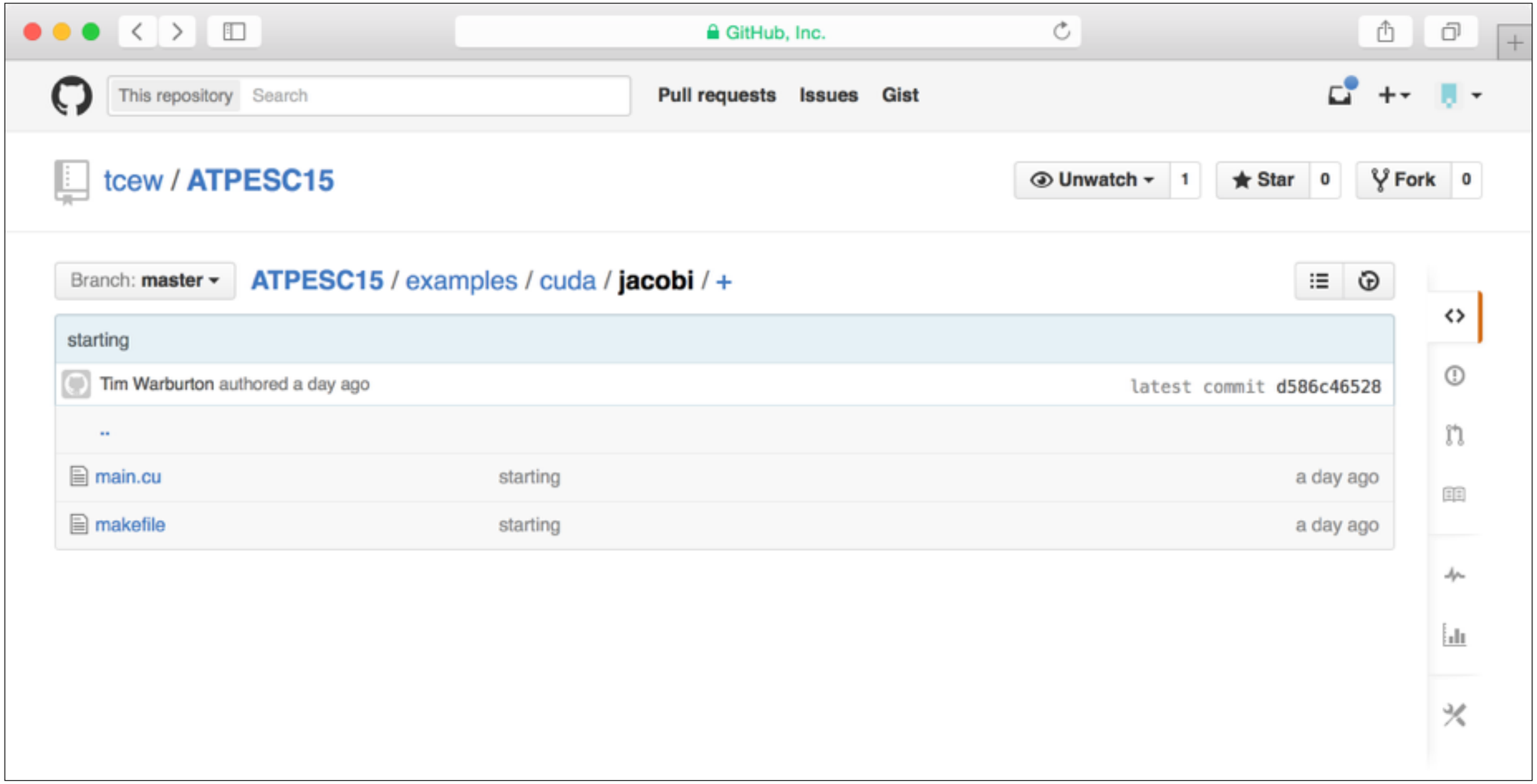
Thread \ Step	0	1	2	3	4	5	6
0	1	1+7	8	8+11	19	19+17	36
1	3	3+8	11	11+6	17		
2	5	5+6	11				
3	2	2+4	6				
4	7						
5	8						
6	6						
7	4						

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

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

CUDA: elliptic solver sample code

See the ATPESC15 github for a full implementation [with some optional goodies]



Compile each example with: `make`

Run with a 102x102 grid and tolerance 1e-4: `./main 100 1e-4`

Let me know if you have any problems with this.

CUDA: list of possible exercises

1.0: Read overview of accelerator hardware from compiler writer's perspective:
- http://www.pgroup.com/lit/articles/insider/v5n2a1.htm

1.1: Study event timing used to estimate compute time:
- see NVIDIA tutorial on event timing: link

1.2: Upgrade to higher order (wider) finite difference stencil (stencil weights: link)

1.3: Change the kernel & host code to use the conjugate gradient method: (wiki link)

1.4: Use shared memory to reduce the high-latency device memory traffic.

1.5: Use MPI+CUDA to partition the nodes across multiple GPUs.

1.6: Use a red-black numbering of the nodes to implement Gauss-Seidel iteration:
- tutorial by Jonathan Cohen (NVIDIA): link

1.7: Experiment with CUDA based OpenCurrent structured grid PDE framework:
- google code page: link

1.8: Study Mark Harris' CUDA optimized reduction: lecture notes: link

None of these tasks are particularly difficult, but they are ordered by difficulty.

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*

Many-core Processor Definition

In much the same vain…

> "A many-core processor is a device for turning a compute-bound problem into a memory-bound problem."
>
> Kathy Yelick

The latest GPUs have O(2800) floating point units
but only O(300) GB/s memory bandwidth off chip…

See more Yelick insight: http://isca09.cs.columbia.edu/ISCA09-WasteParallelComputer.pdf
http://static.og-hpc.org/Rice2011/Workshop-Presentations/OG-HPC%20PDF-4-WEB/Yelick%20Exascale-

… 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

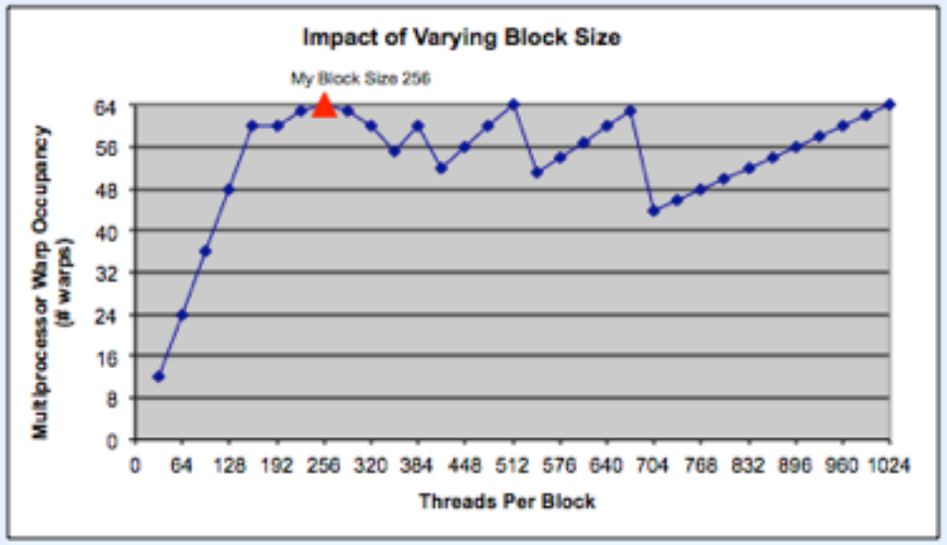
Note: SM is an abbreviation for (Streaming) Multiprocessor

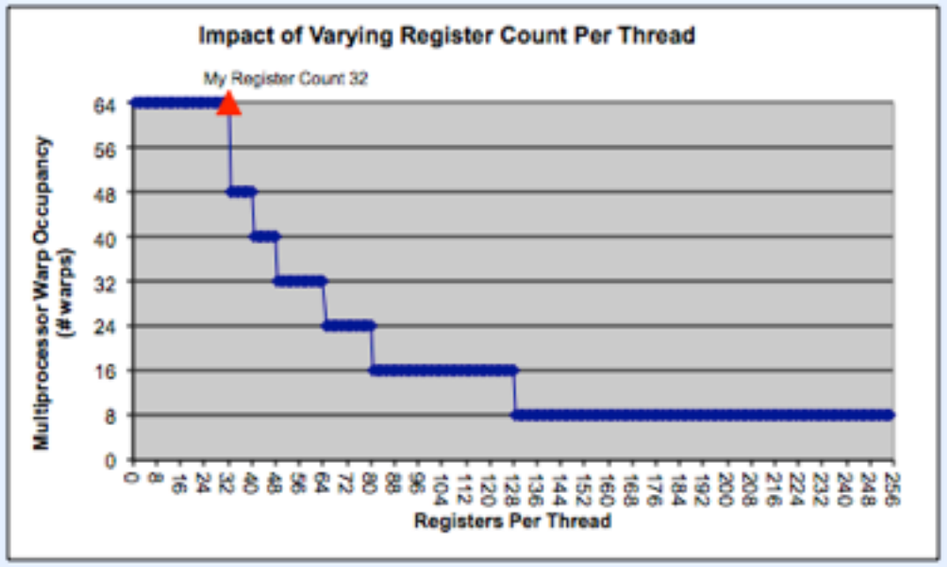
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	8	8	64
Limited by Shared Memory per Multiprocessor	12	8	96

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.

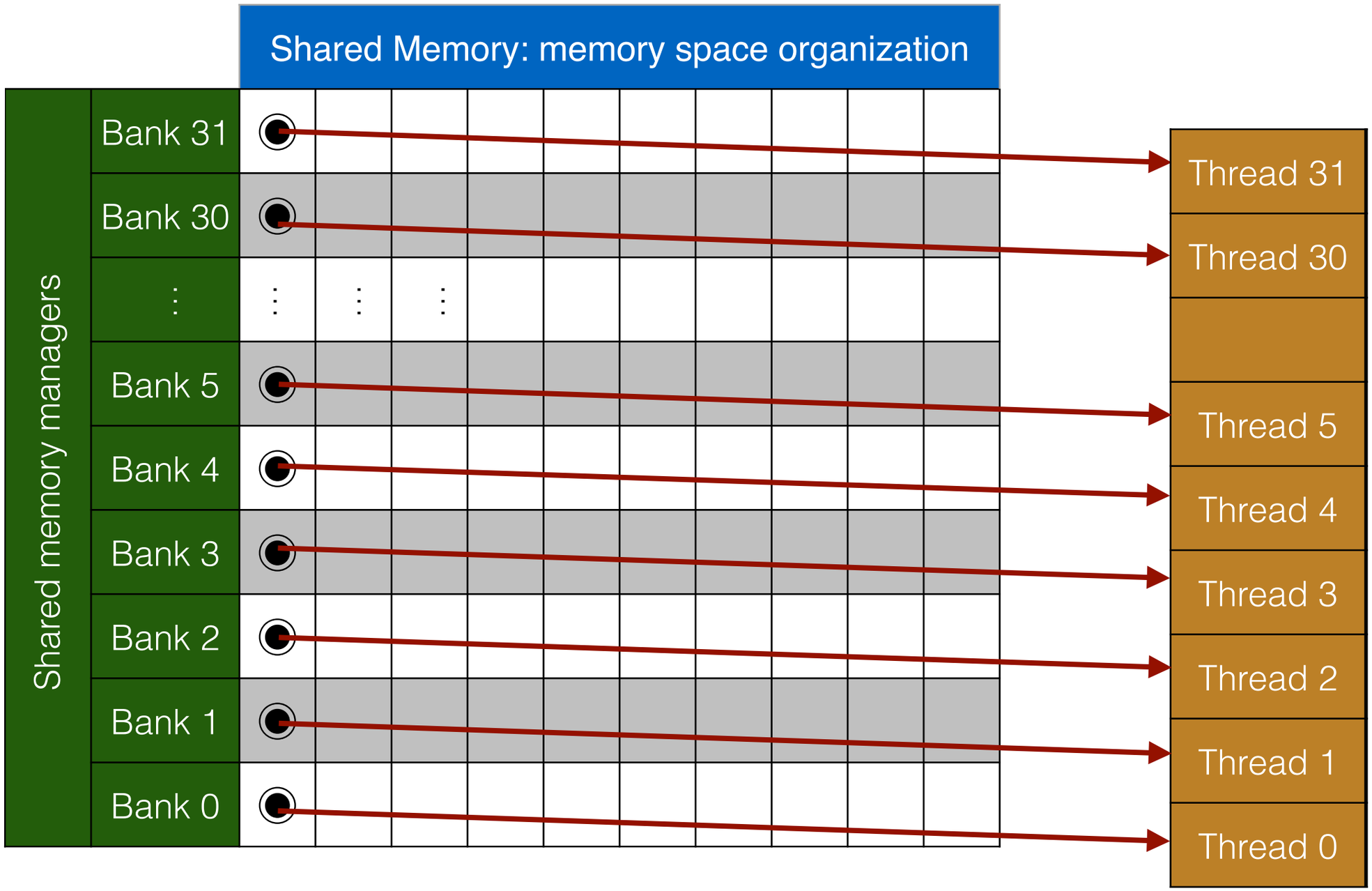
CUDA: shared memory banks

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

Shared memory managers	Shared Memory: memory space organization									
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

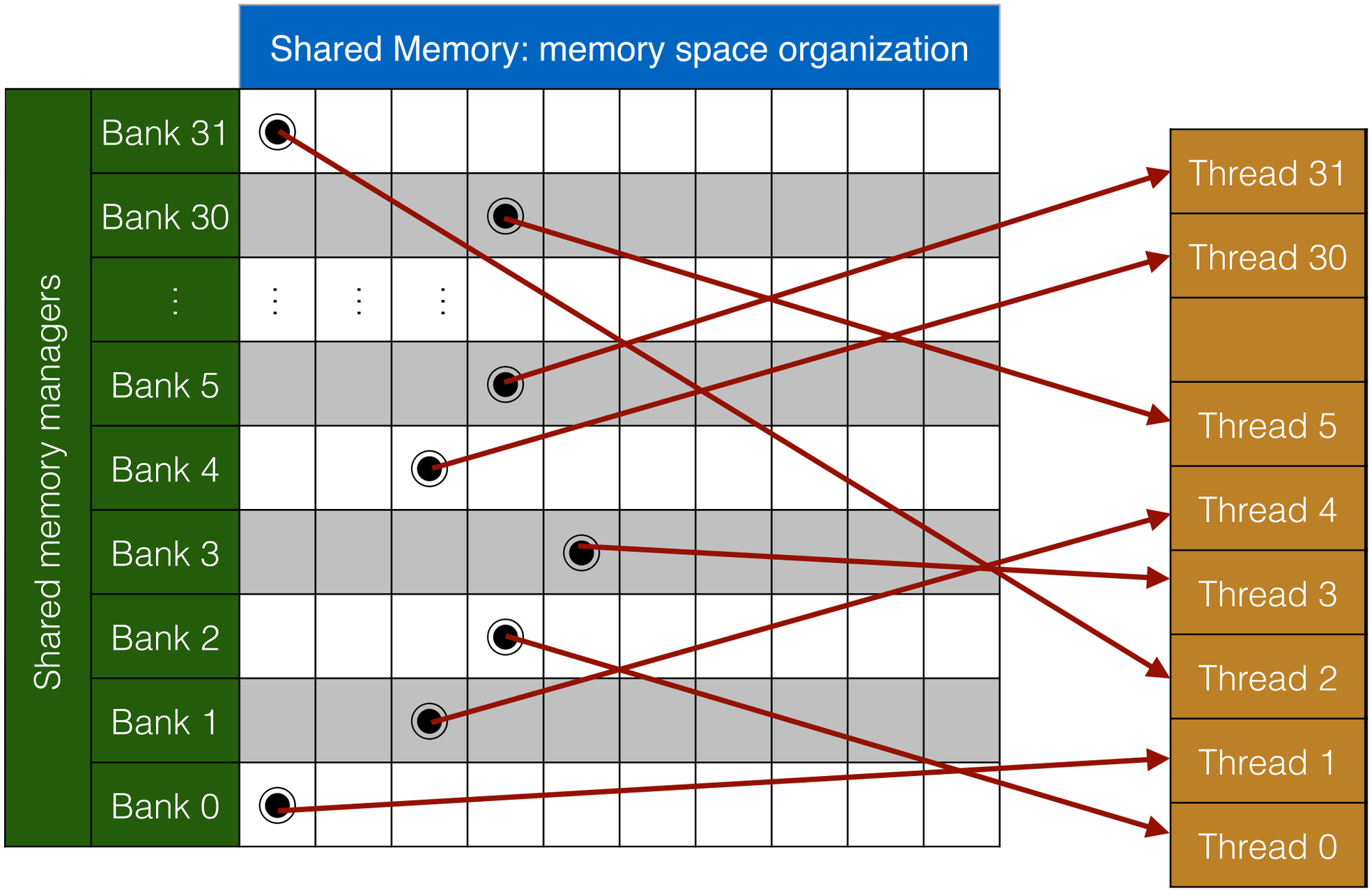
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.



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

CUDA: shared memory banks

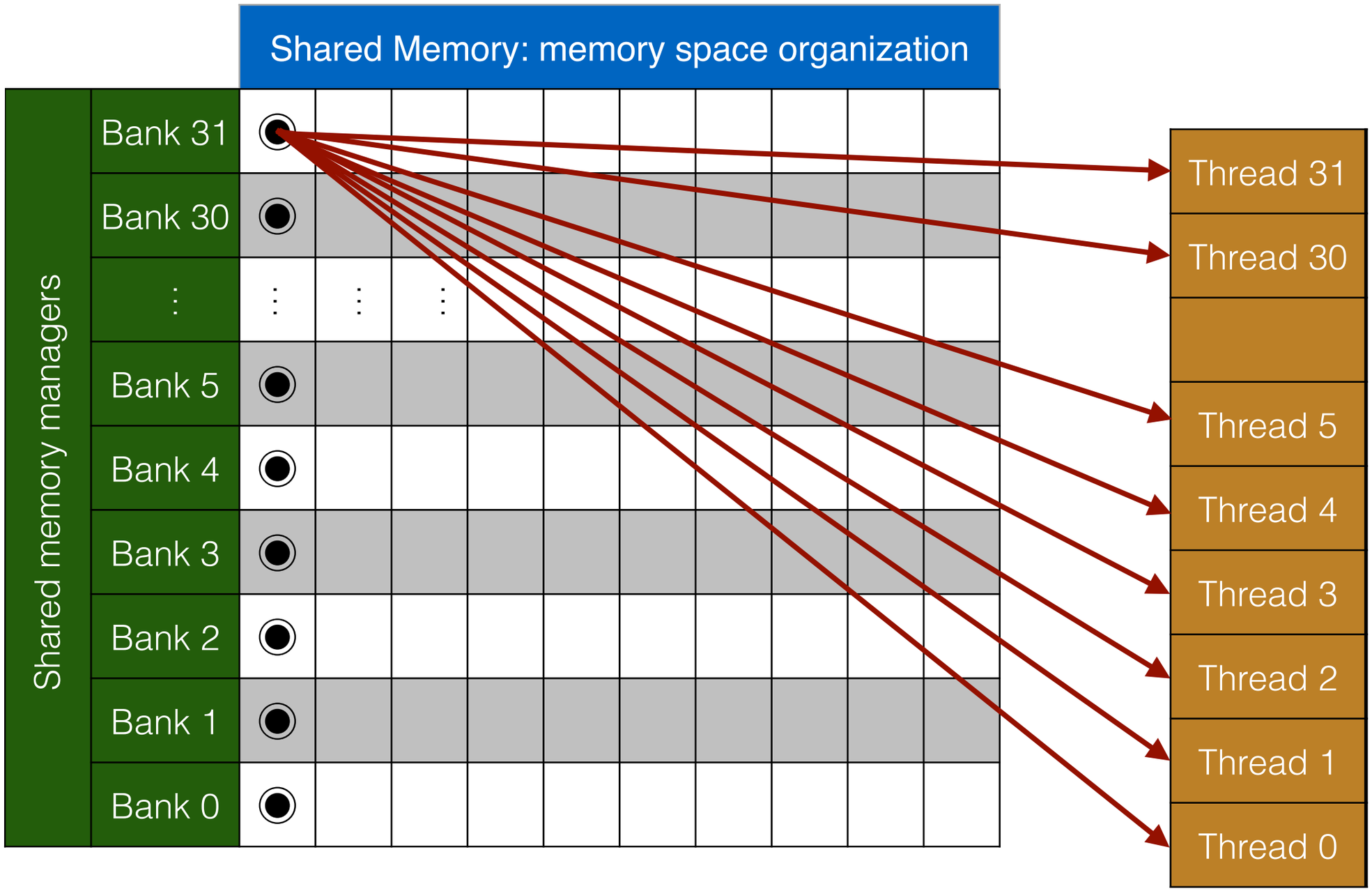
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.



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

CUDA: shared memory broadcast

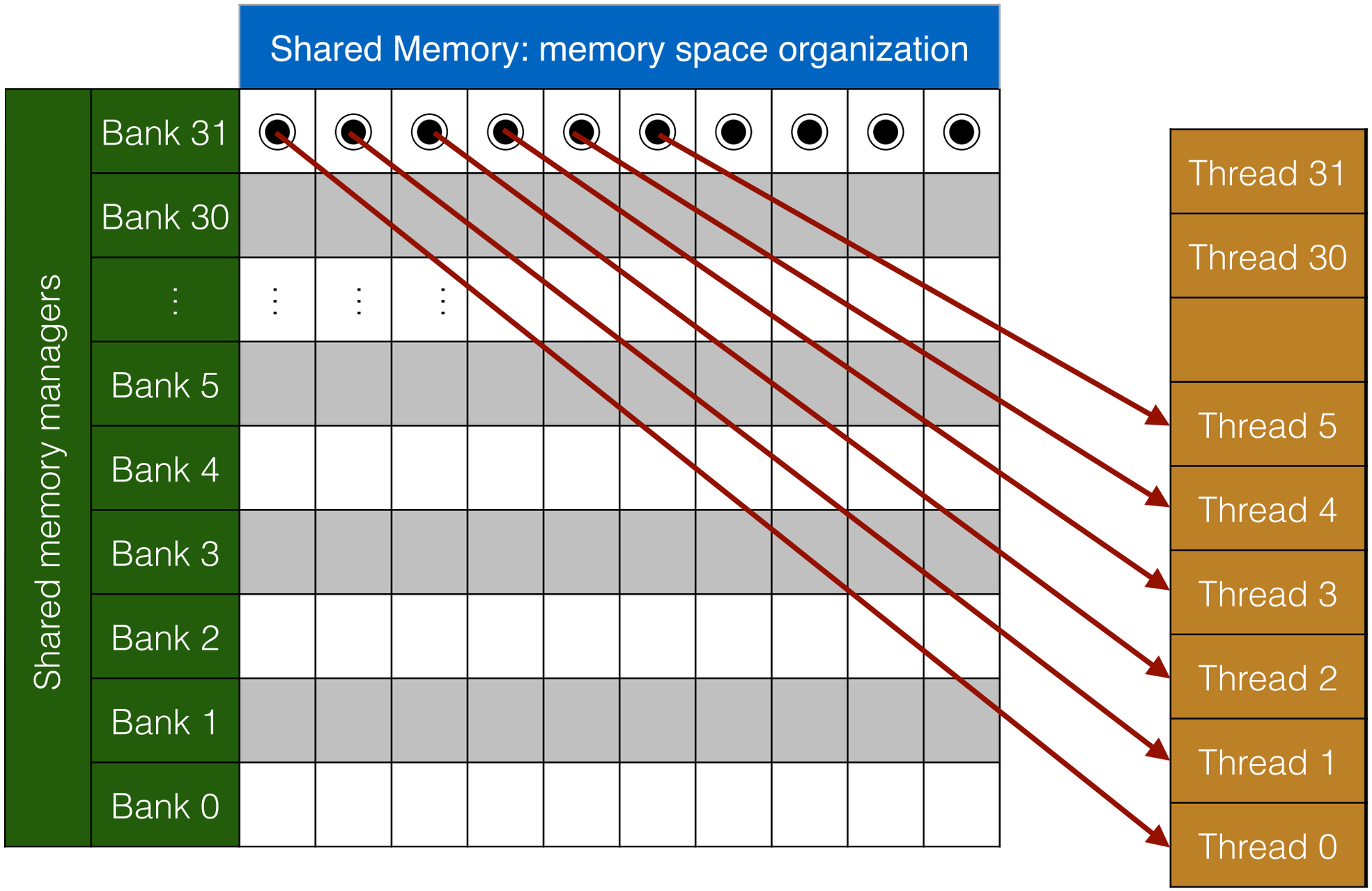
To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.



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.



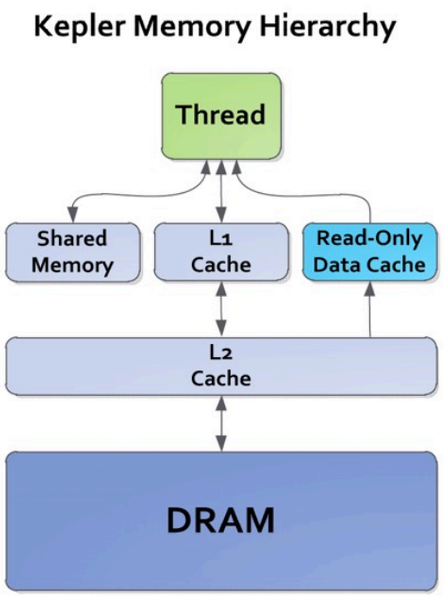
BAD: all threads in the SIMD group access the <u>same bank</u> resulting in serialization.

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 SIMD lane 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.

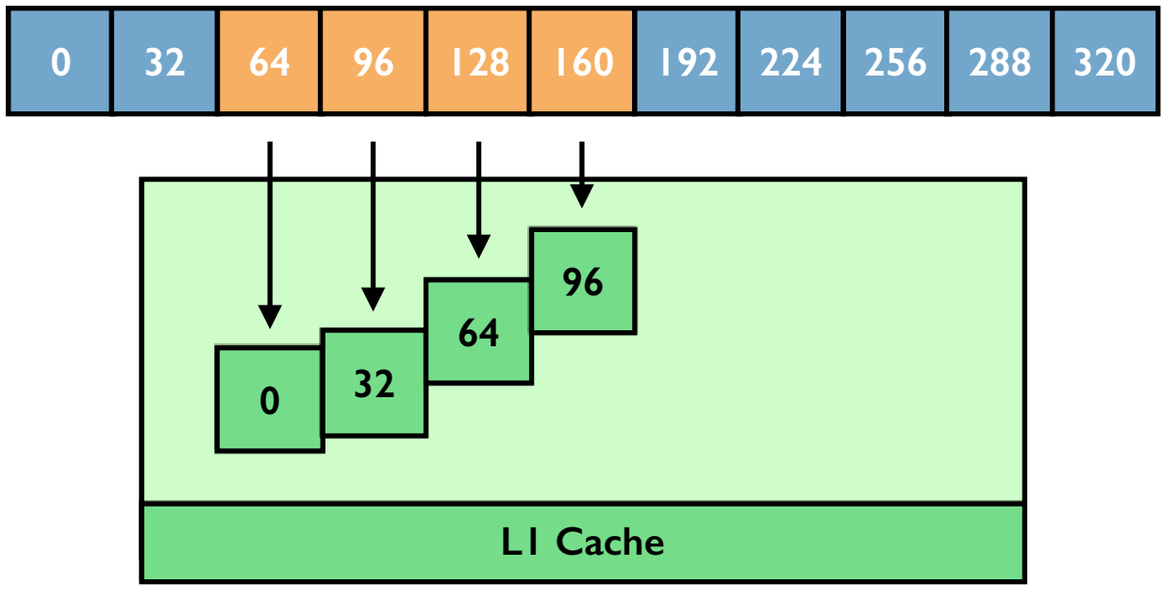


Rule of thumb: avoid non unitary stride DEVICE (DRAM) array access.
Useful slides , these , and image credit: link

CPU Optimization Techniques

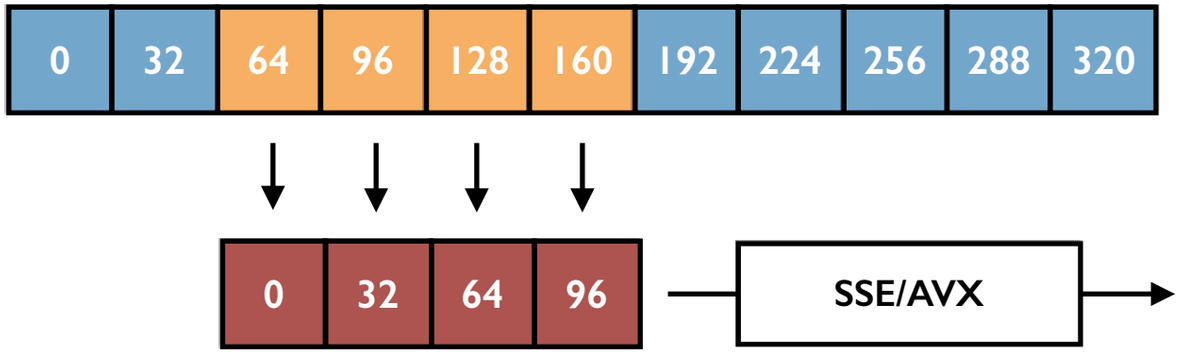
Cache

- 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.

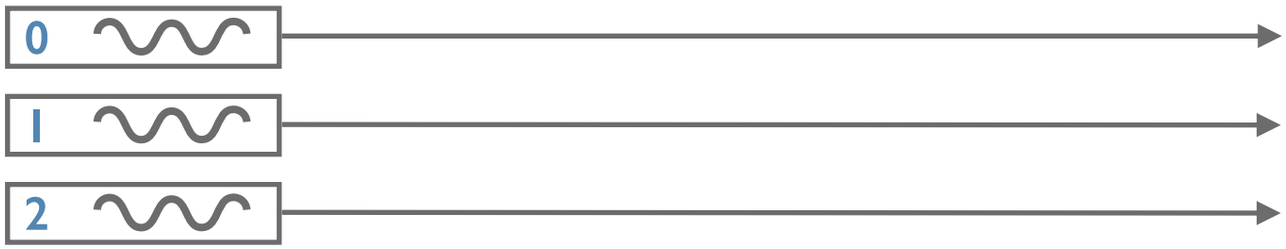


Continuous memory accesses are used for both, cache storage and vectorization

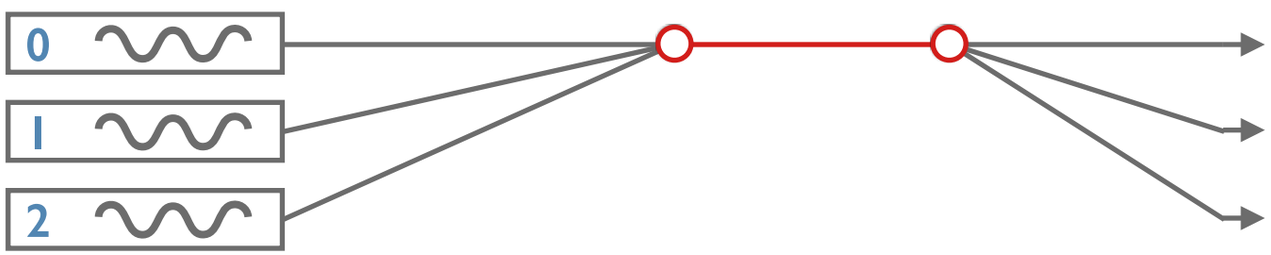
CPU Optimization Techniques

Multithreading

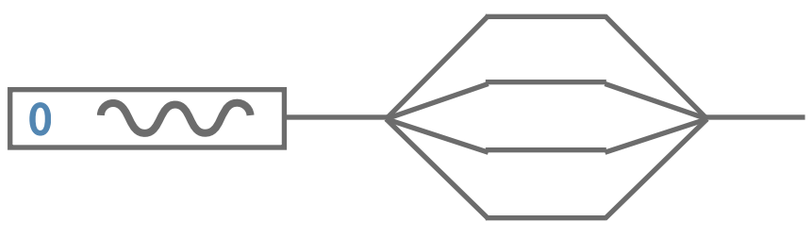
- Threads capable of fully parallelizing generic instructions (ignoring bandwidth).



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



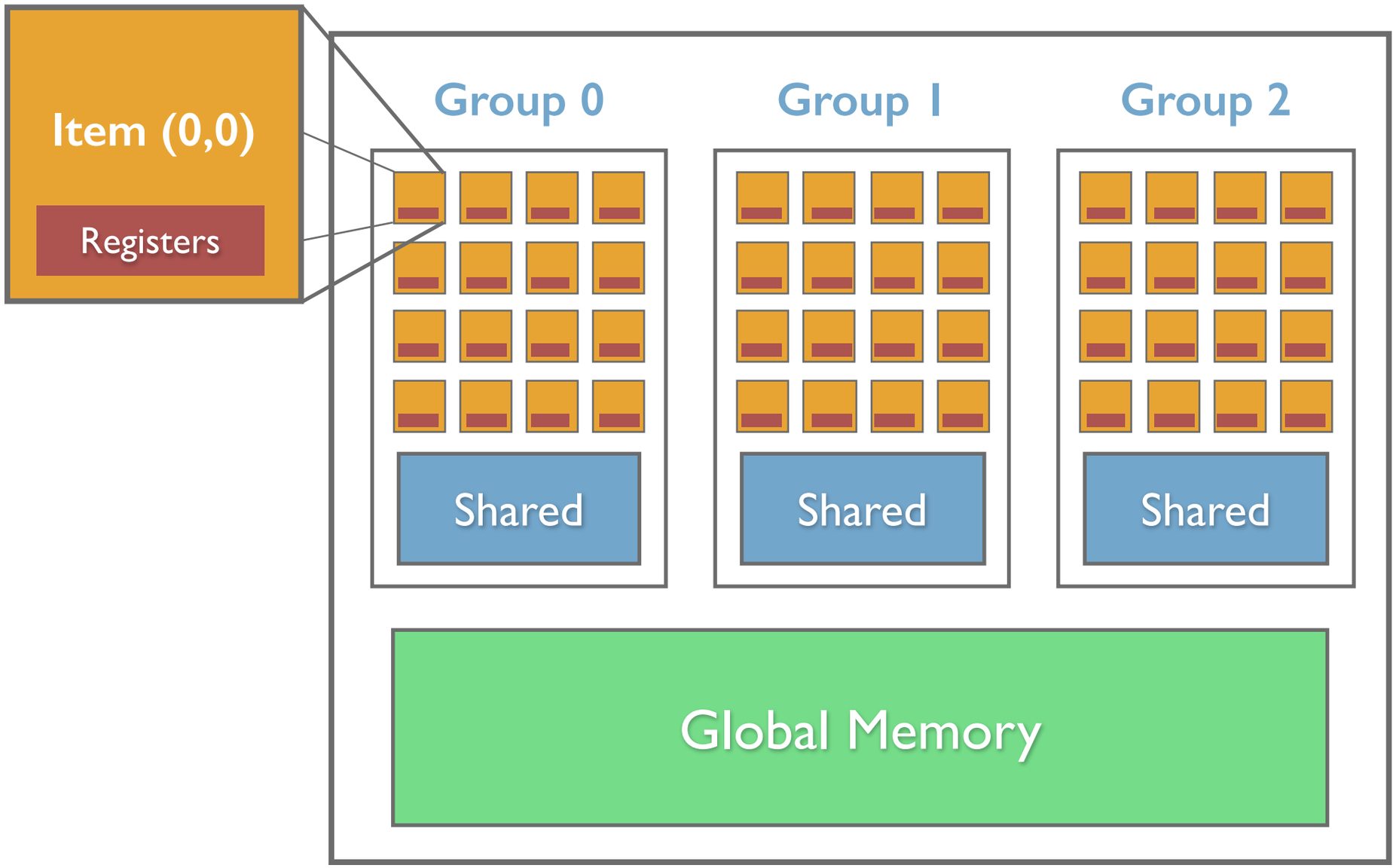
- SIMD Lanes



GPU Optimization Techniques

GPU Architecture

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

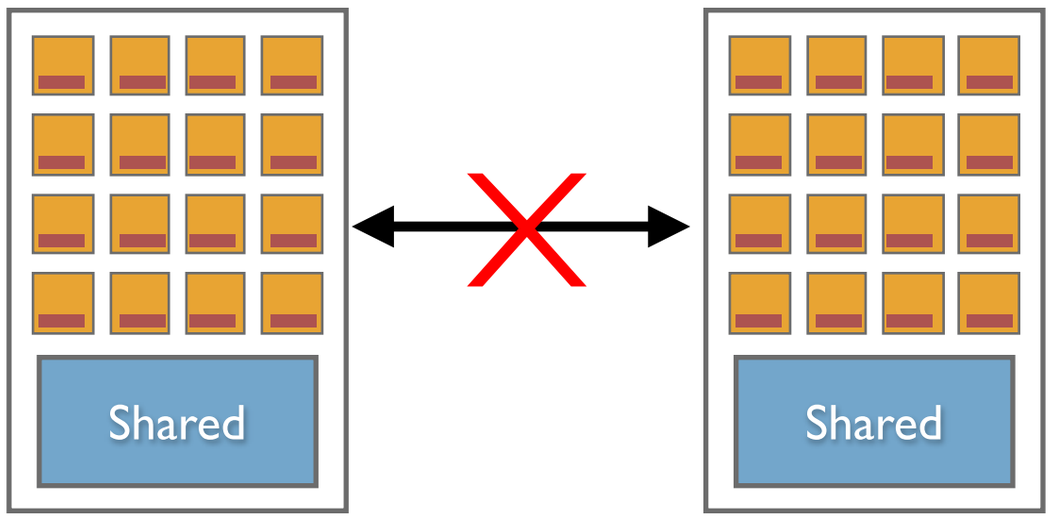


Kernel code describes the work-item operations

GPU Optimization Techniques

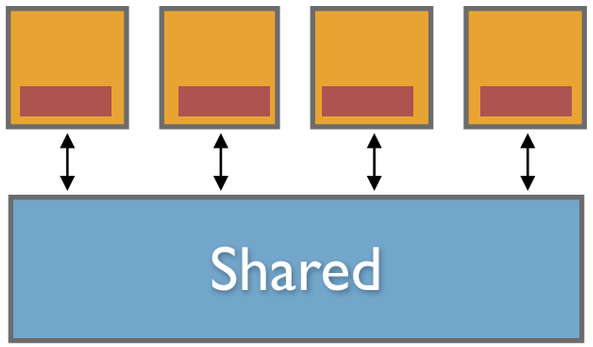
Work-groups

- 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

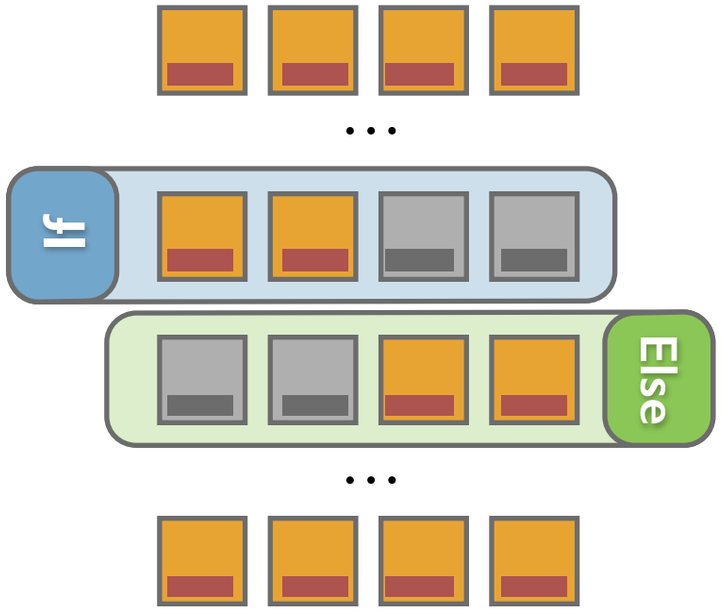
- Work-items are executed in parallel, able to barrier and share data using shared memory (& CUDA's shuffle).
- Avoid data race dependencies between work-items.



GPU Optimization Techniques

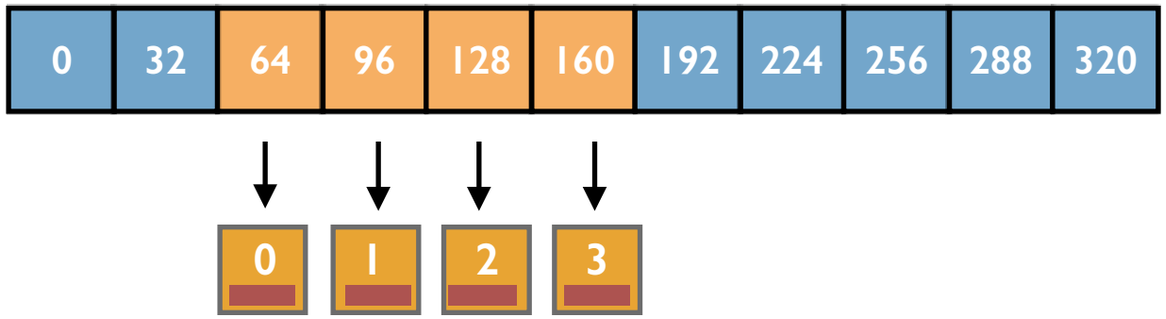
Parallel Work-item Execution

- 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

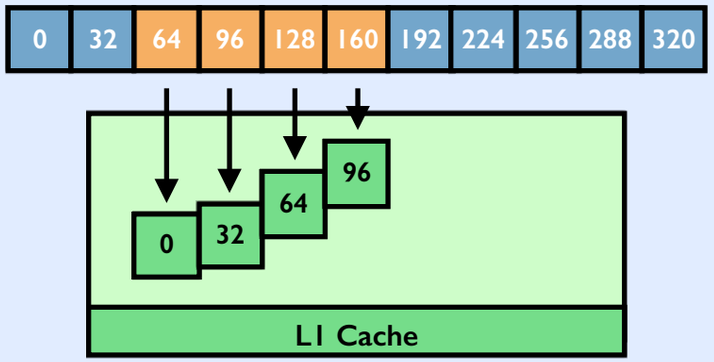
- Low individual bandwidth and high latency.
- Coalesced memory access on contiguous and aligned work-items.



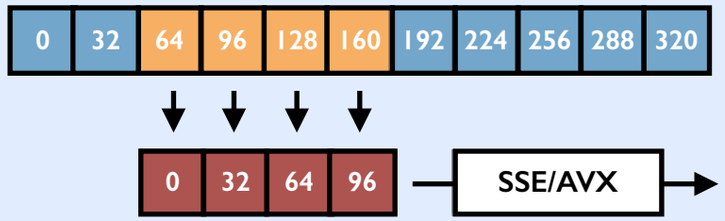
CPU & GPU Similarities

CPU Optimizations

Cache

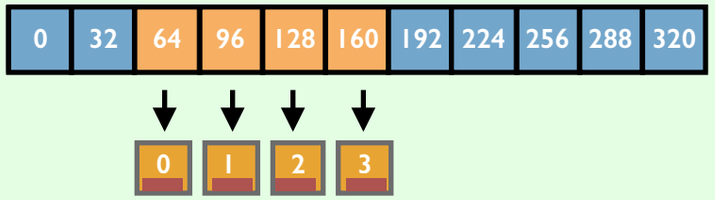


Vectorization

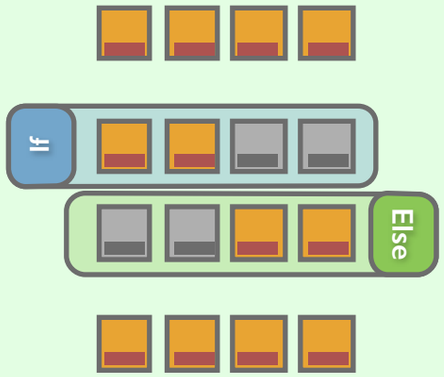


Thread Independence

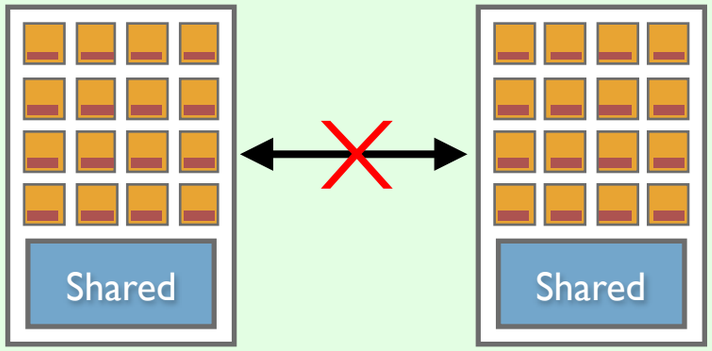
Coalescing



No Branching



Work-group Independence



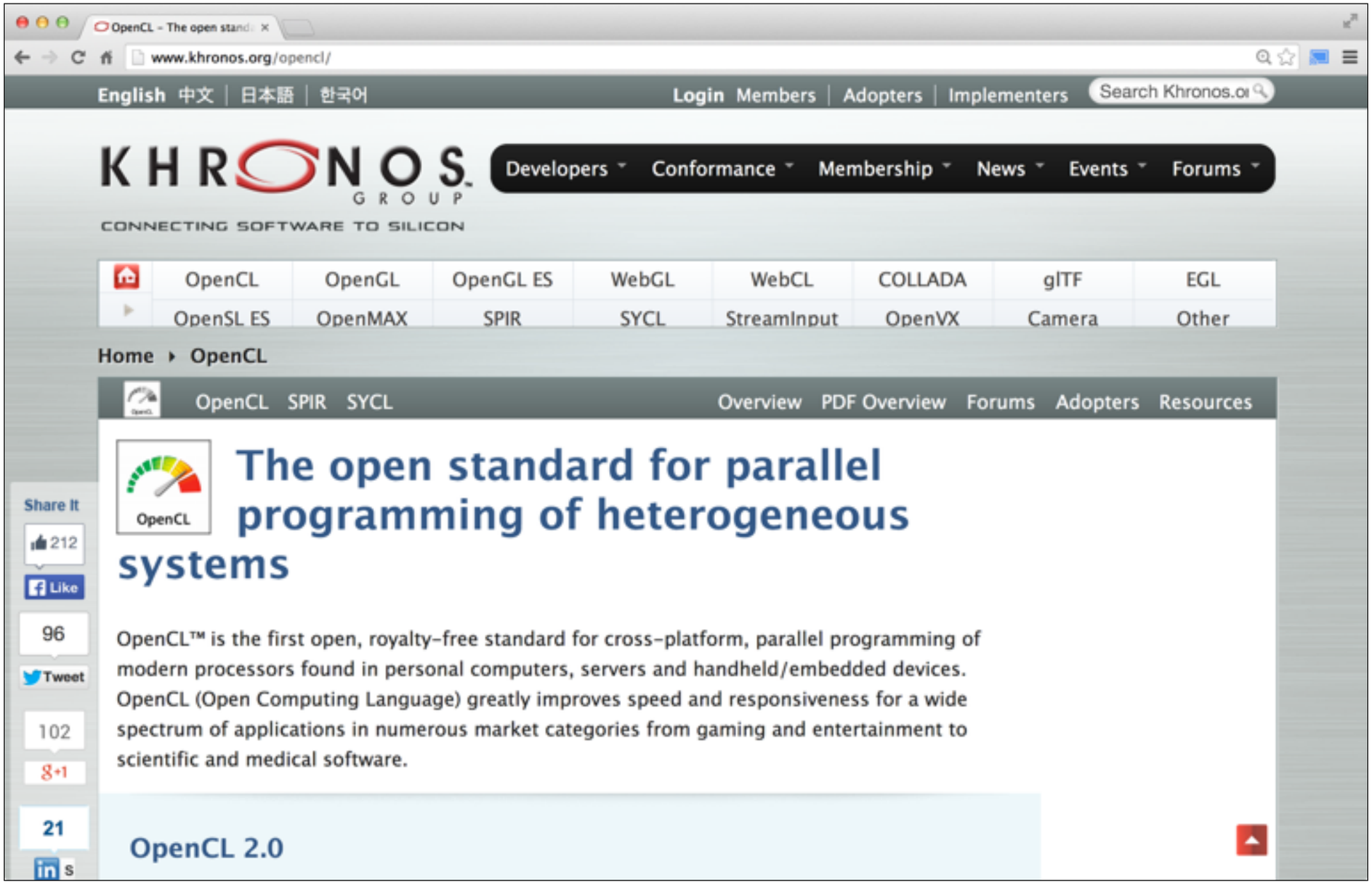
GPU Optimizations

Exposing vectorization / SIMD parallelism are vital in both architectures

Part 4: Open Compute Language (OpenCL)

Apple drove the creation of the OpenCL standard for portable cross-vendor many-core programming.

OpenCL: standards body



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 ?

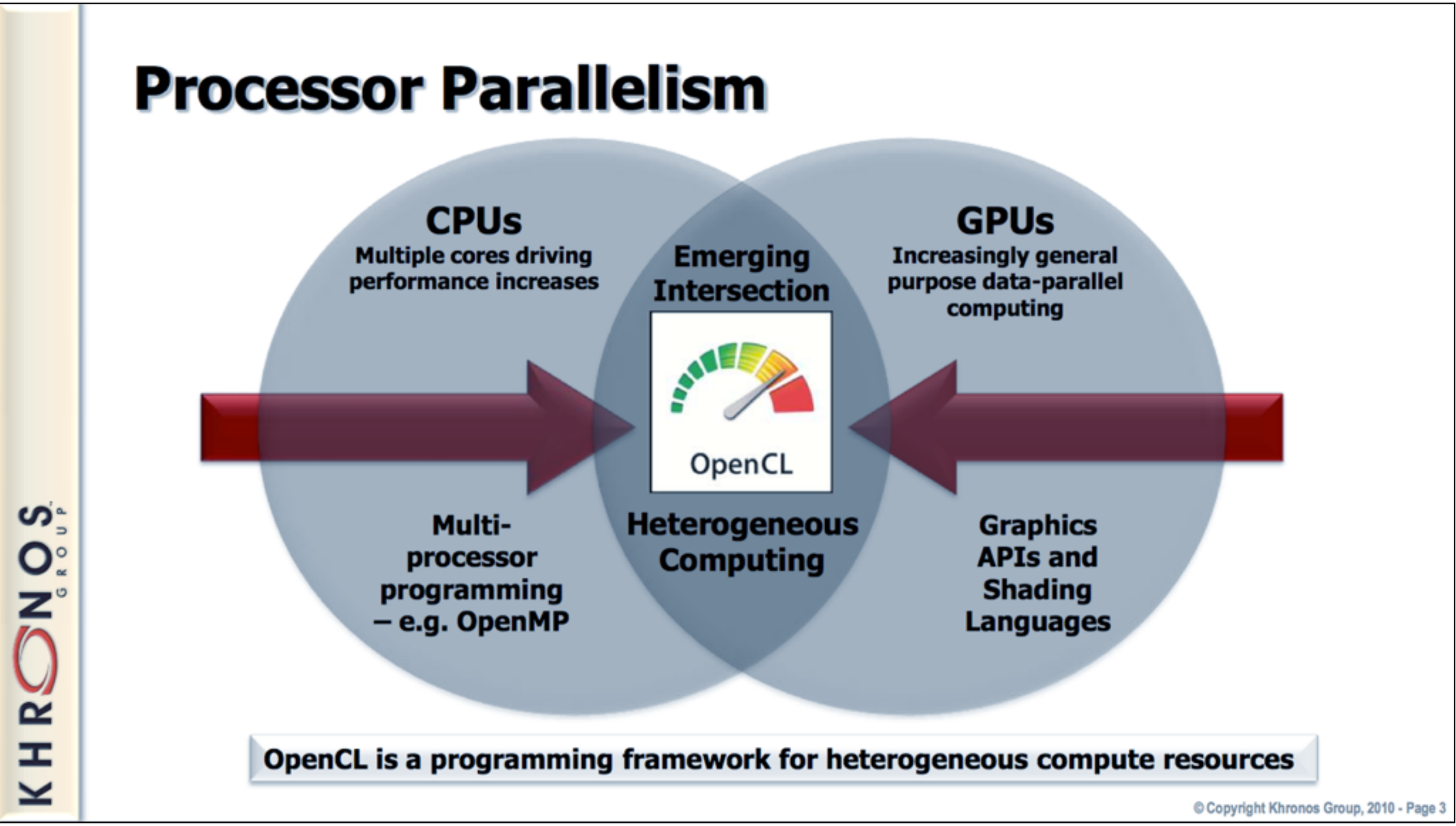
OpenCL Working Group

- **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

KHRONOS GROUP

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

OpenCL: why ?



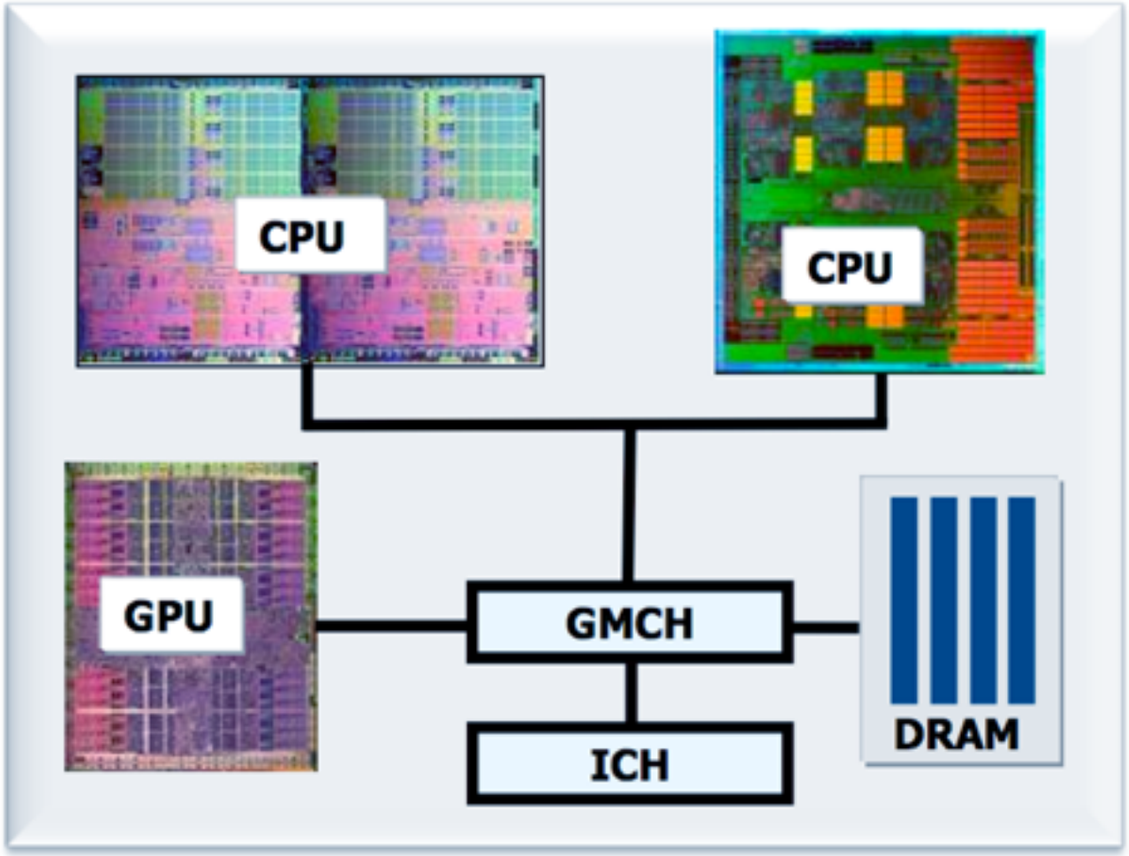
Emphasis on heterogeneous computing.
Credit: Khronos Group

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

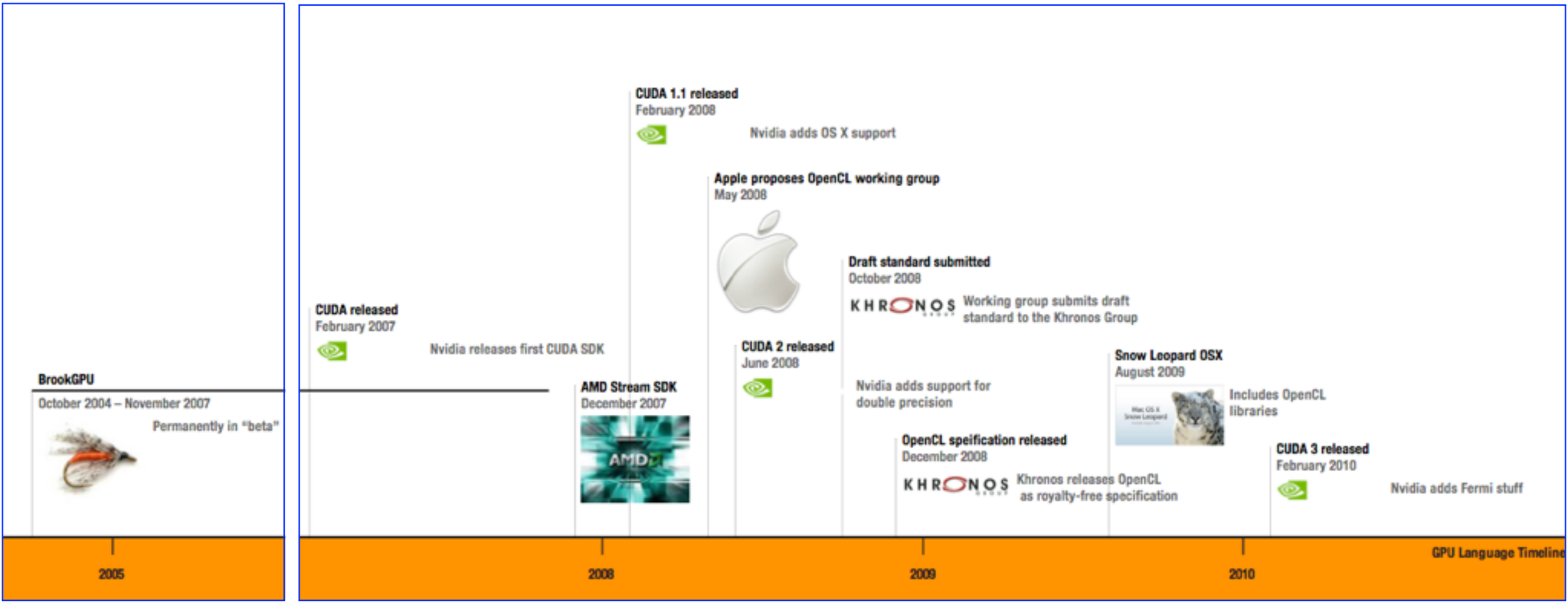


GMCH = graphics memory control hub
ICH = Input/output control hub

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

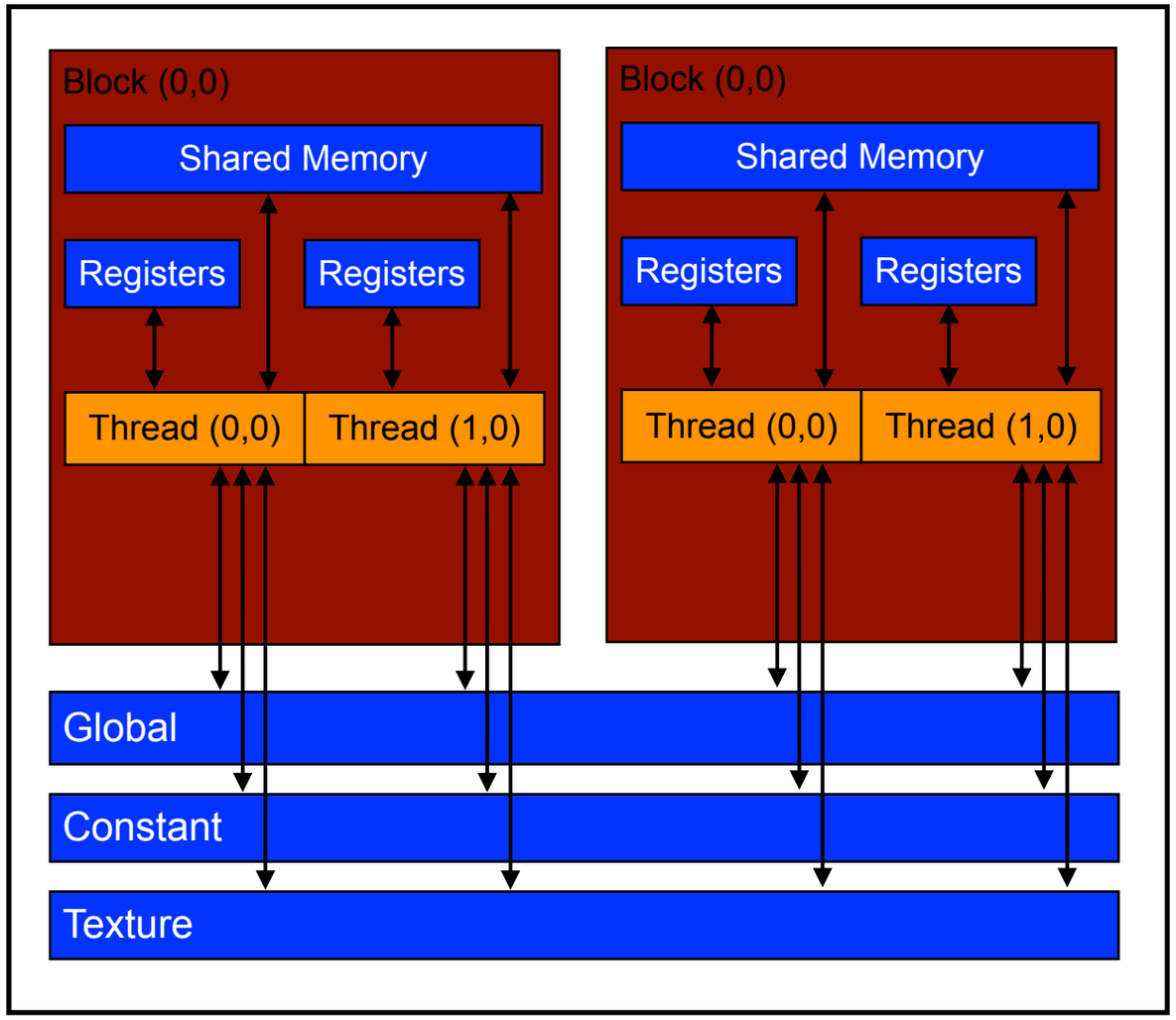
OpenCL is ** very ** closely related to CUDA

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

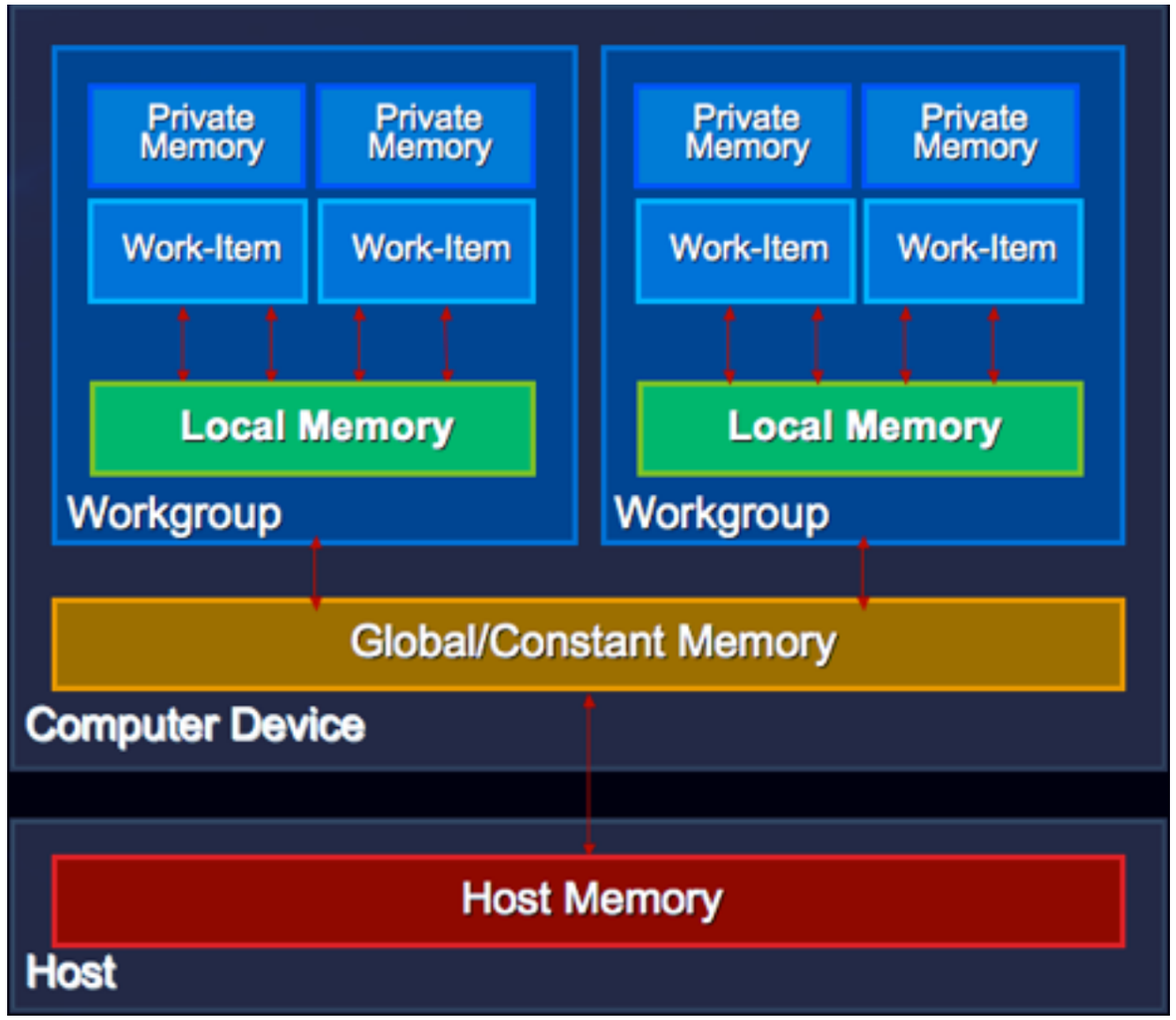
The rapid development of OpenCL helps explain the similarities

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:

1. Represent kernel source code as a C character array:

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

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: simple kernel example

You can look at the example OpenCL code in its full glory in the ATPESC15 github

GitHub, Inc.

This repository Search Pull requests Issues Gist

tcew / ATPESC15 Unwatch 1 Star 0 Fork 0

Branch: master ATPESC15 / examples / opencl / simple / simple.cpp

Tim Warburton a day ago starting

0 contributors

169 lines (127 sloc) 4.692 kB Raw Blame History

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

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

void pfn_notify(const char *errinfo, const void *private_info, size_t cb, void *user_data)
{
  fprintf(stderr, "OpenCL Error (via pfn_notify): %s\n", errinfo);
}

int main(int argc, char **argv){
```

Queue the live demo.

OpenCL: compiling & running example

There may be variations depending on how your system is set up.

```
# compile on node with the g++ compiler and an OpenCL framework

# Linux:
g++ -o simple simple.cpp -lOpenCL

# OS X
g++ -o simple simple.cpp -framework OpenCL

# run on node that has OpenCL installed
./simple
```

Make sure you can complete this exercise !

Source code: https://github.com/tcew/HPC15/examples/cuda/simple

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 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;

 // 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 datafloat *rhs
                     __global const datafloat *u,
                     __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]);
  }
}
```

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!!

The example code in ATPESC15/examples/opencl/jacobi/partialReduce.cl is verbose

OpenCL: partial 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;

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

  s_blocksum[threadIdx.x] = 0;

  if(id < entries){
    const datafloat 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];
}
```

The example code in ATPESC15/examples/opencl/jacobi/partialReduce.cl is verbose

OpenCL: partial reduction

Standard tree reduction at the thread-block level!!

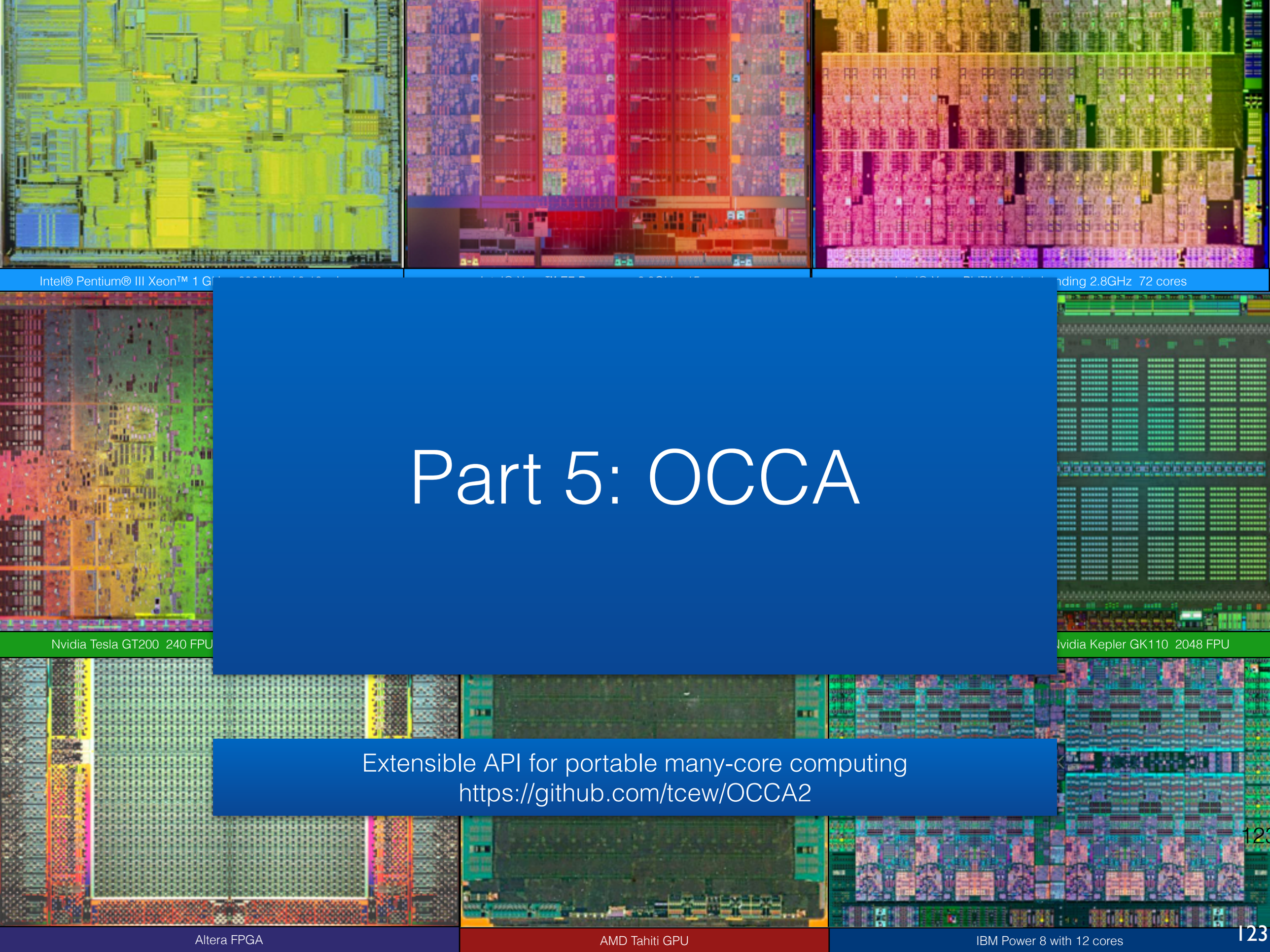
CUDA partial reduction kernel:

```
__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;

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

  s_blocksum[threadIdx.x] = 0;

  if(id < entries){
    const datafloat 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,
                            __global const datafloat *u,
                            __global const datafloat *newu,
                            __global datafloat *blocksum){

 __local datafloat s_blocksum[BDIM];

  const int id = get_global_id(0);

  int alive = get_local_size(0);
  int t = get_local_id(0);

  s_blocksum[t] = 0;

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

  while(alive>1){

    barrier(CLK_LOCAL_MEM_FENCE);  // 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];
}
```

The example code in ATPESC15/examples/opencl/jacobi/partialReduce.cl is verbose



Part 5: OCCA

Extensible API for portable many-core computing
https://github.com/tcew/OCCA2

Overview of OCCA

Introduction

- Background on many-core architectures

Device Abstractions

- OCCA Intermediate Representation (IR)
- Unified kernel languages (OKL/OFL) extend C and Fortran

Automation

- Automating data transfer
- Automatic kernel generation

Results

- Numerical methods (FD, MC, SEM, DG) and benchmarks

Conclusion

Review: common multi-threading APIs

OpenMP:

- Shared memory model.
- Directive based code parallelization.
- Standard maintained by the OpenMP Architecture Review Board.
- Targets: currently CPUs + Intel Xeon Phi, with plans for GPU support.

OpenACC:

- Directive based accelerator coding:
- Compilers: http://www.openacc-standard.org/content/tools
- Work in progress, may merge with OpenMP ?

CUDA:

- Manages host-device (accelerator) communications and on-device calculations.
- Kernel based calculations.
- Proprietary language developed by NVIDIA that targets NVIDIA accelerators [and x86].

OpenCL:

- Similar in structure to CUDA.
- Standard maintained by the Khronos Group.
- Targets: almost all CPUS, GPUs, FPGAs, and Accelerators.

OpenACC will be discussed this afternoon.

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 + ?

Does it even matter what “X” is ?

May's Law

Have you noticed that computers get better specs
but do not seem to get substantially faster:

"First, a disappointment. It is widely accepted that hardware efficiency doubles ever 18 months, following Moore's law. Let me now introduce you to May's law:

software efficiency halves every 18 months,
compensating Moore's law!

It's not clear what has caused this, but the tendency to add features, programming using copy-paste techniques, and programming by 'debugging the null-program' - starting with a debugger and an empty screen and debugging interactively until the desired program emerges - have probably all contributed."

David May
CSP, occam, and Transputers, Communicating Sequential Processes, pp. 75-84, 2004.

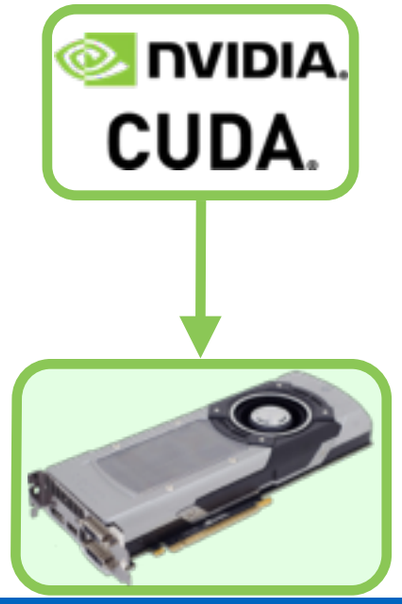
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012),
(Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

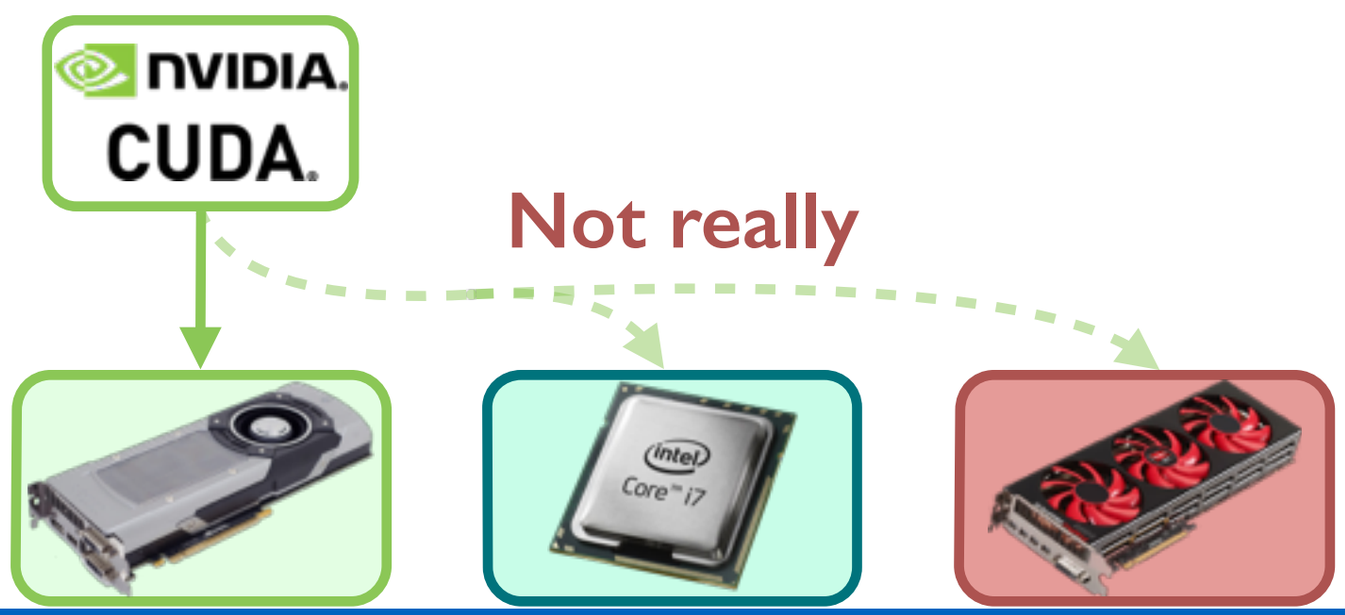
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012),
(Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

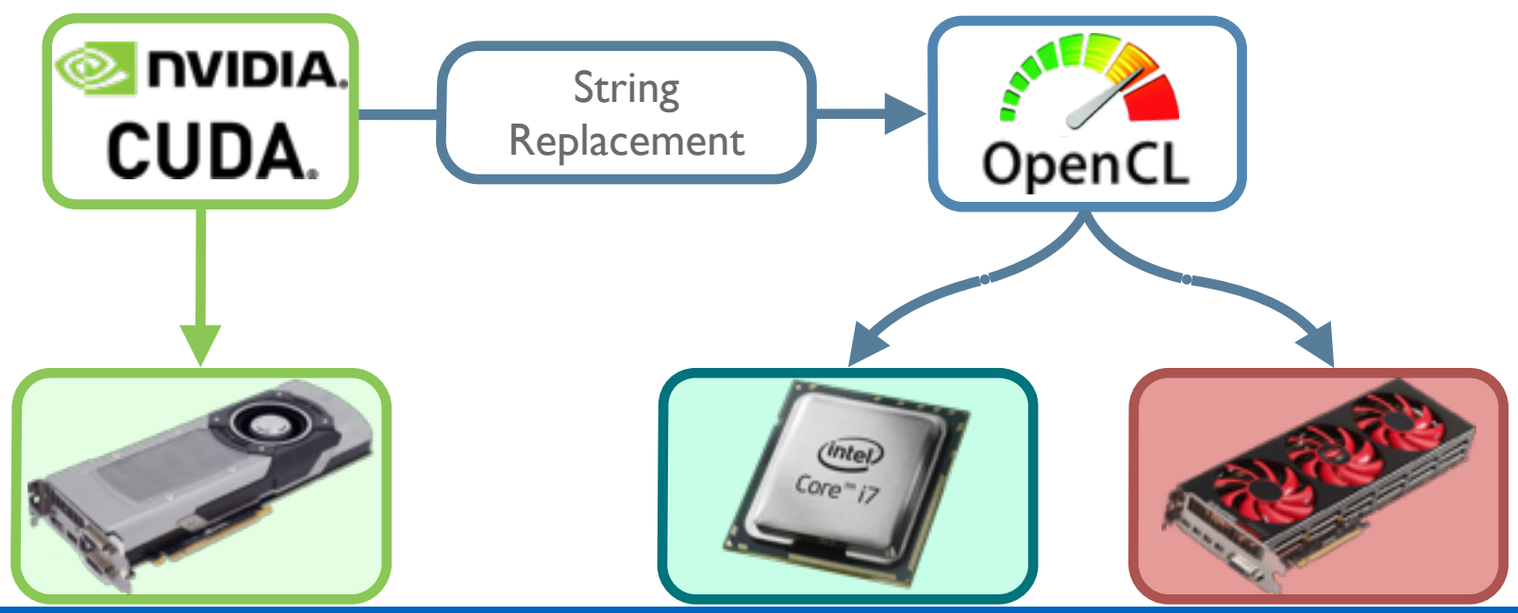
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012), (Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

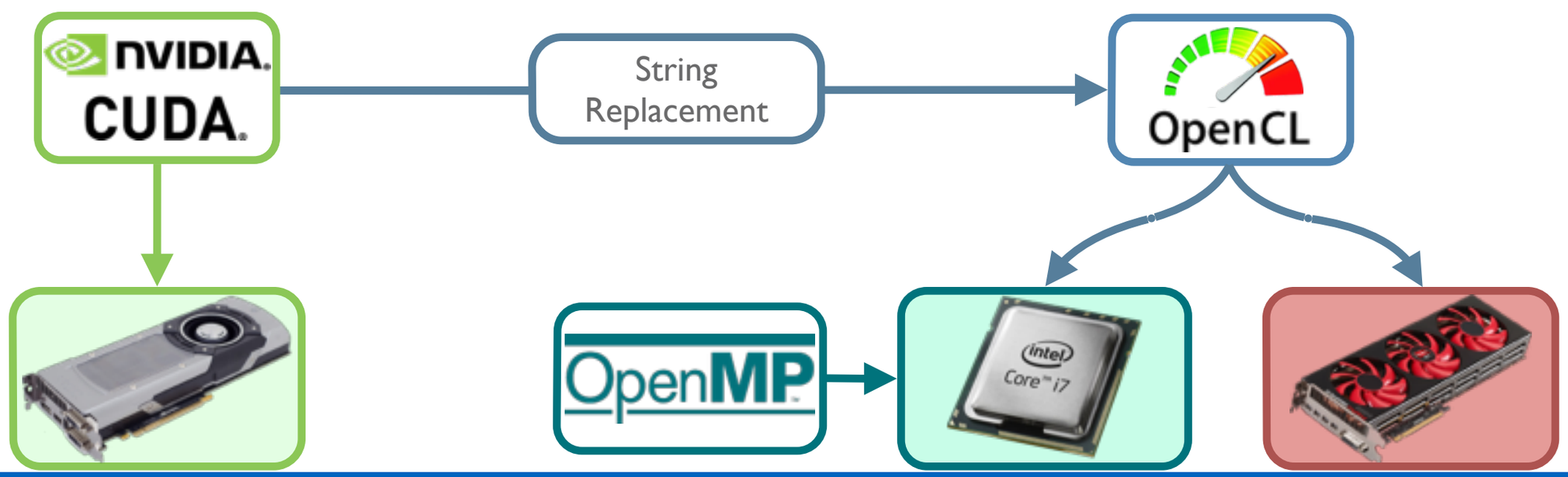
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012), (Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

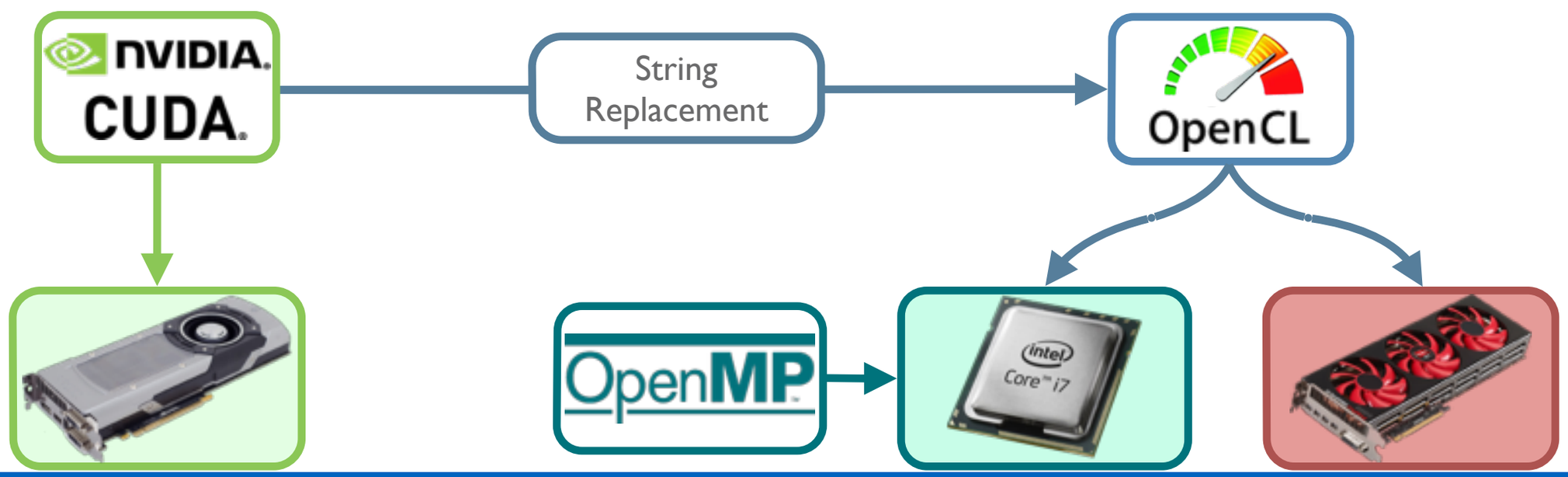
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012), (Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

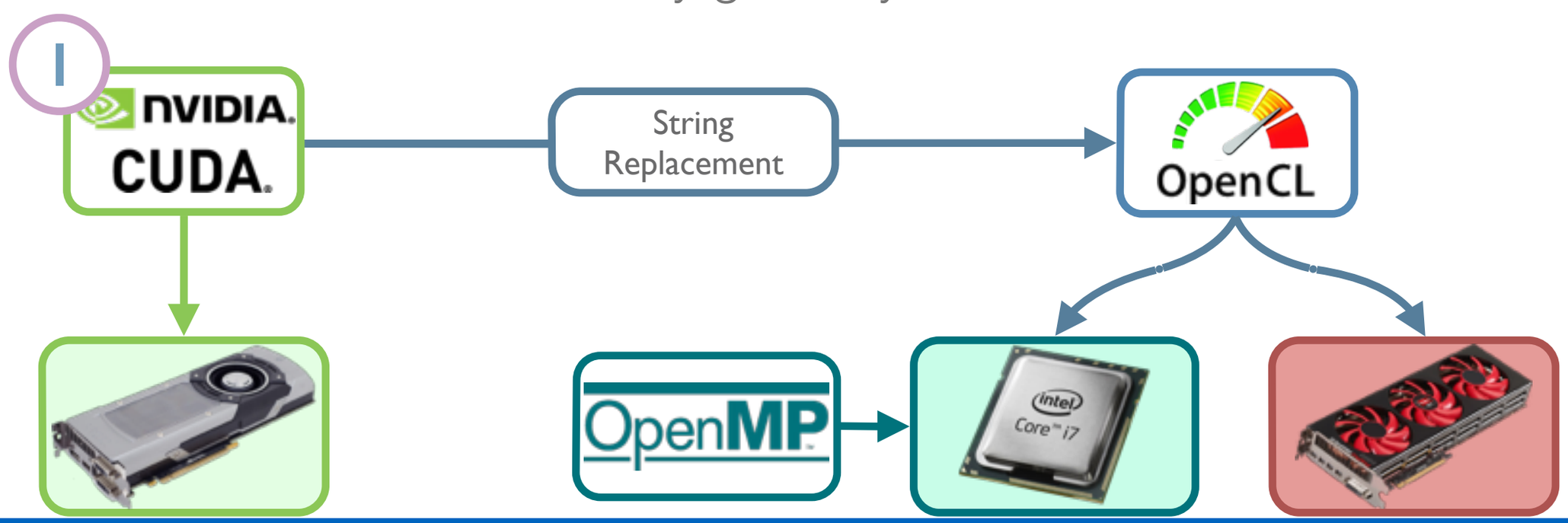
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012), (Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

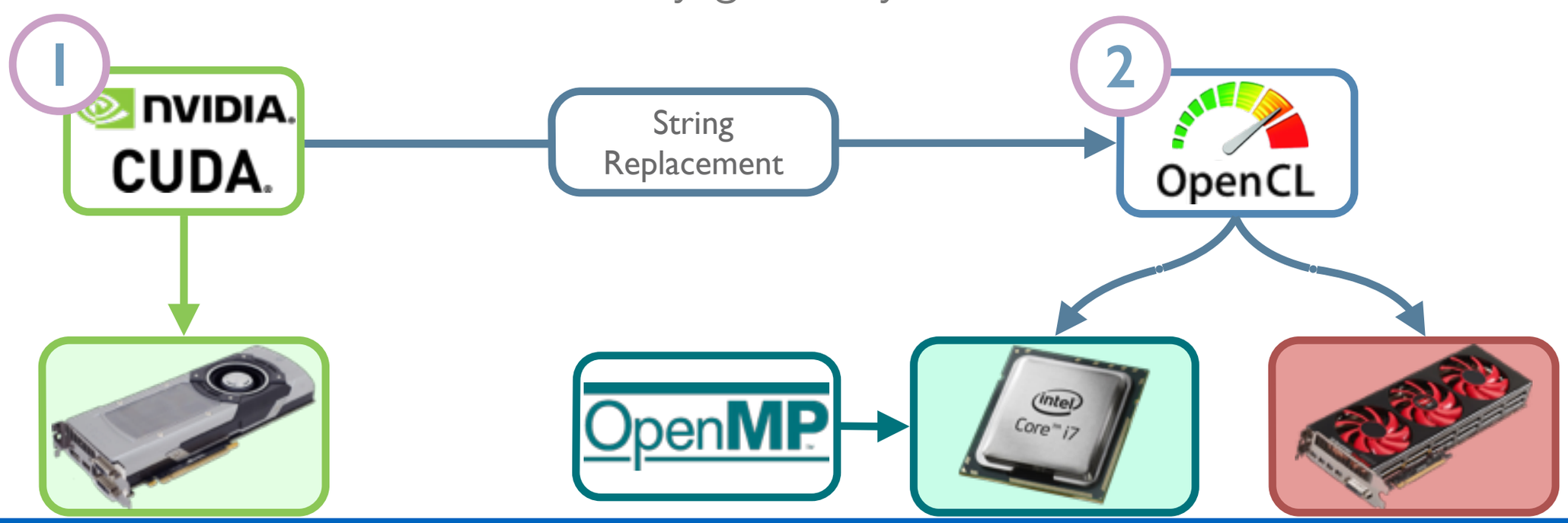
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012),
(Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

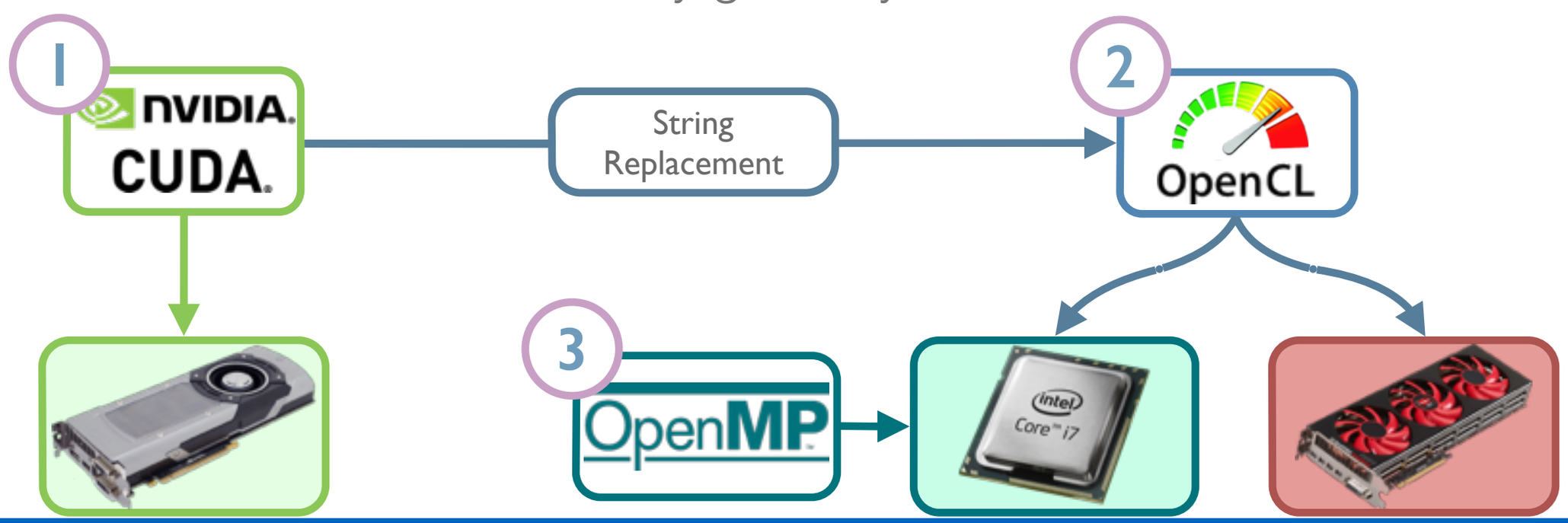
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012),
(Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

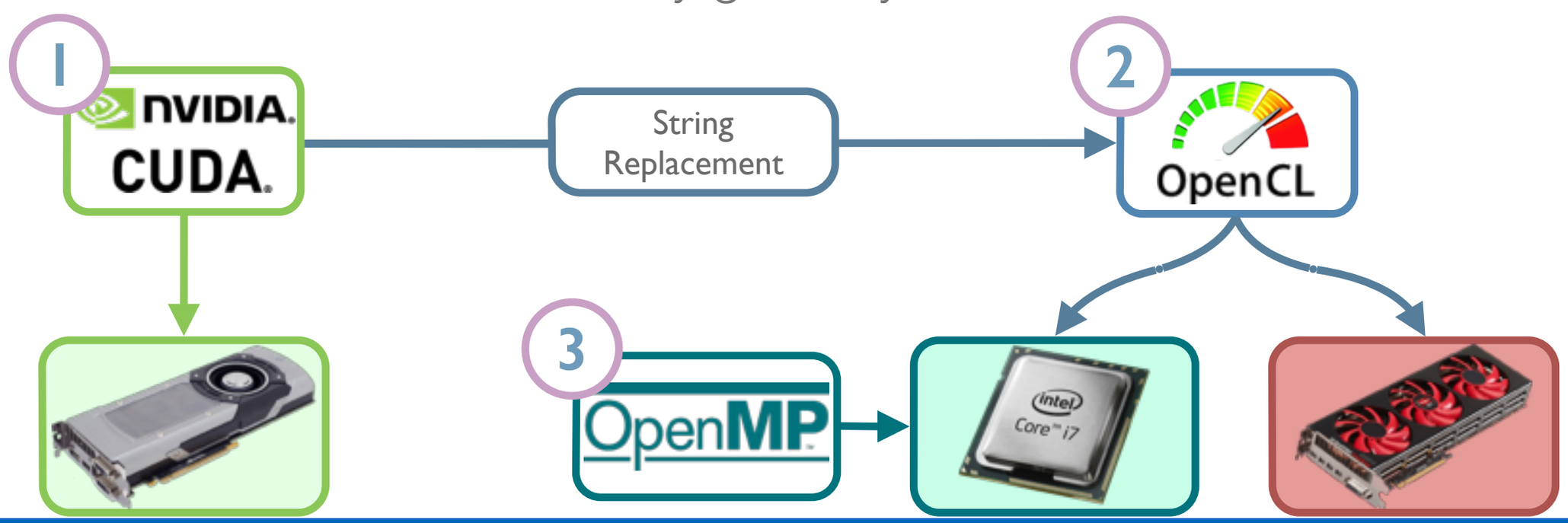
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012),
(Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

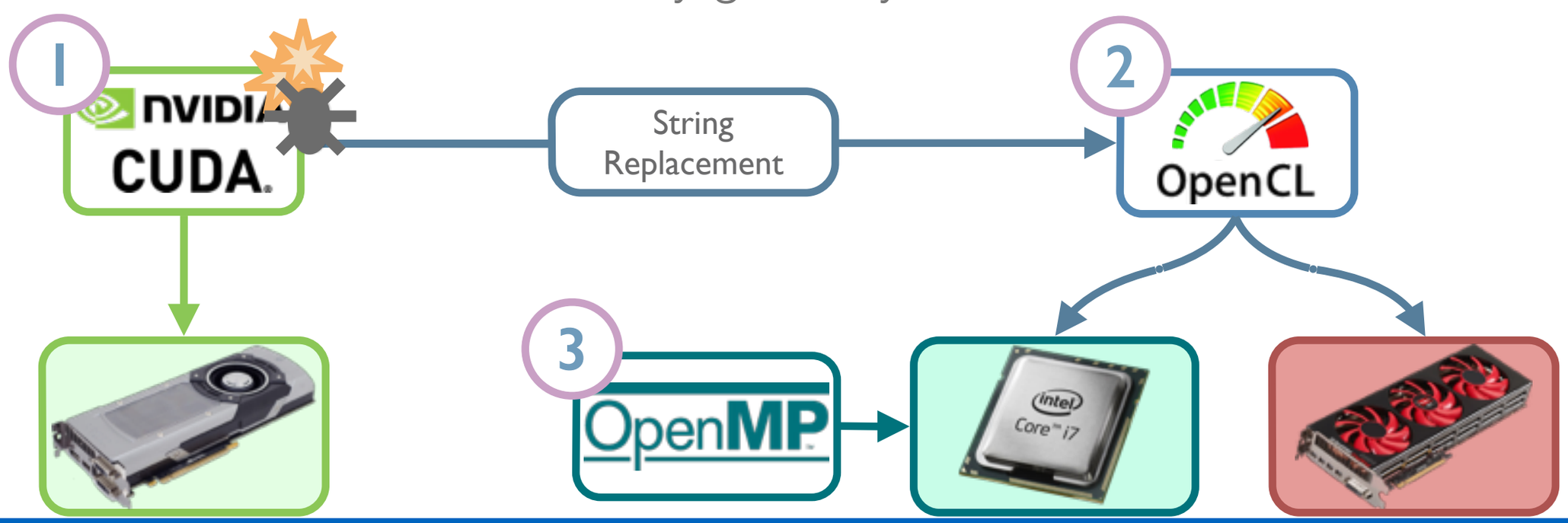
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012), (Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

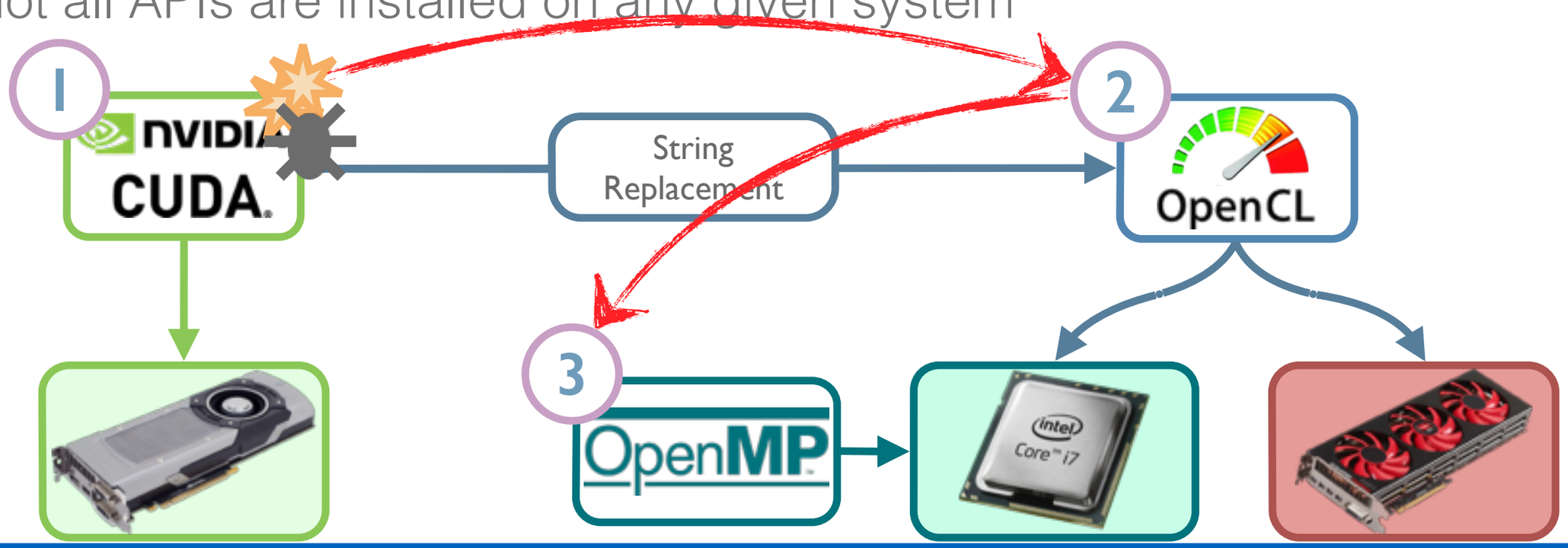
Motivation: Causes

Uncertainty

- Code life cycle measured in decades
- Architecture & API life cycles measured in Moore doubling periods
- Example: IBM Cell processor, IBM Blue Gene Q, PVM (Software)

Portability

- CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not code compatible
- Not all APIs are installed on any given system



(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012),
(Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

Subset of Approaches to Portability

Numerous approaches to portability

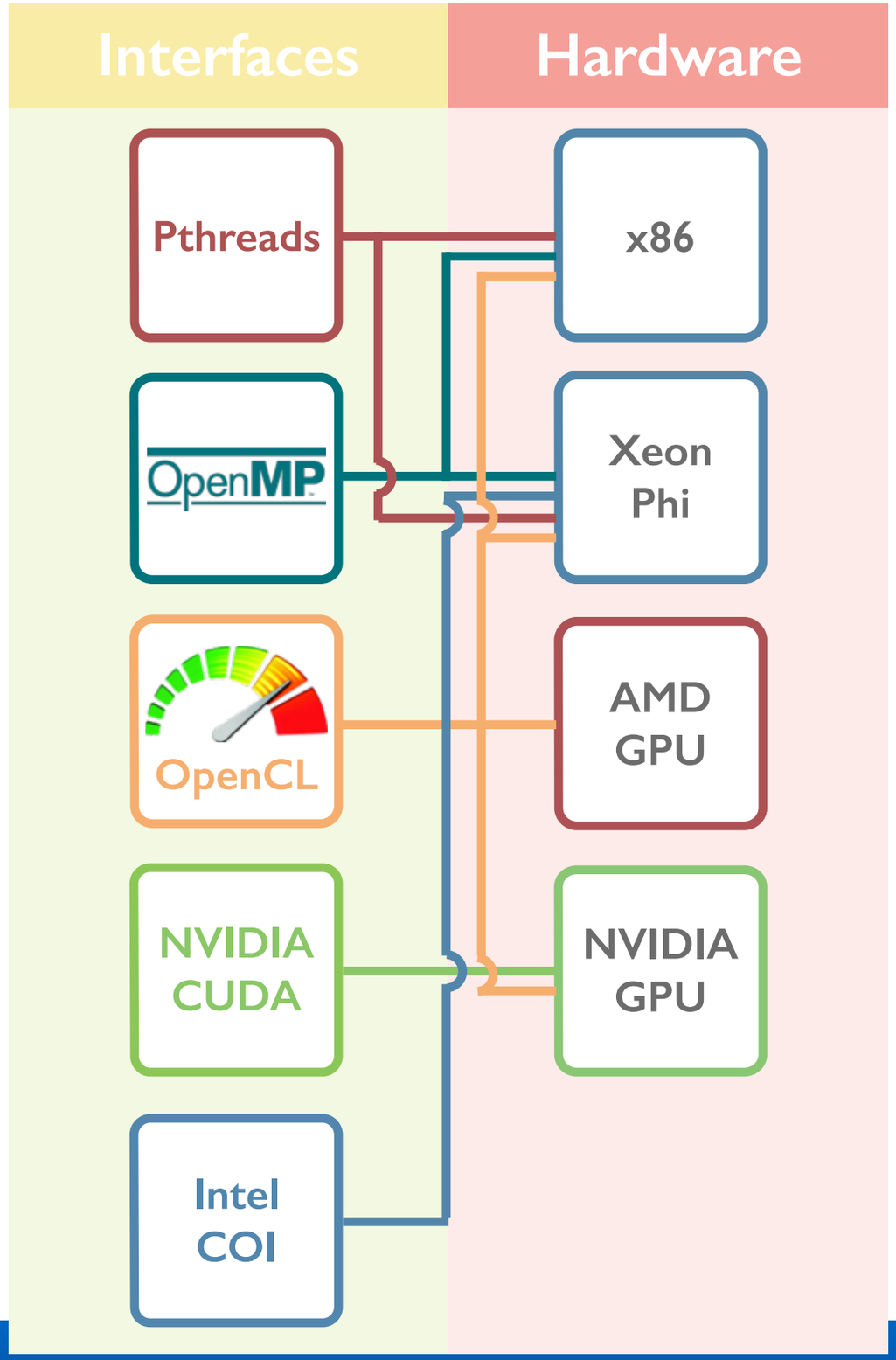
API	Type	Front-ends	Kernel	Back-ends
Kokkos	ND arrays	C++	Custom	CUDA & OpenMP
VexCL	Vector class	C++	-	CUDA & OpenCL
RAJA	Library	C++	C++ Lambdas	CUDA, OpenMP, OpenACC
OCCA2	API, Source-to-source, Kernel Languages	C,C++,C#, F90, Python,MATLAB, Julia	OpenCL, CUDA,& custom unified kernel language	CUDA, 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

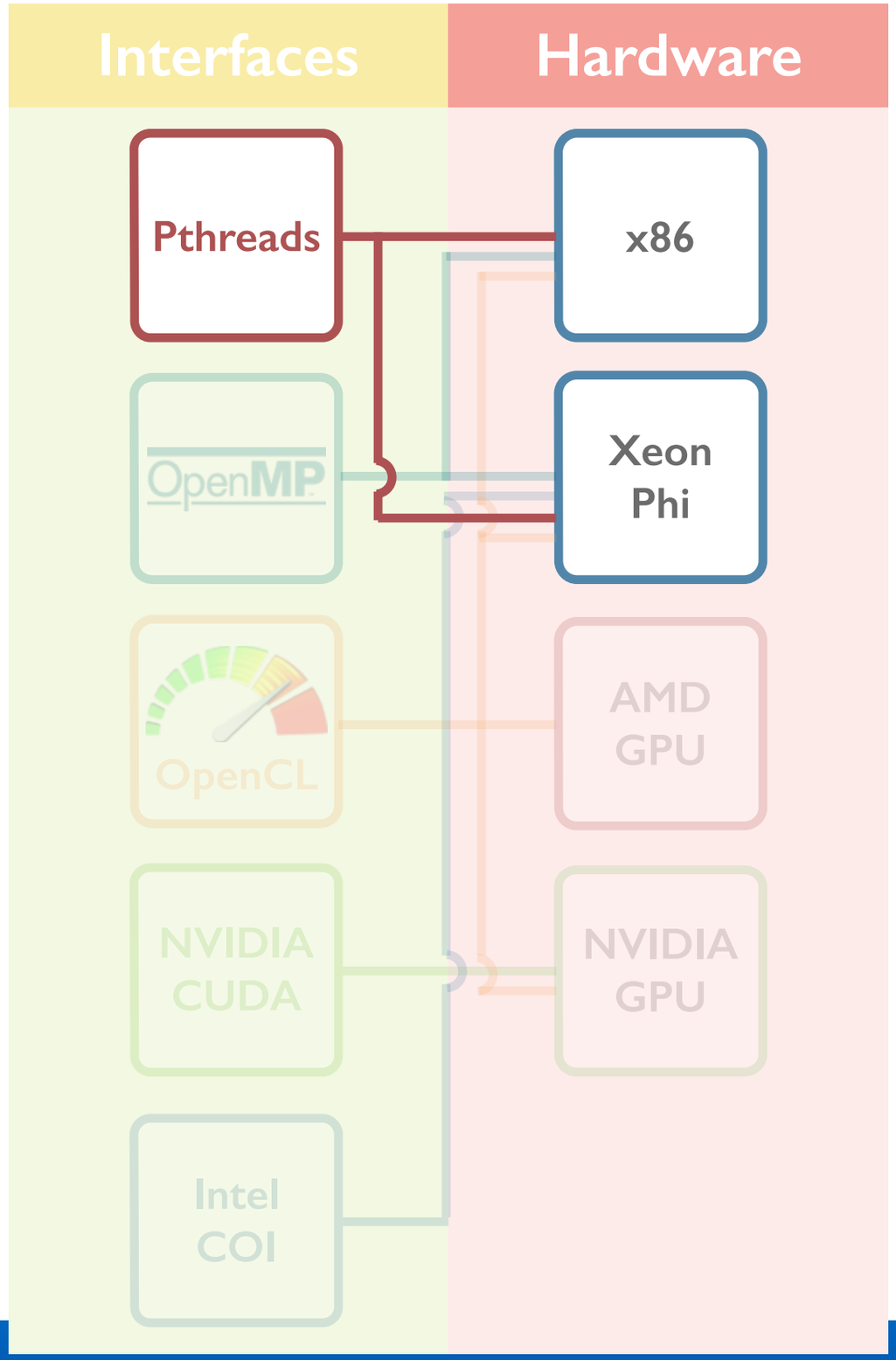
Compiler

OCCA emphasis: lightweight and extensible.

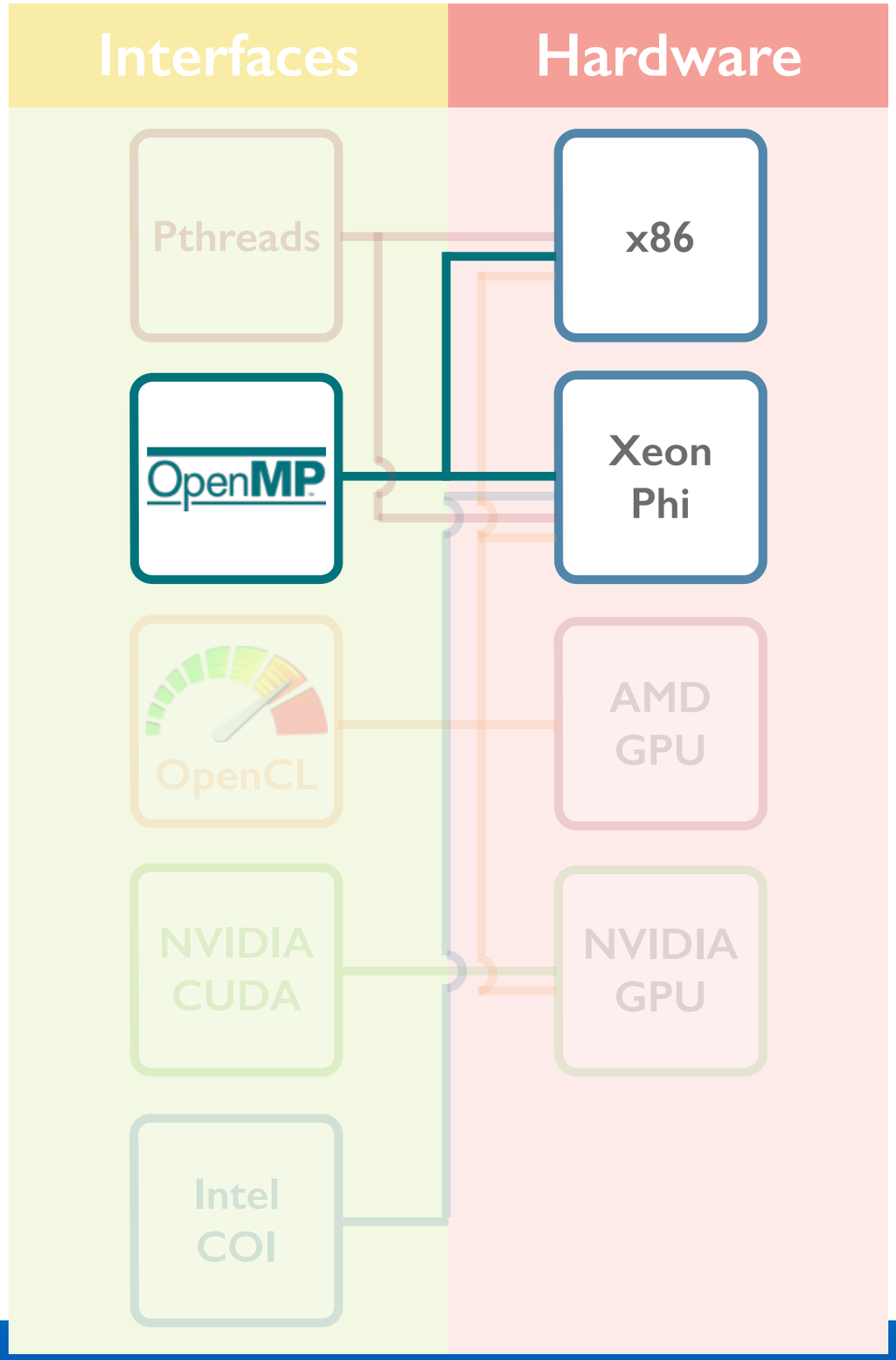
OCCA2: supported backends



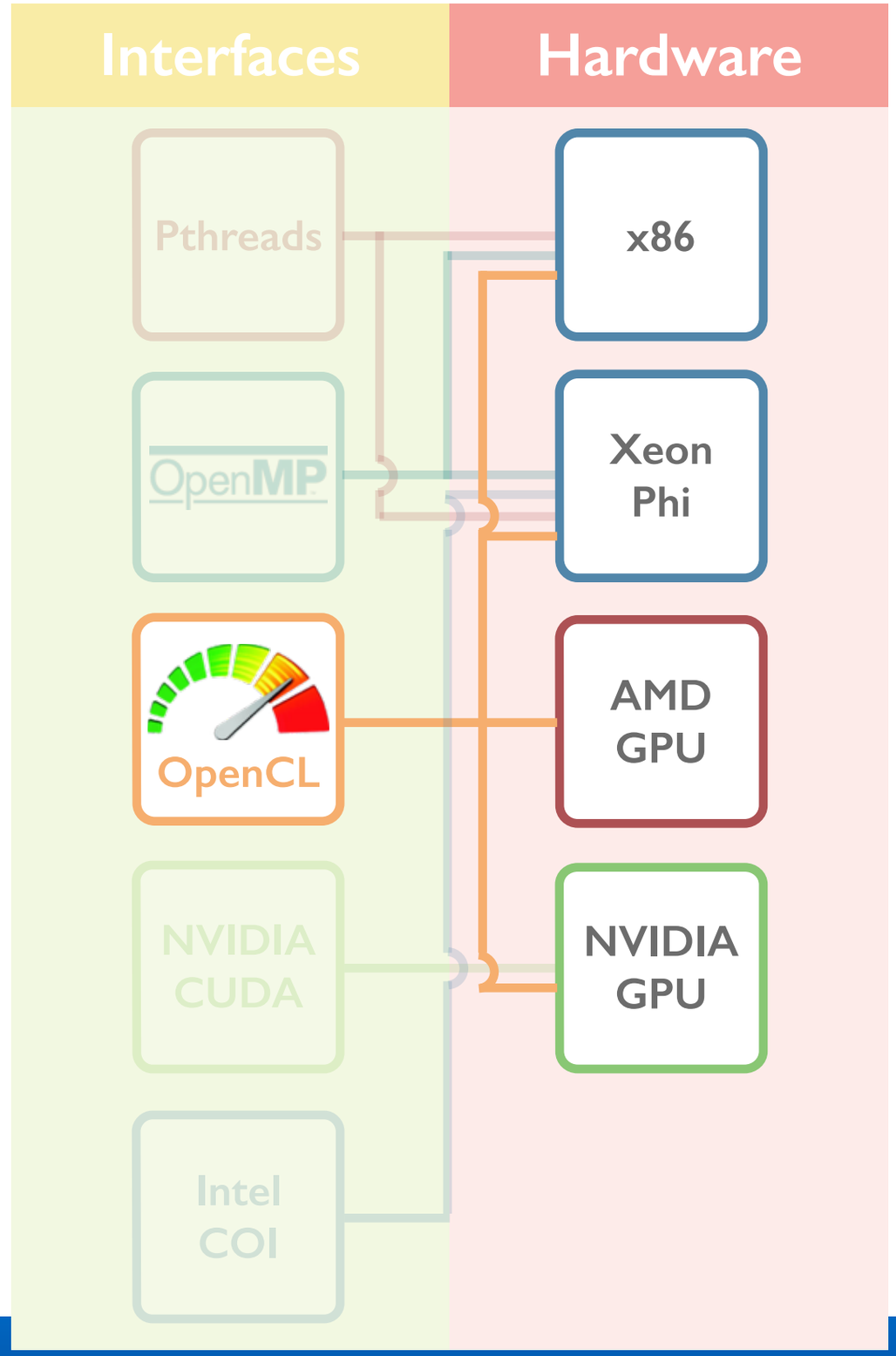
OCCA2: supported backends



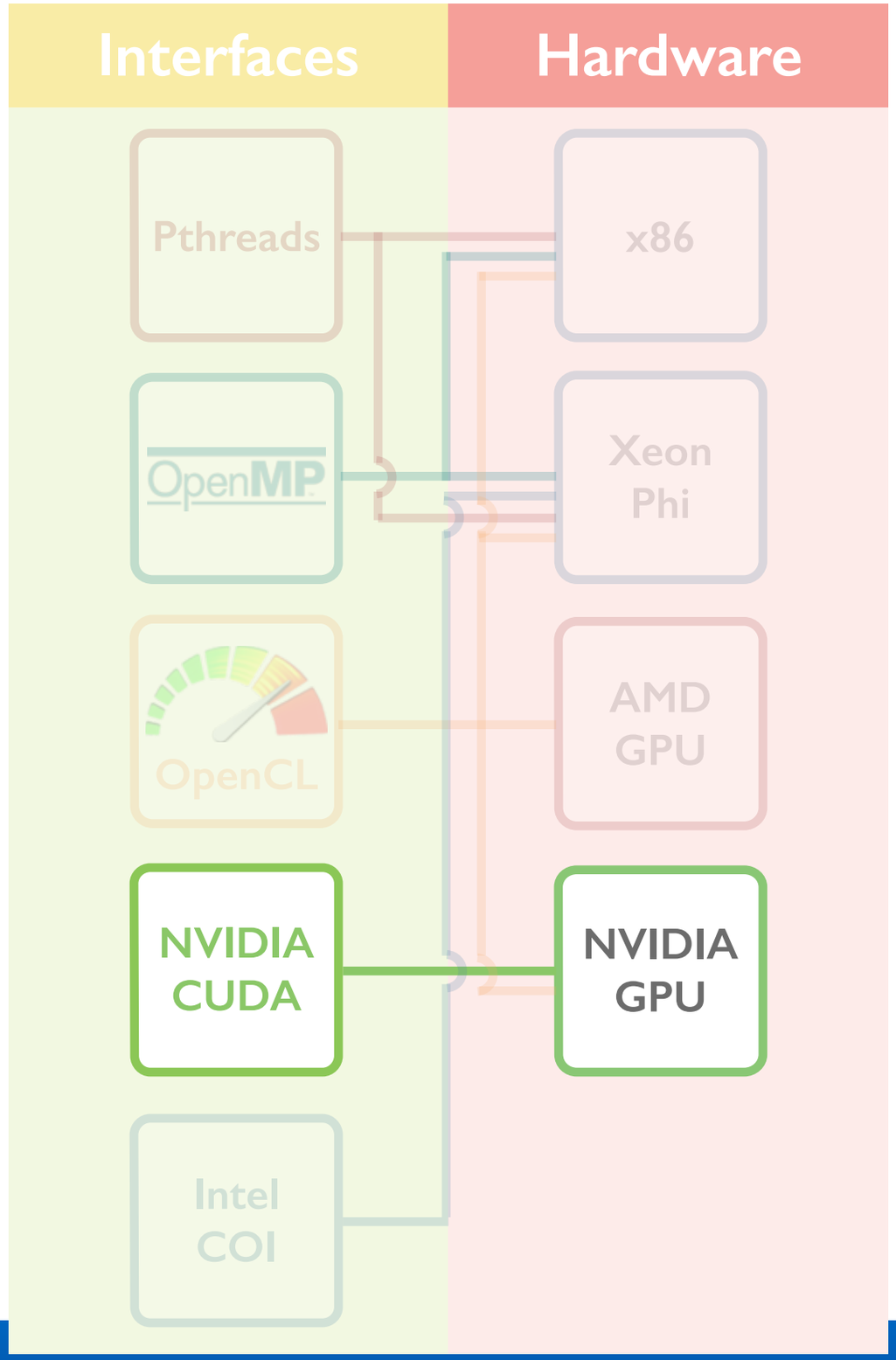
OCCA2: supported backends



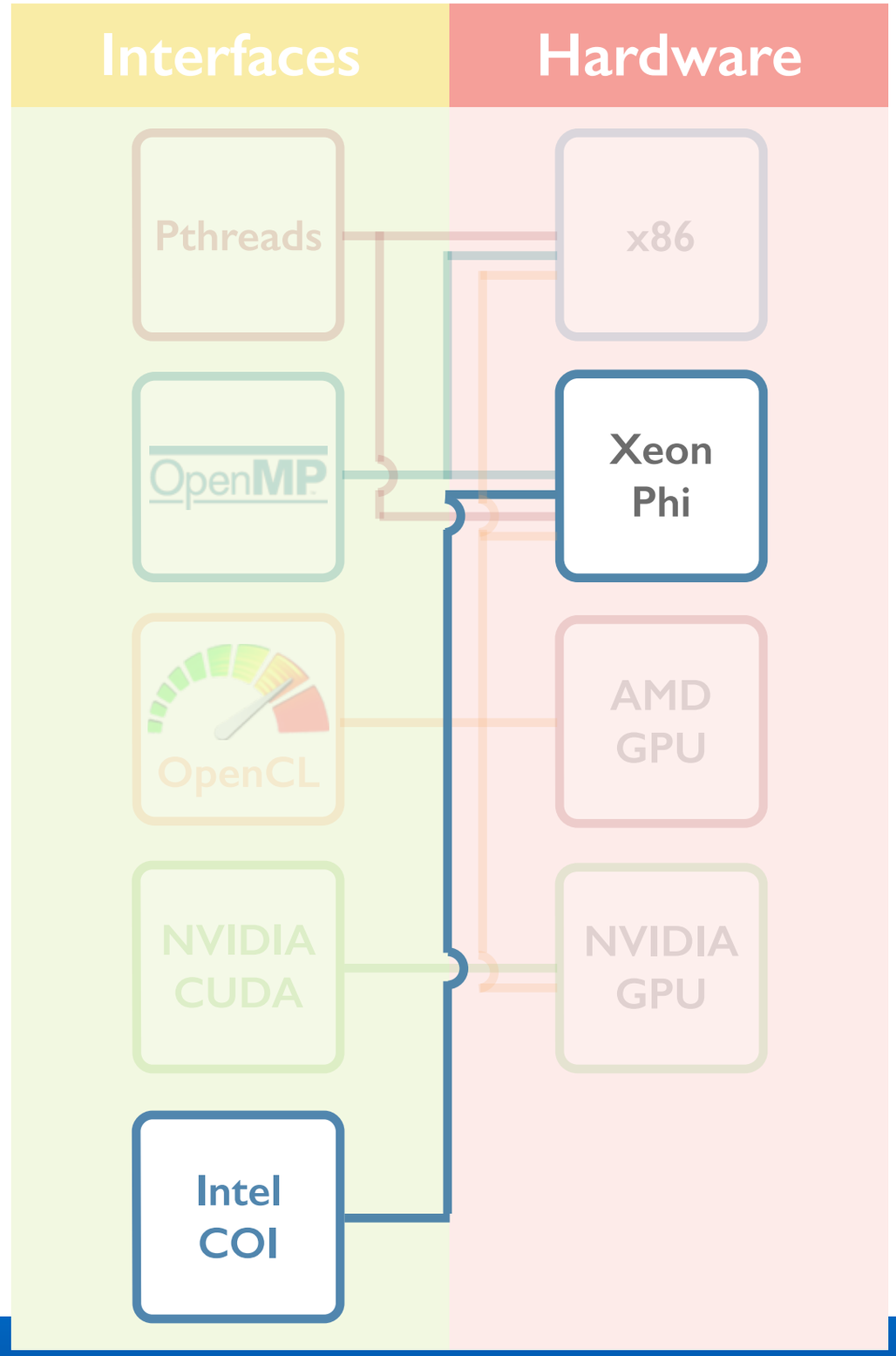
OCCA2: supported backends



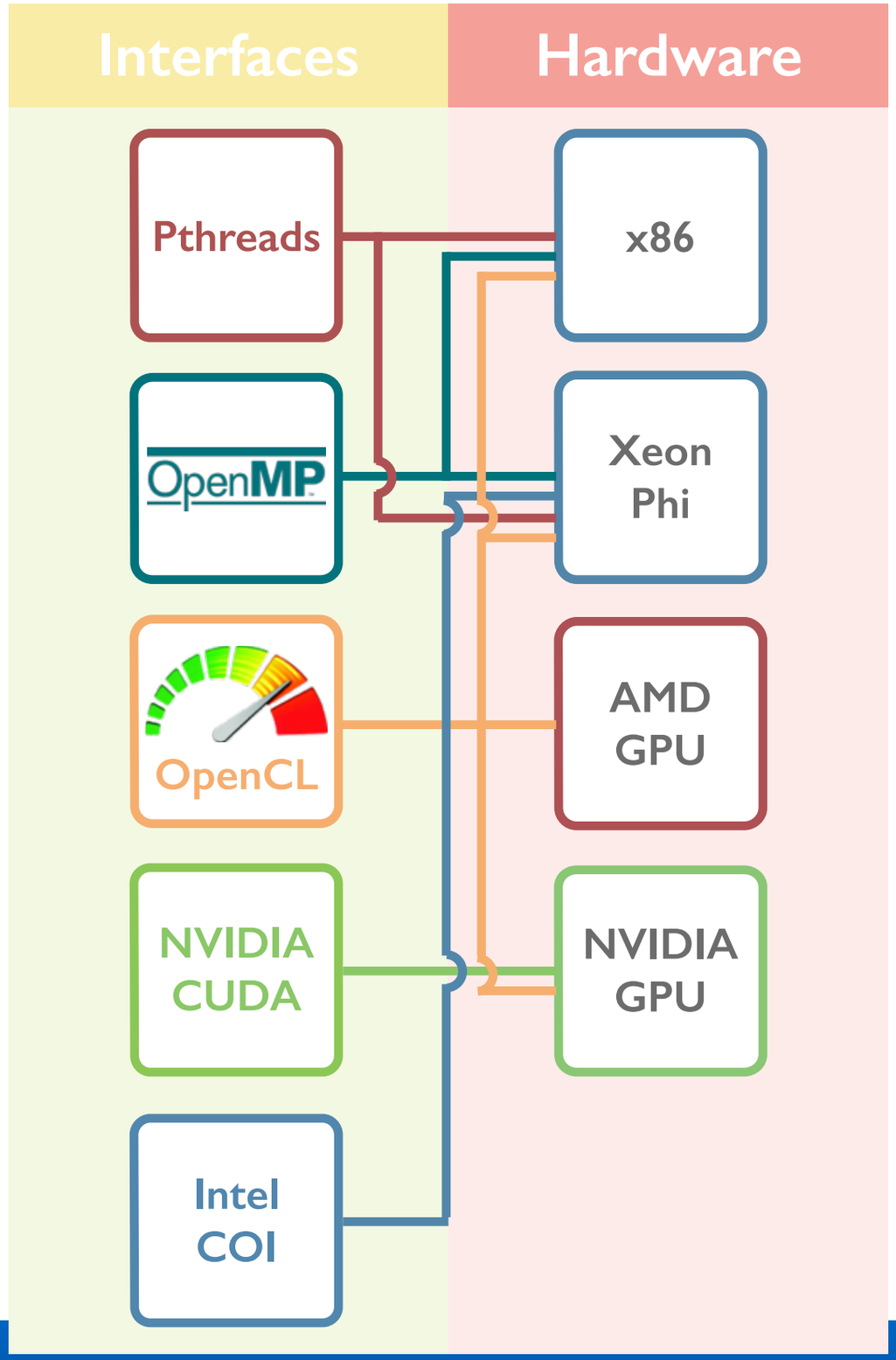
OCCA2: supported backends



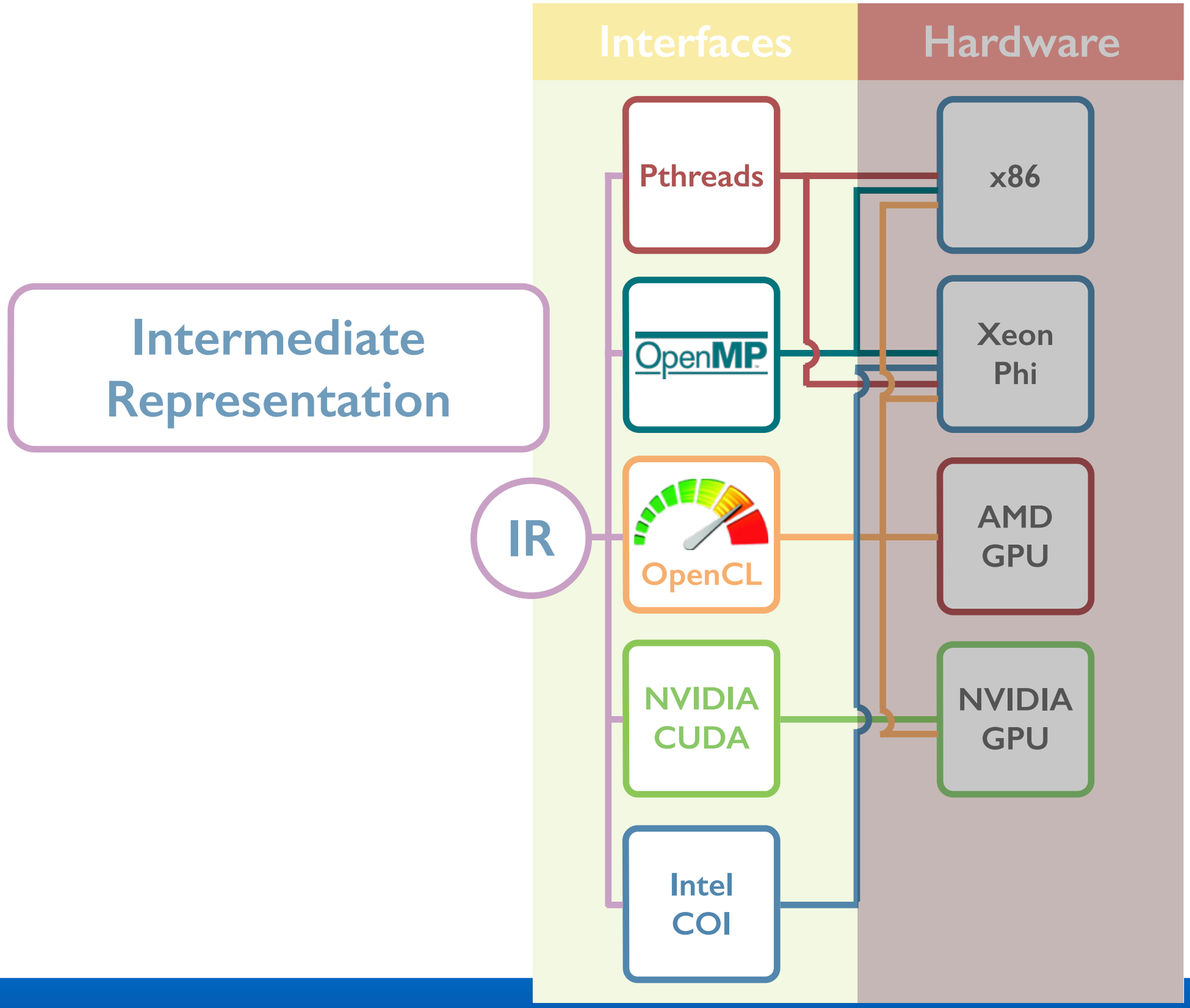
OCCA2: supported backends



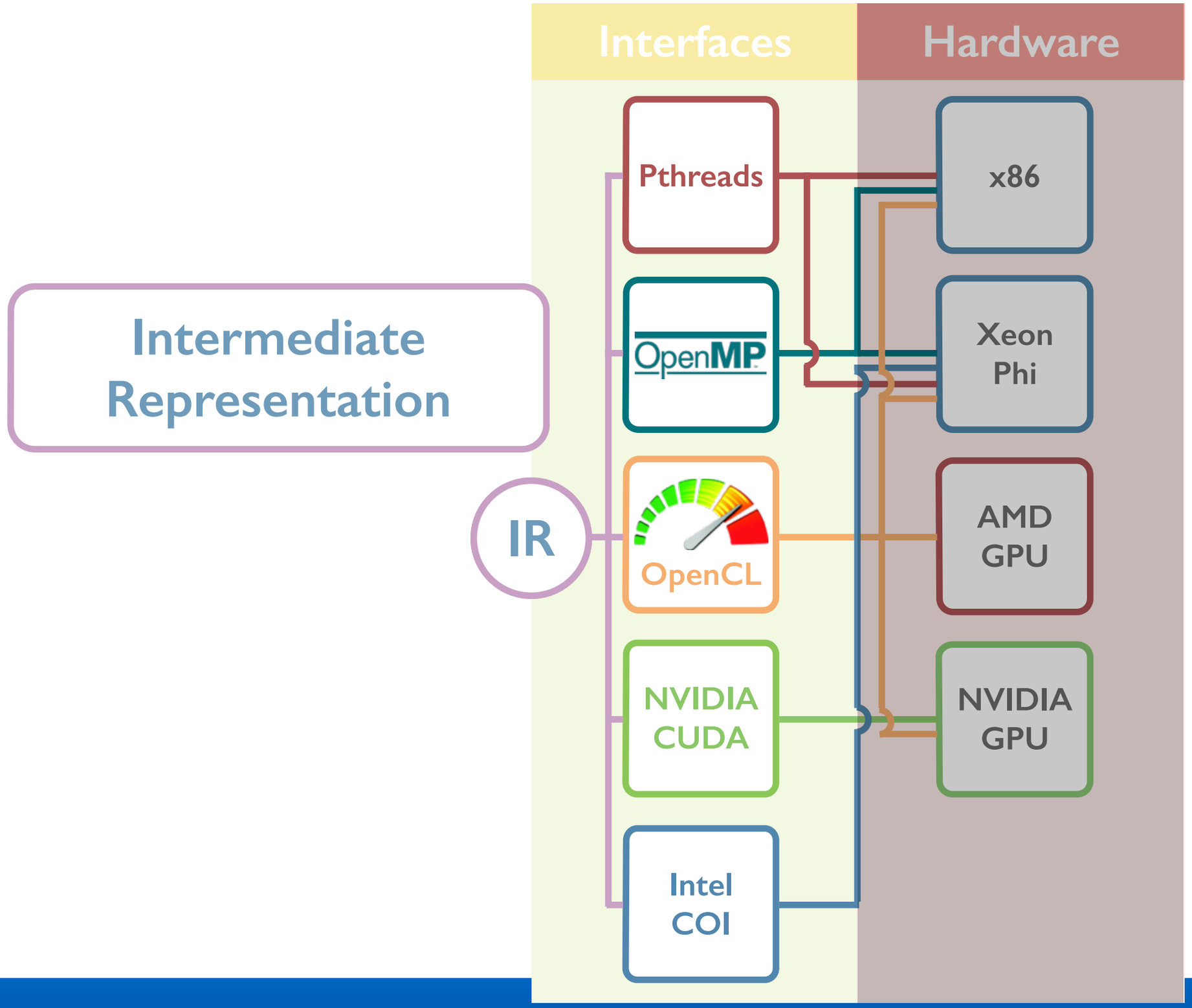
OCCA2: intermediate representation



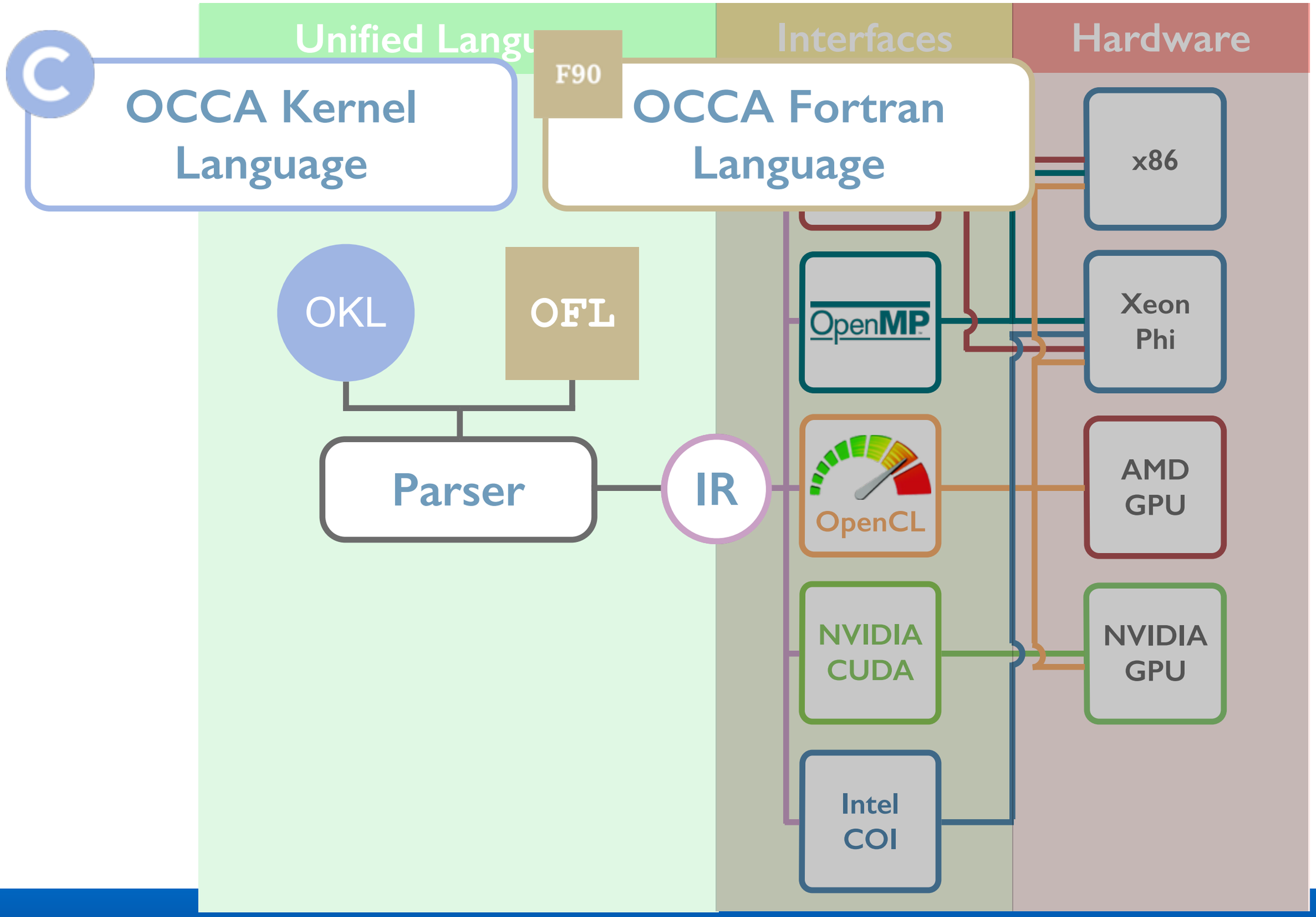
OCCA2: intermediate representation



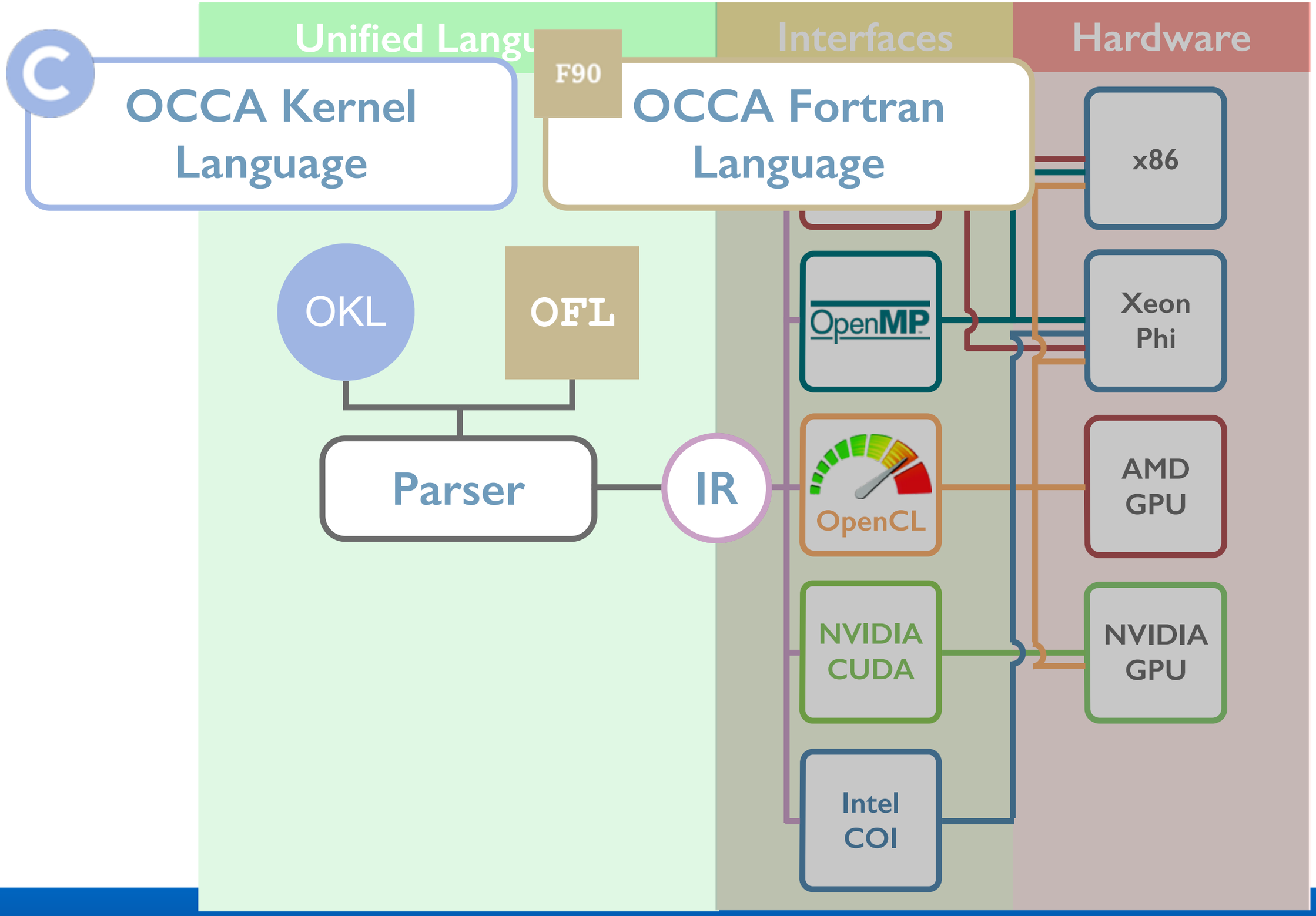
OCCA2: kernel languages



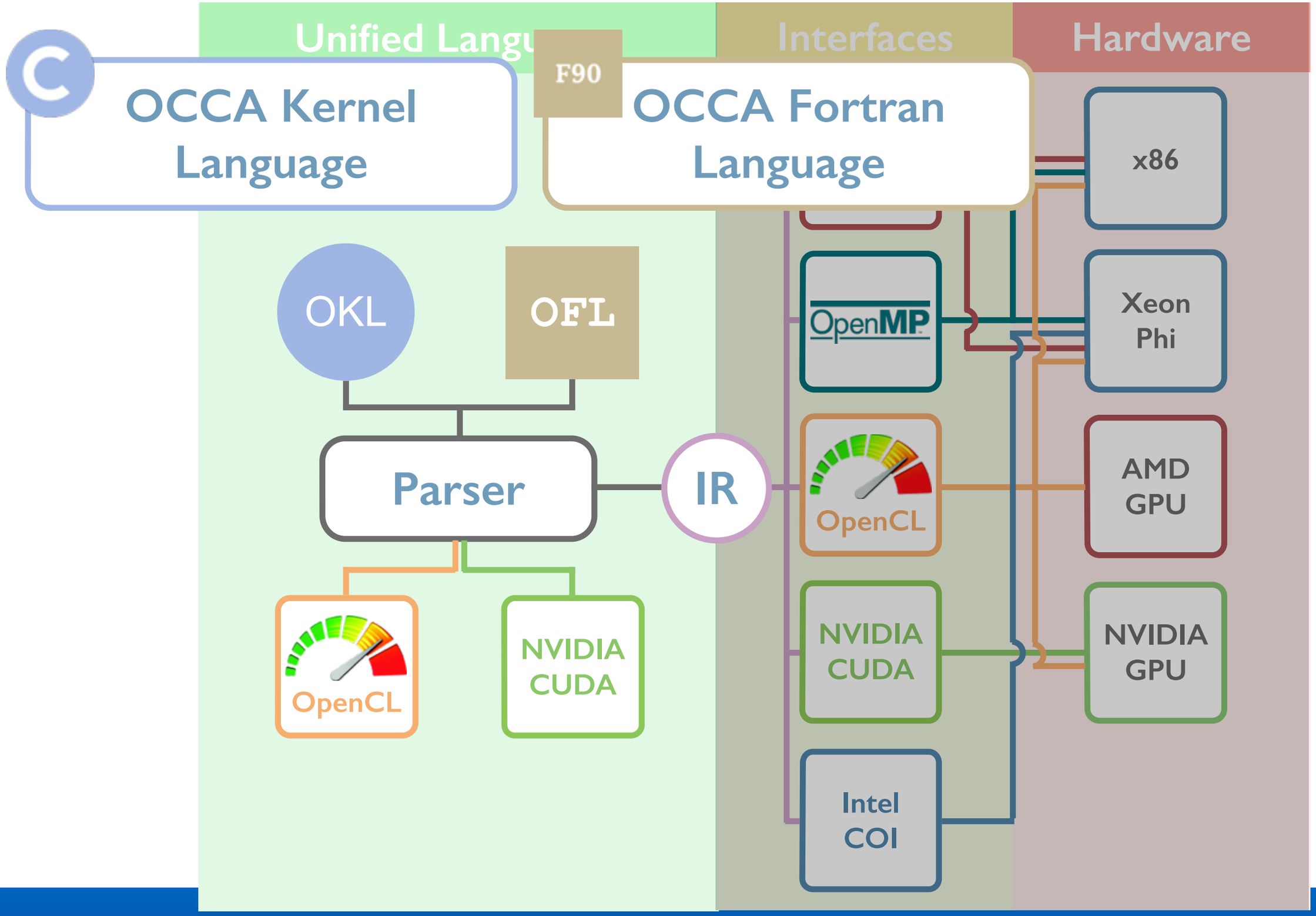
OCCA2: kernel languages



OCCA2: Kernel Languages

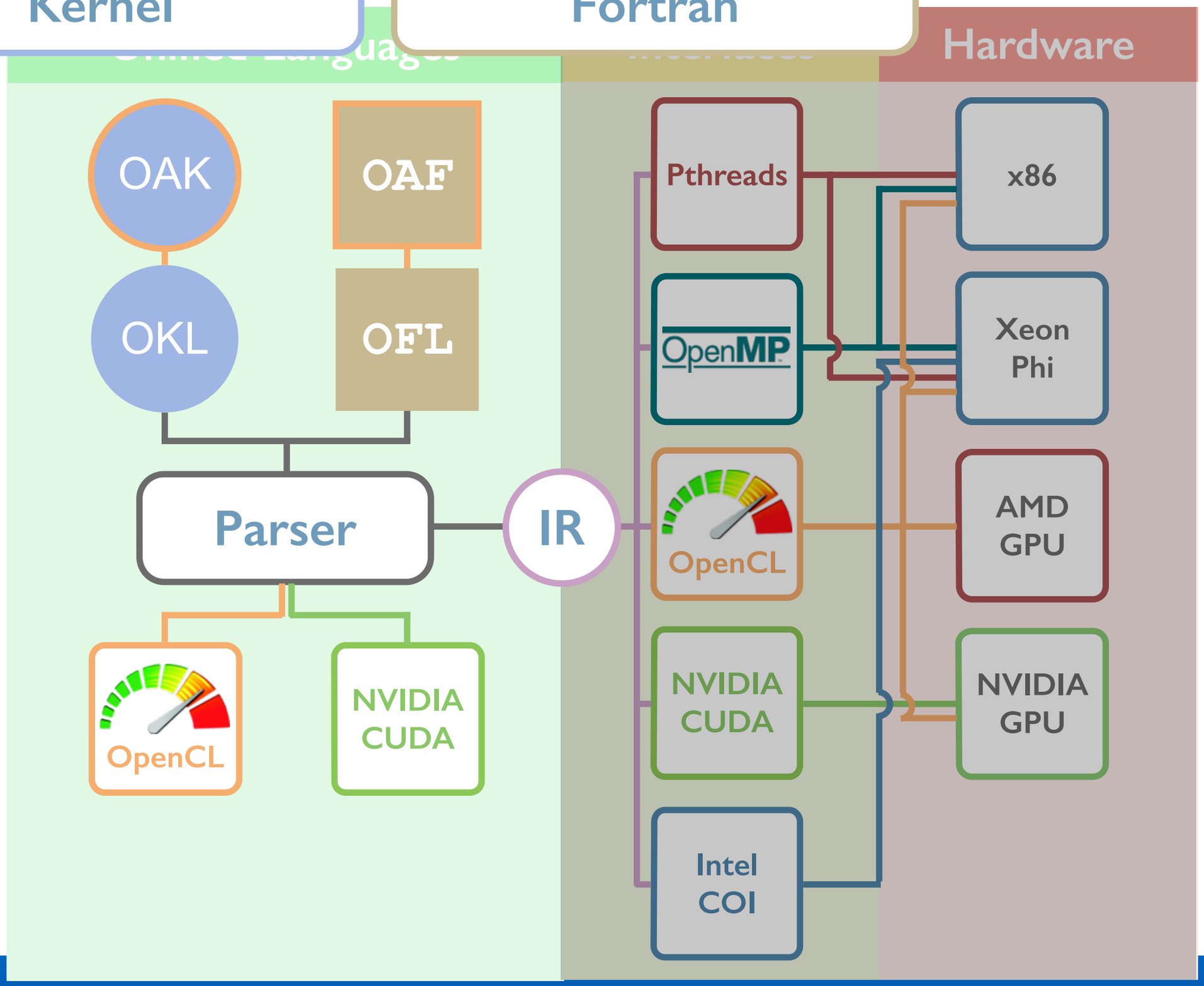


OCCA2: Kernel Languages



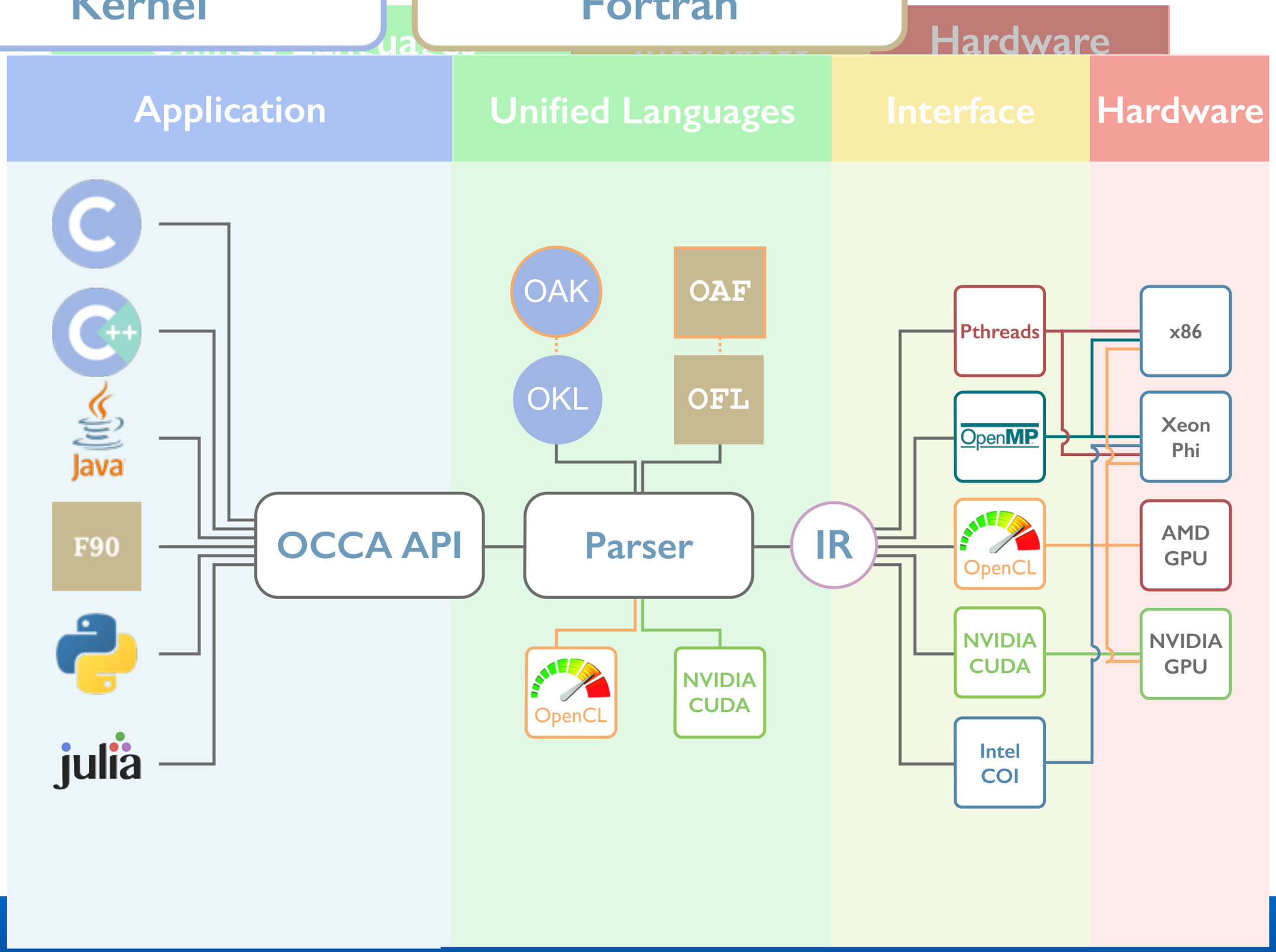
OCCA Automagic Kernel

OCCA Automagic Fortran

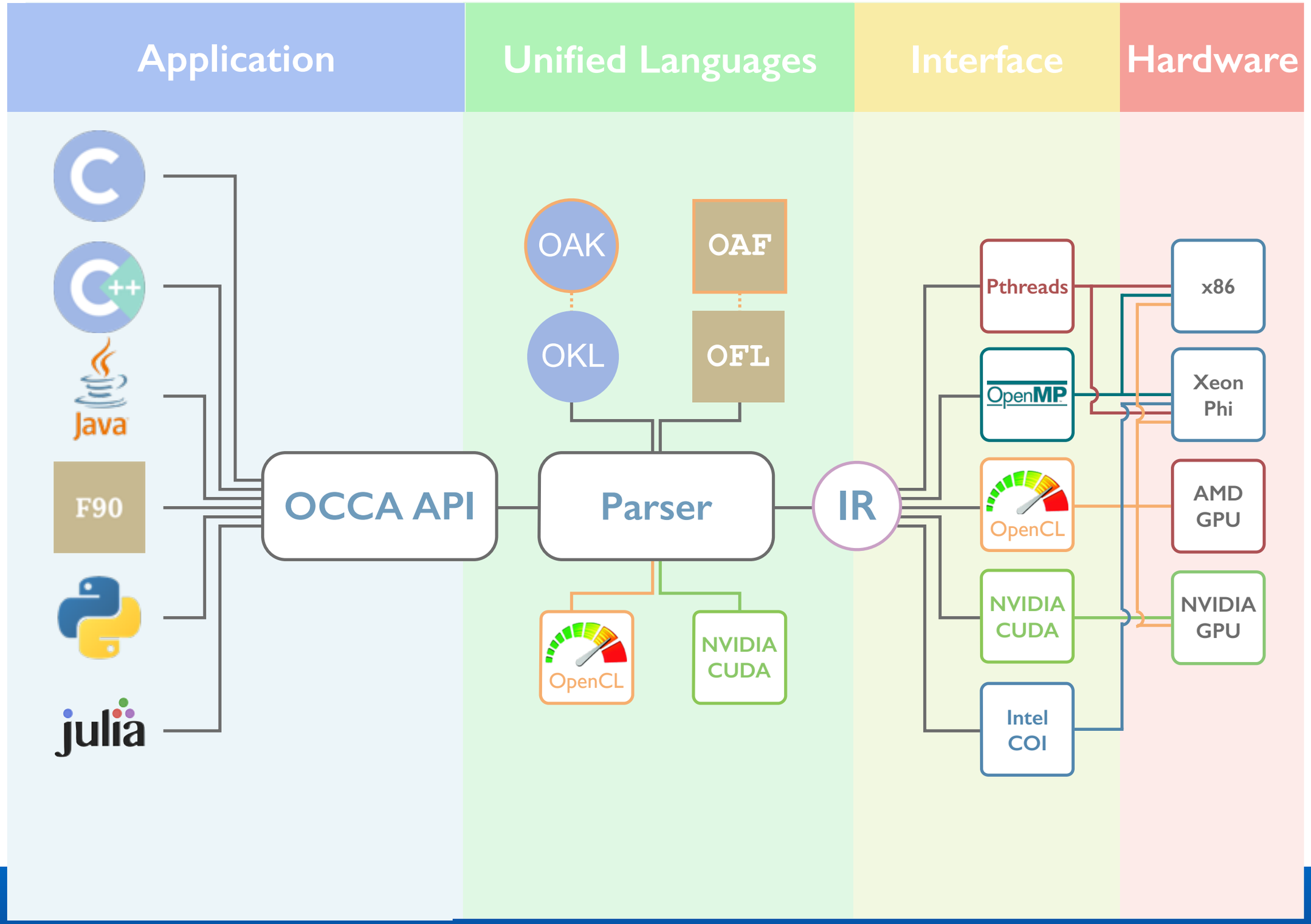


OCCA Automagic Kernel

OCCA Automagic Fortran



OCCA2: automatic code generation



libocca.org

Completely open source

Motivation: speed and usability

Performance

- Logically similar kernels differ in performance:
  - GCC v. ICPC
  - OpenCL v. CUDA
- Naively porting OpenMP to CUDA or OpenCL will likely yield low performance

Programmability

- Multiple approaches have been taken
- Expose parallel paradigm … without introducing an exotic programming model

Portability Approaches

Directive Approach

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

Source-to-Source Approach

- Compiler tools can be used to translate across specifications/languages
- Performance is not always portable
- Maintenance of original and translated codes

Wrapper Approach

- Create a tailored library with optimized functions
- Restricted to a pre-canned set of operations
- Flexibility comes from functors/lambdas at compile-time

Portability Approaches: compiler 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)
- There are compilers which support the 1.0 specifications
- OpenACC 2.0 introduces support for inlined functions

- OpenMP has been around for a while (1997)
- OpenMP 4.0 specifications (2013) includes accelerator support
- Few compilers (ROSE) support parts of the 4.0 specifications

OpenMP

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

(Reyes, R., Lopez, I., Fumero, J. and de Sande, F. 2012),
(Grewe, D., Wang, Z. and O'Boyle, M. F. 2013)

Portability Approaches: directives

Directive Approach

- OpenACC and OpenMP begin to resemble an API rather than code decorations
- Not centralized anymore (intrusive [#pragma]'s) for optimizations

OpenACC

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

OpenACC

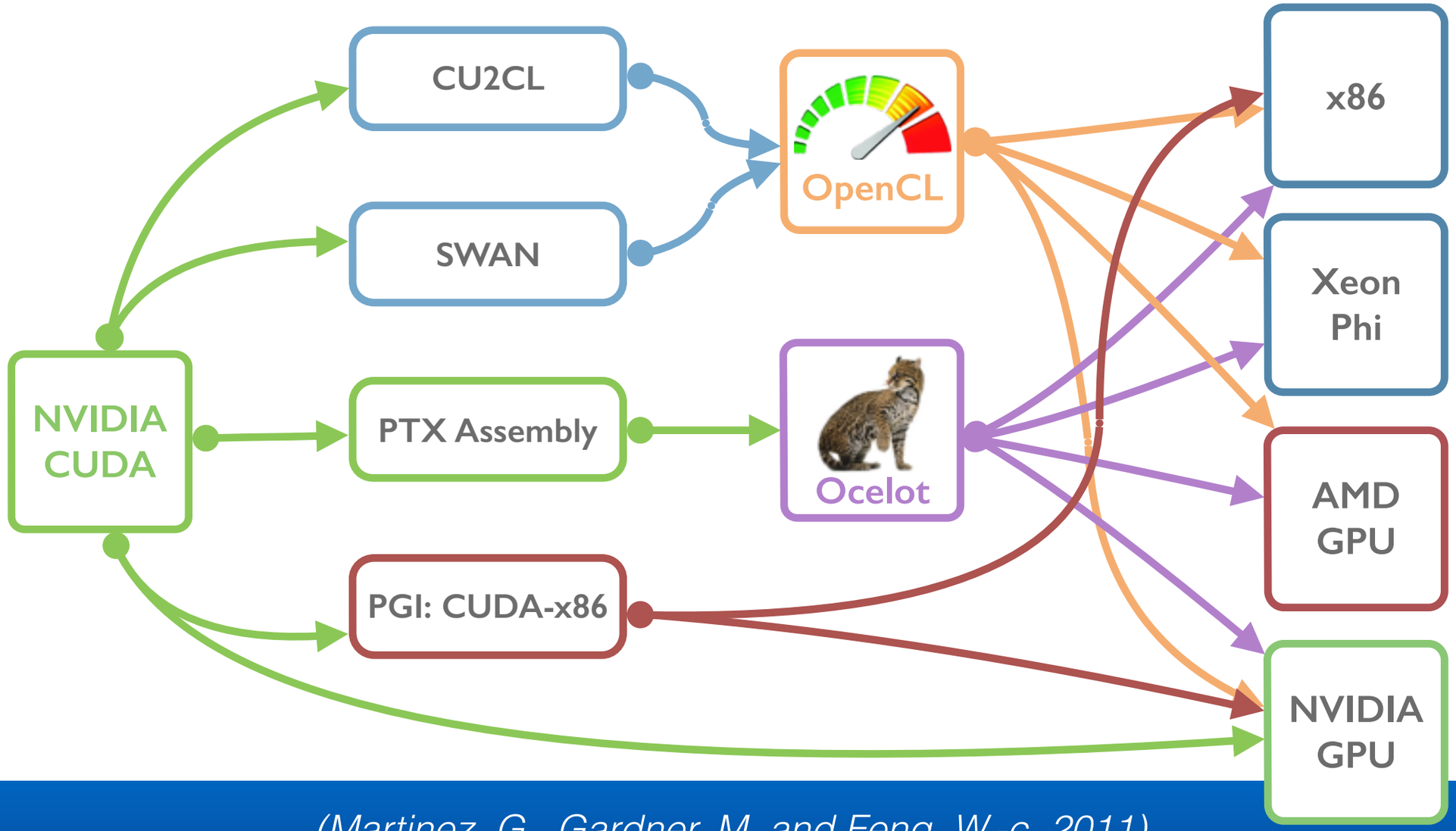
```
class Matrix {
  double *v;
  int len
  Matrix(int n) {
    len = n;
    v = new double[len];
    #pragma acc enter data create(v[0:len])
  }
  ~Matrix() {
    #pragma acc exit data delete(v[0:len])
    delete[] v;
  }
};
```

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

Portability Approaches: source conversion

Source-to-Source Approach

- CU2CL and SWAN have limited CUDA support (3.2 and 2.0 respectively)
- GPU Ocelot supports PTX from CUDA 4.2 (5.0 partially)
- PGI: CUDA-x86 appears to have been put in hiatus since 2011



(Martinez, G., Gardner, M. and Feng, W.-c. 2011)
(Harvey, M. J. and De Fabritiis, G. 2011)

Portability Approaches: libraries

Wrapper Approach

- Create a tailored library with optimized functions
- Restricted to a set of tools with flexibility from functors/lambdas

- C++ library masking OpenMP, Intel's TBB and CUDA for x86 processors and NVIDIA GPUs
- Vector library, such as the standard template library (STL)

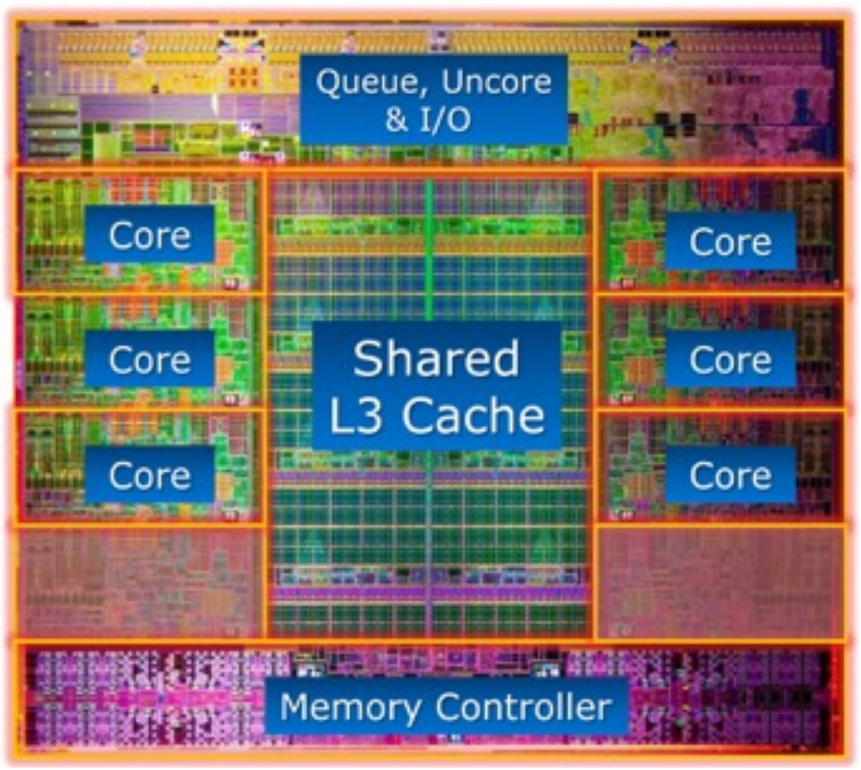
- Kokkos is from Sandia National Laboratories
- C++ vector library with linear algebra routines
- Uses OpenMP and CUDA for x86 and NVIDIA GPU support

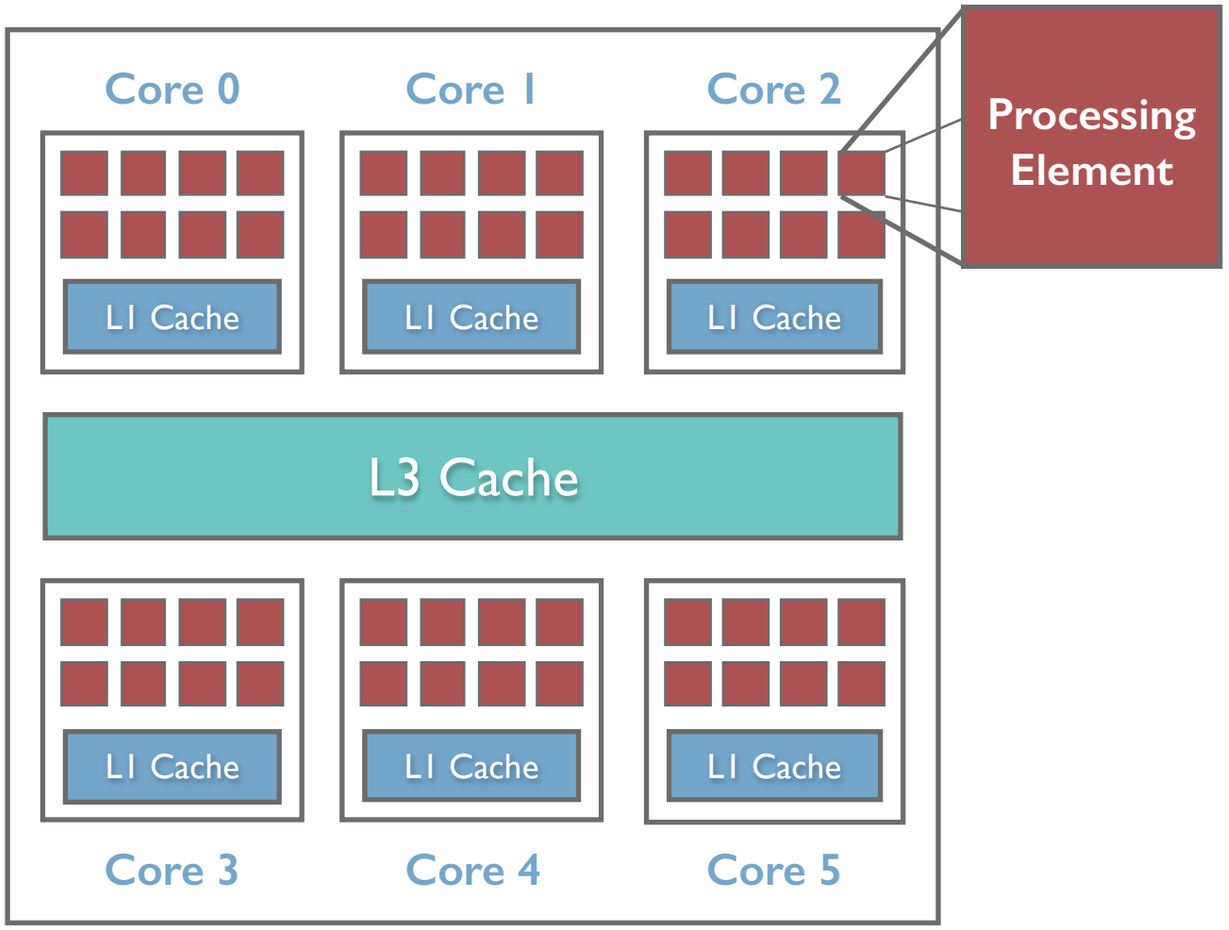
SkePU

- C++ template library
- Uses code skeletons for map, reduce, scan, mapreduce, …
- Uses OpenMP, OpenCL and CUDA as backends

(Bell, N. and Hoberock, J. 2011), (Edwards, H. C. and Trott, C. R. 2013), (Enmyren, J. and Kessler, C. W. 2010)

CPU: Architecture



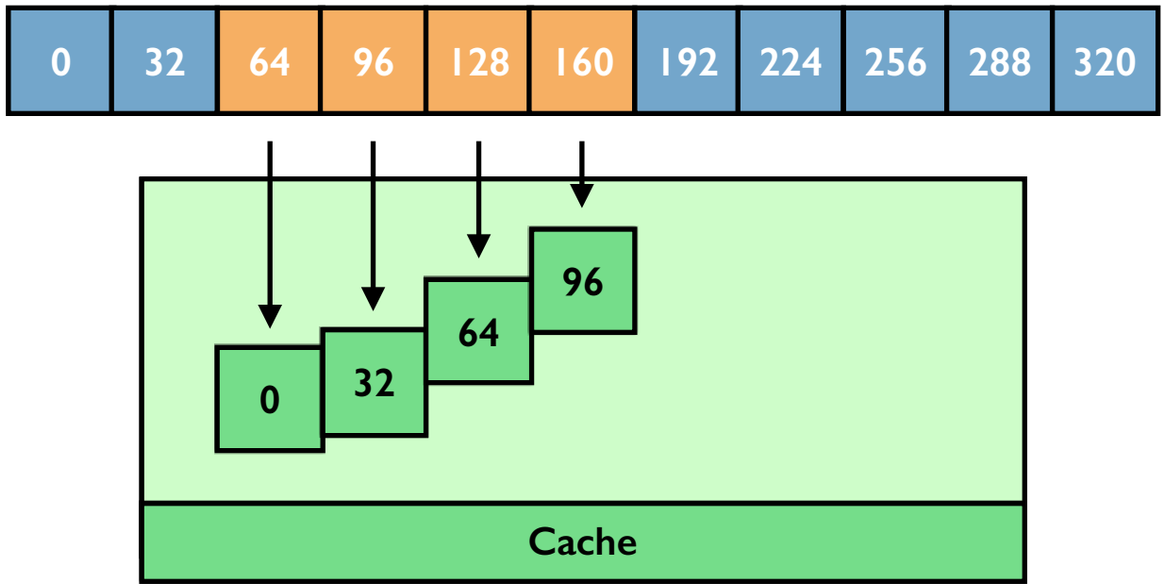


Parallelism in cores (multi-threading) and processing elements (vectorization)

CPU: Architecture

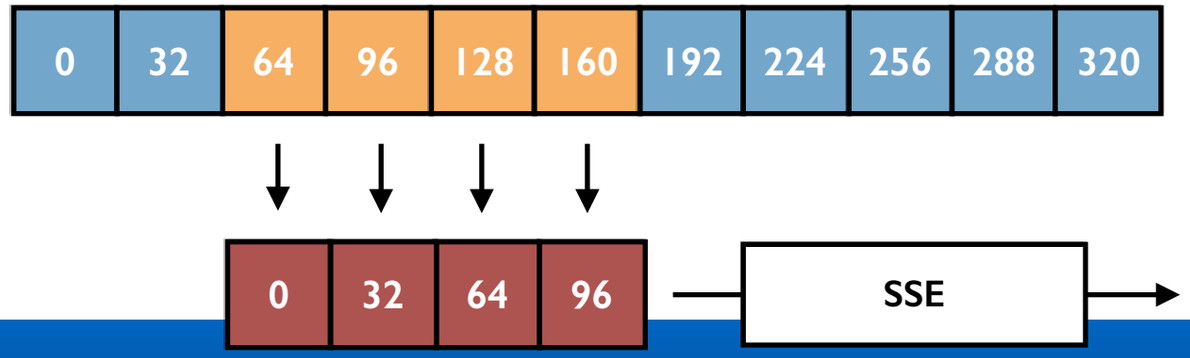
Cache

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



Vectorization

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

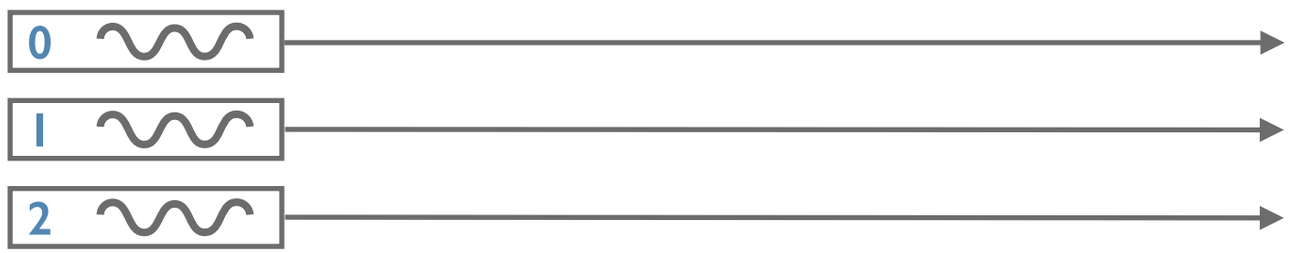


Continuous memory accesses are used for both, cache storage and vectorization

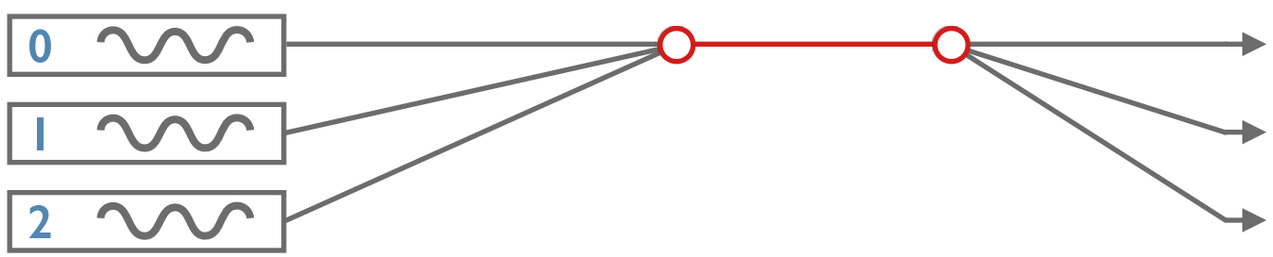
CPU: Architecture (Multi-core)

Multithreading

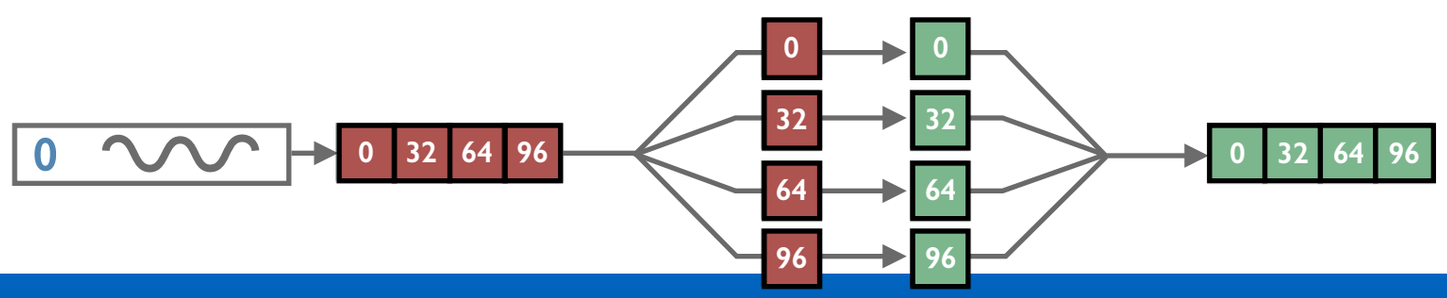
- Threads capable of fully parallelizing generic instructions (ignoring bandwidth)



- Perfect scaling … without barriers, joins, or other thread/memory-dependencies

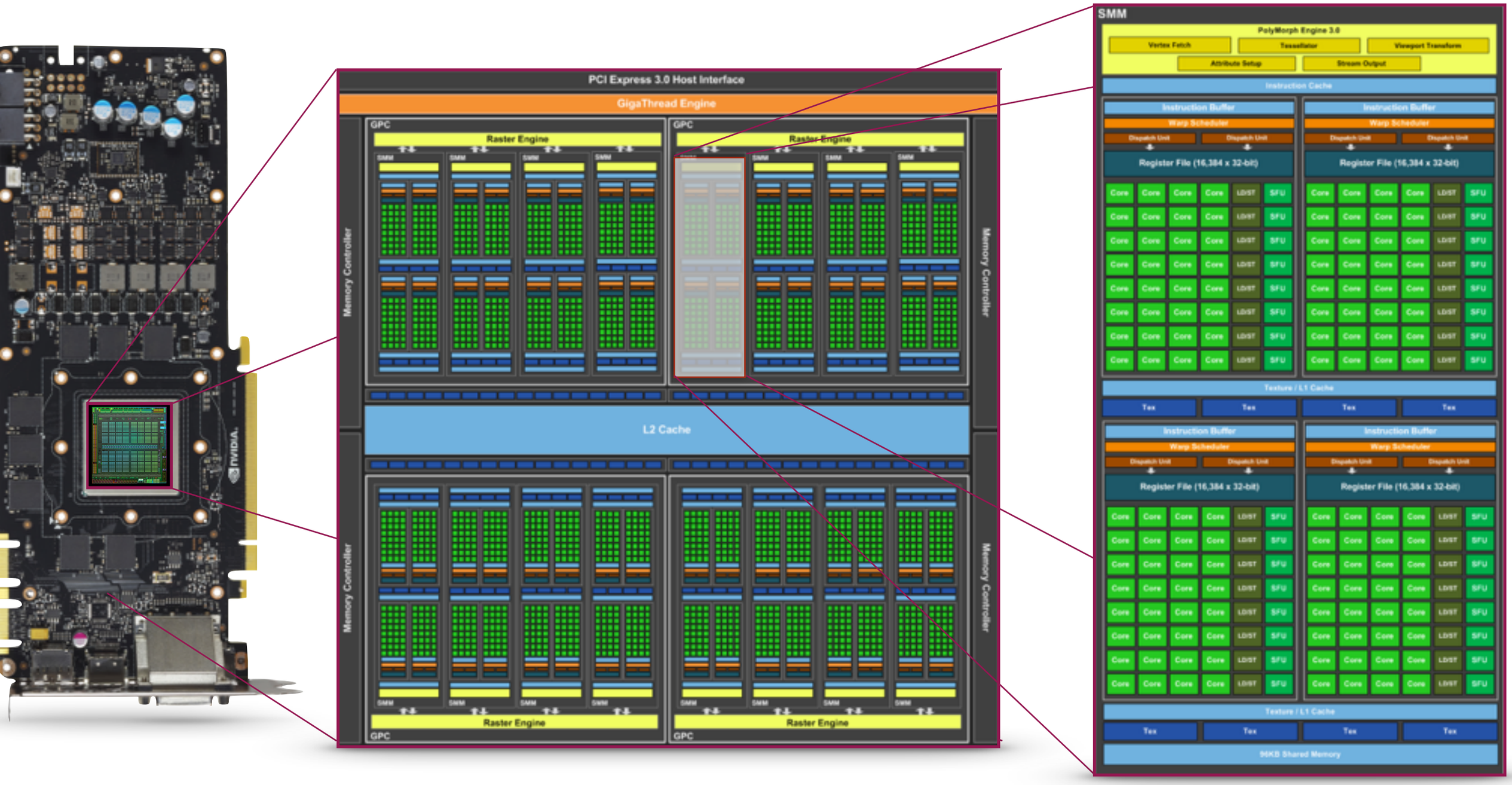


- Vectorization can be done at the thread-level



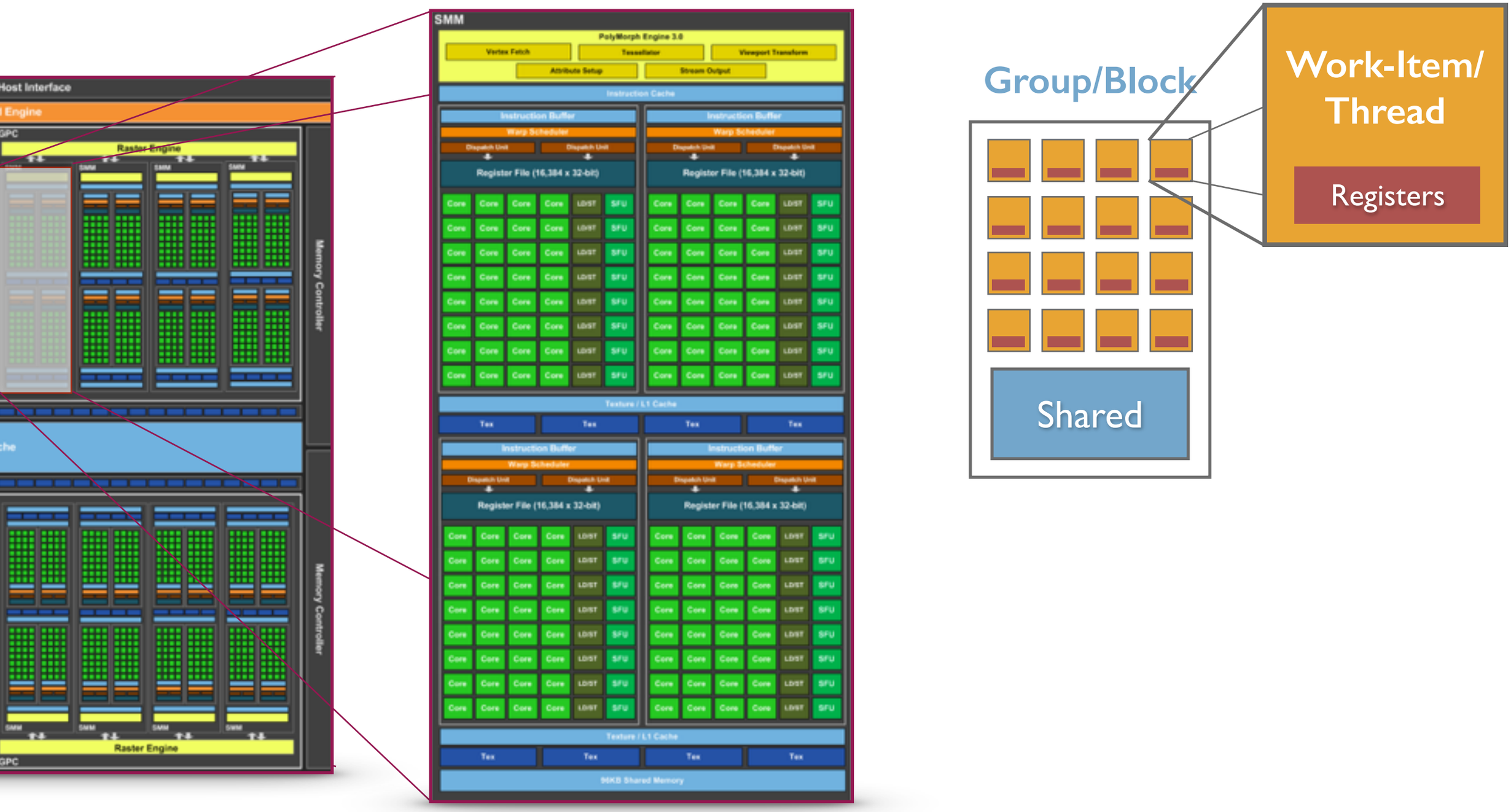
Executing parallel instructions using multithreading

GPU: Architecture



Hierarchy of groups (blocks/work-groups) of processing units (threads/work-items)

GPU: Architecture

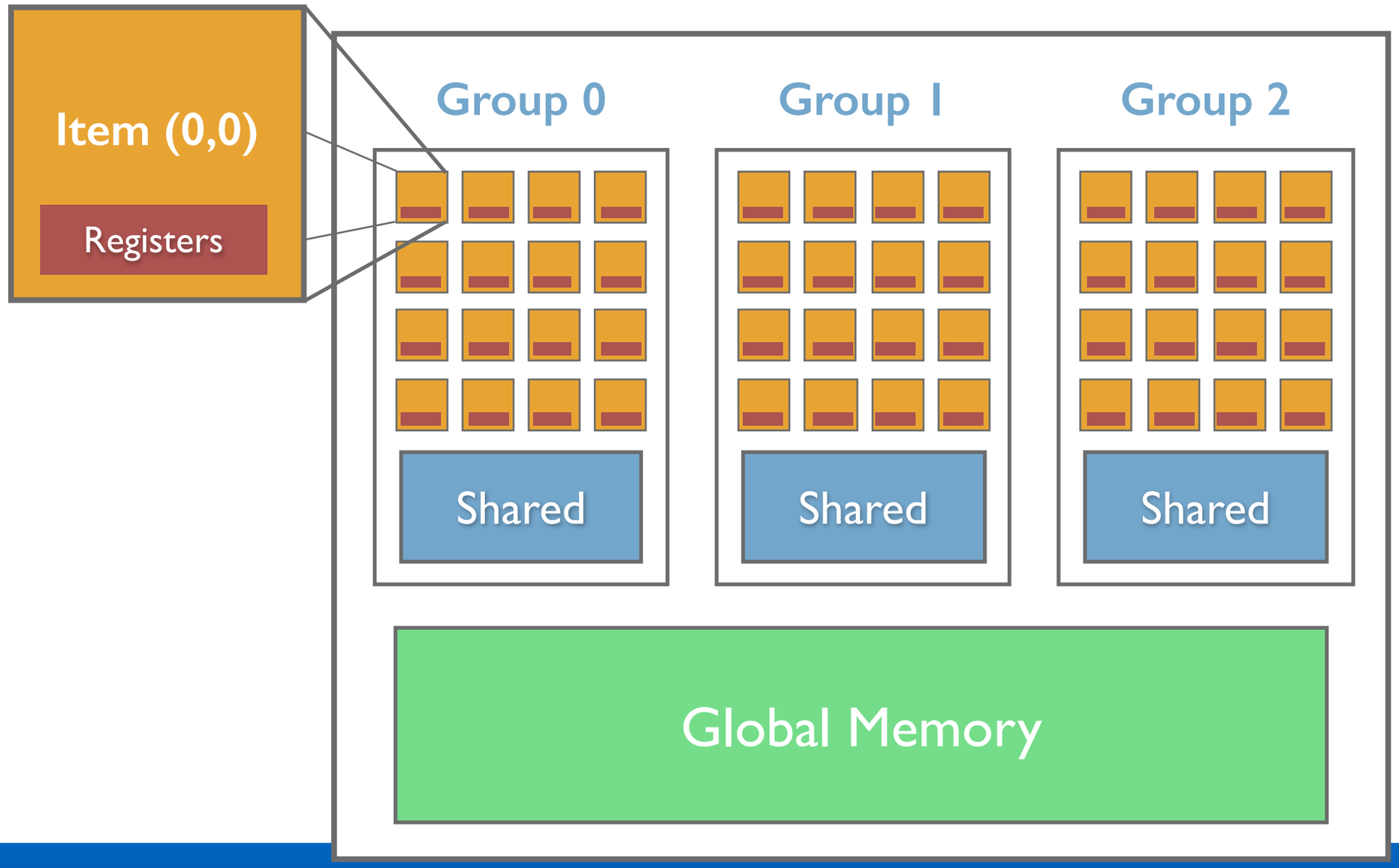


Hierarchy of groups (blocks/work-groups) of processing units (threads/work-items)

GPU: Architecture

GPU Architecture

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

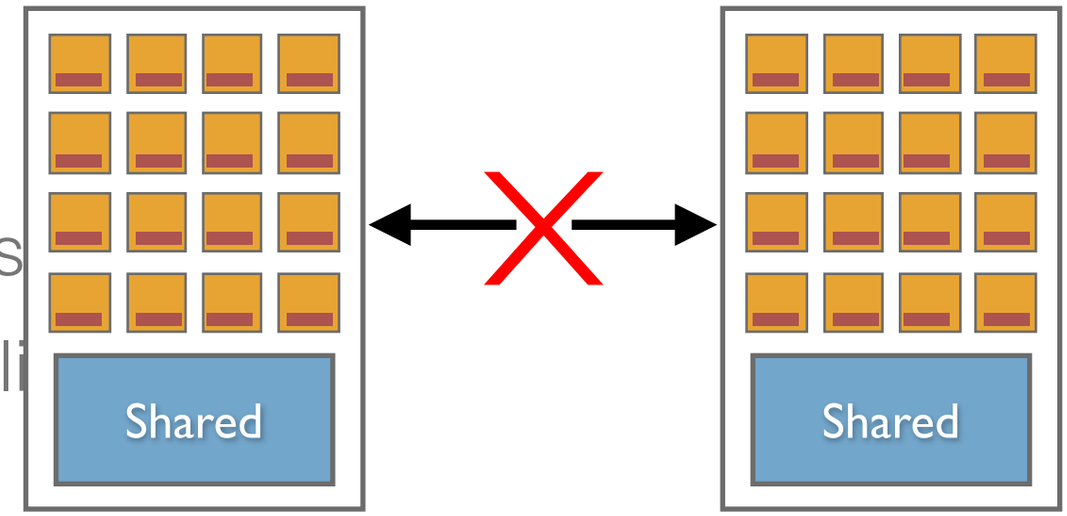


GPU Functions (Kernels) execute at the work-item level

GPU: Architecture

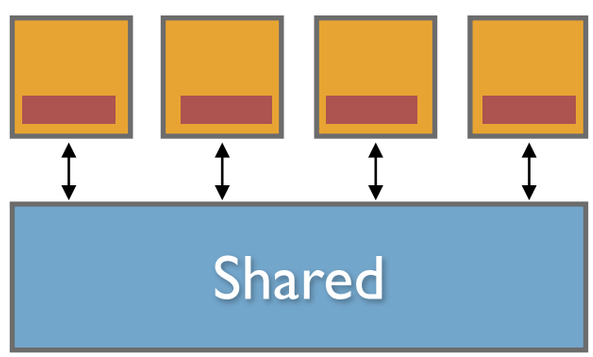
Work-groups

- Groups of work-items
- No communication between work-groups
- Designed for independent group paralleli



Work-items

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

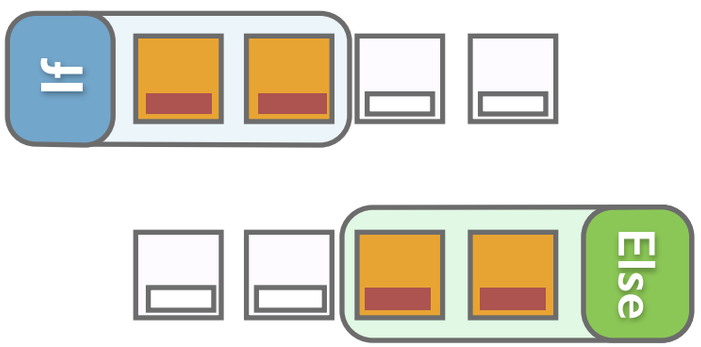


GPU: Architecture

Parallel Work-item Execution

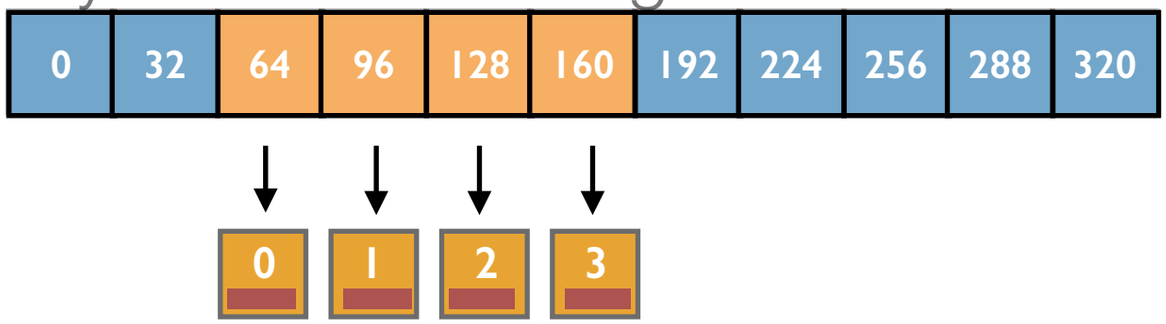
- Work-items are launched in subsets of 32 or 64
- Each set of work-items execute same instructions
- No parallel branching (in hardware-group (warp/wavefront))

```
if(threadID < 2)
  DO STUFF 1

else
  DO STUFF 2
```



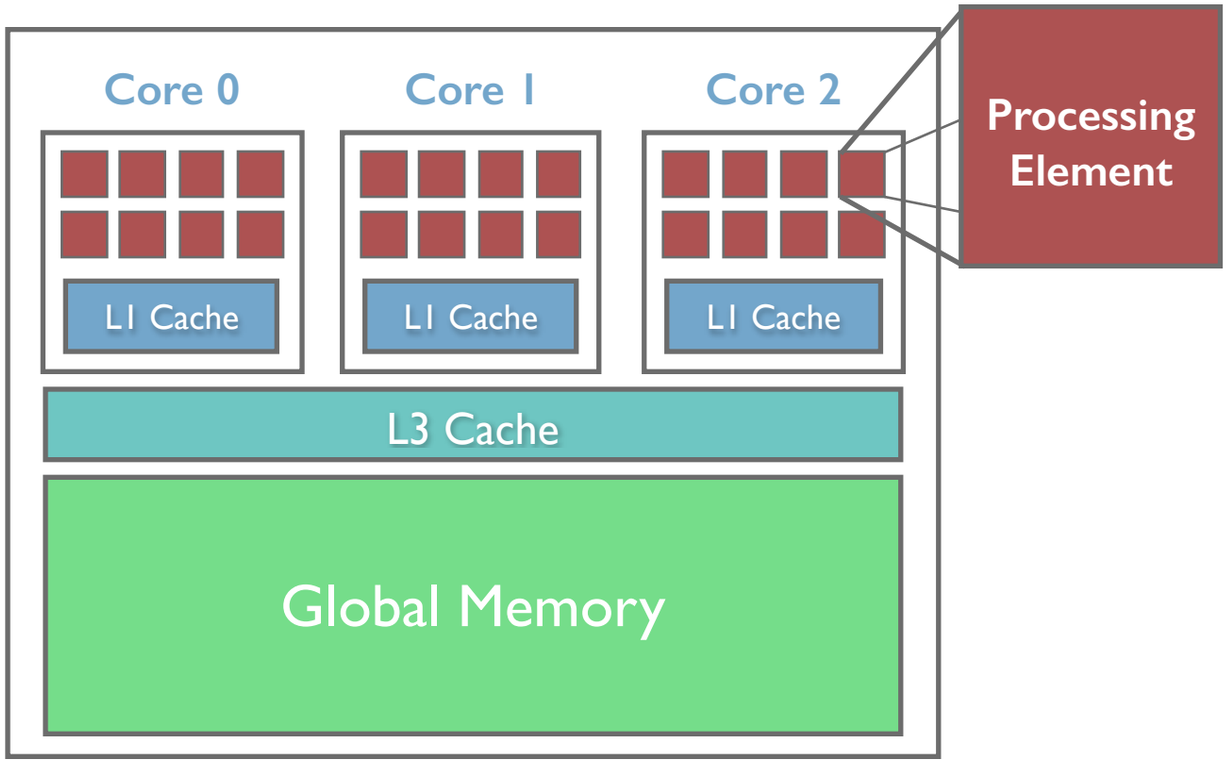
Data Transfer

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

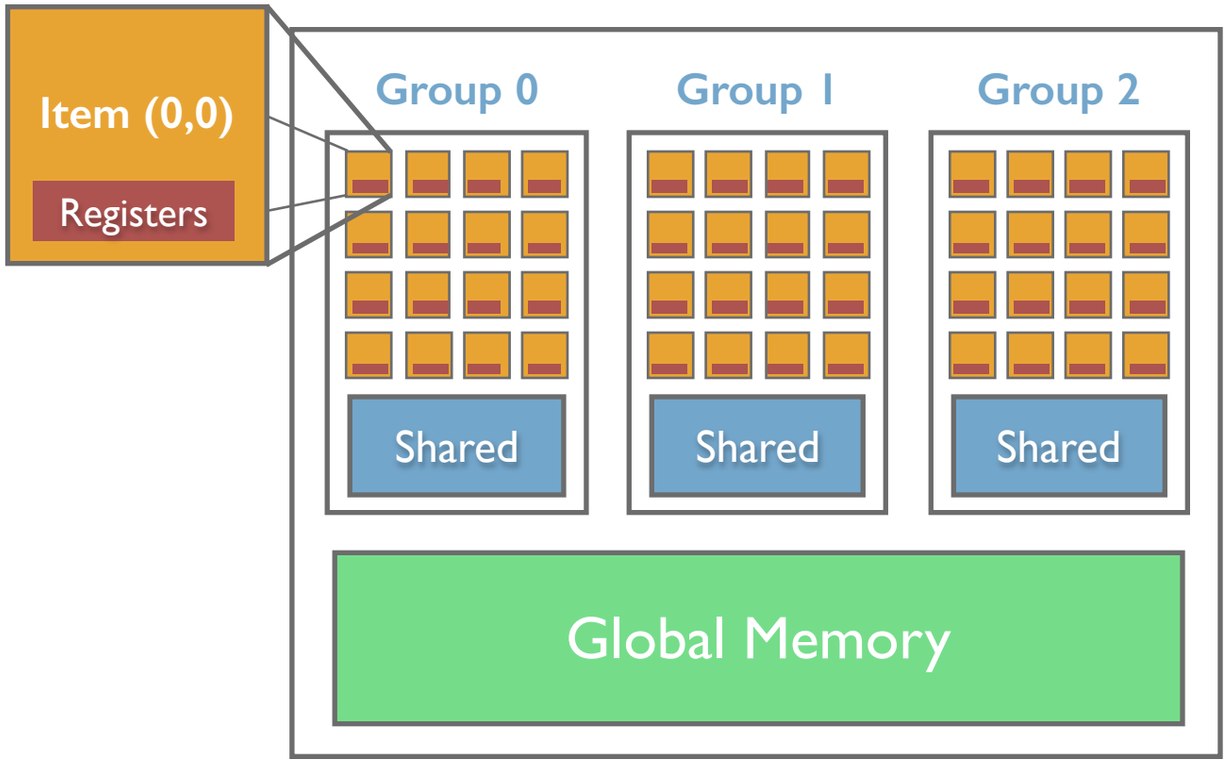


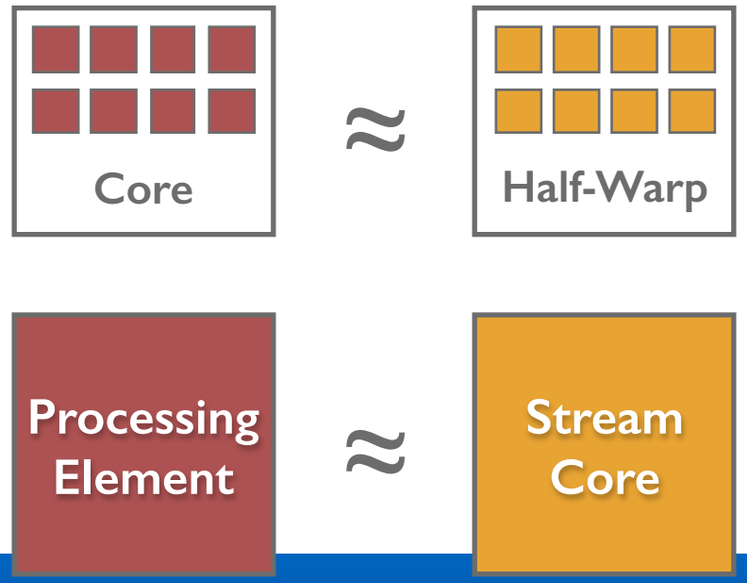
Parallelization Paradigm

CPU Architecture



GPU Architecture

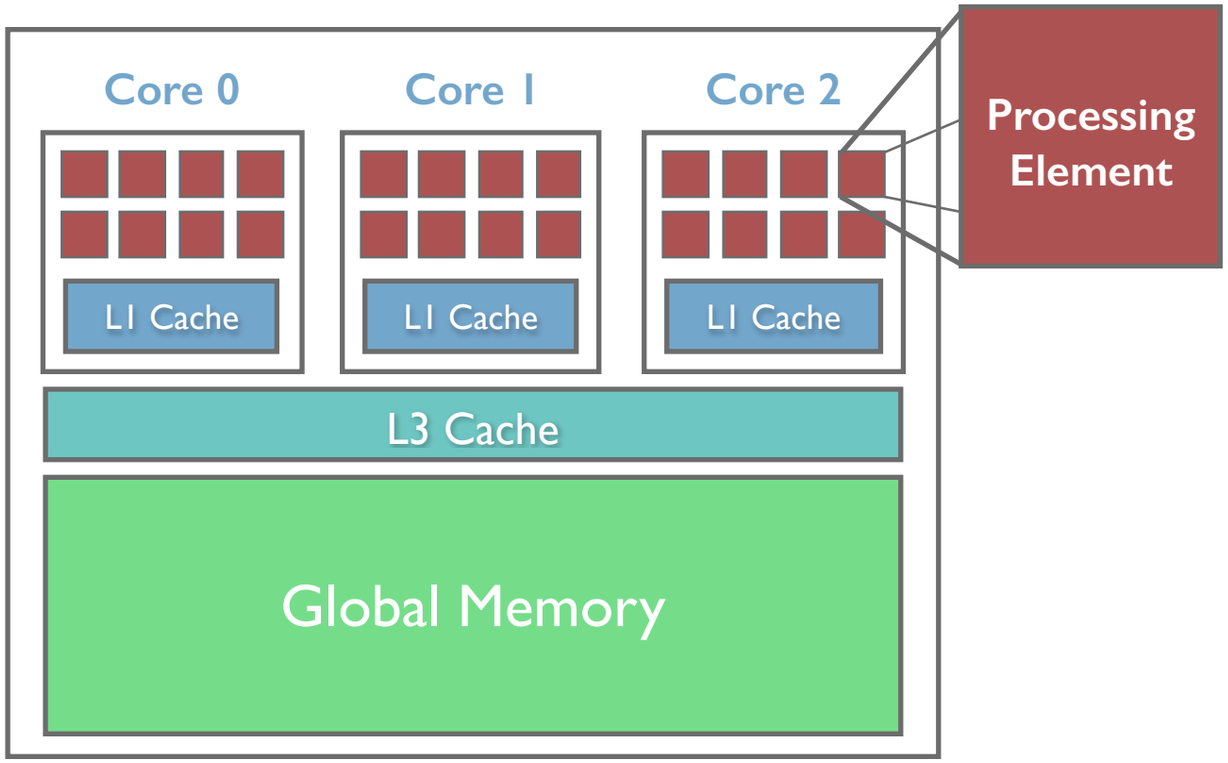




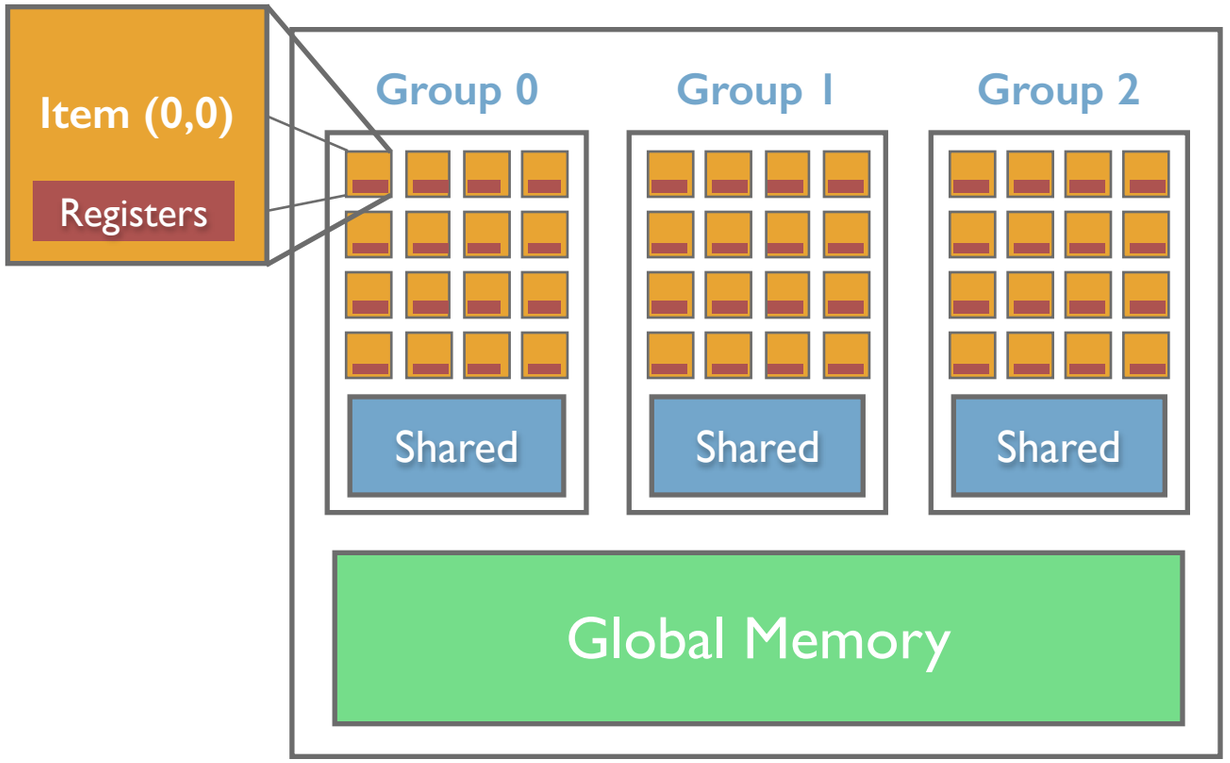
Computational hierarchies are similar

Parallelization Paradigm

CPU Architecture



GPU Architecture




```
void cpuFunction(){
  #pragma omp parallel for
  for(int i = 0; i < work; ++i){

    Do [hopefully thread-independent] work

  }

}
```

```
__kernel void gpuFunction(){
// for each work-group {
//   for each work-item in group {

  Do [group-independent] work

//   }
// }
}
```

Computational hierarchies are similar

OCCA Device Abstractions

OCCA Intermediate Representation (IR)

Device Kernel Language

- Composed of macros which mask supported languages
- Uses the GPU programming model (work-groups / work-items)

Exposes implicit work-groups / work-items

IR

```
occaOuterFor2{        // Work-group implicit loop
  occaOuterFor1{
    occaOuterFor0{

      occaInnerFor2{ // Work-item implicit loop
        occaInnerFor1{
          occaInnerFor0{
            // GPU kernel scope
      }}}
}}}
```

OpenMP

```
_Pragma("omp parallel for
        firstprivate(occaInnerId0, occaInnerId1, occaInnerId2,
                     occaDims0   , occaDims1   , occaDims2)")
for(occaOuterId2 = 0; occaOuterId2 < occaOuterDim2; ++occaOuterId2){
  for(occaOuterId1 = 0; occaOuterId1 < occaOuterDim1; ++occaOuterId1){
    for(occaOuterId0 = 0; occaOuterId0 < occaOuterDim0; ++occaOuterId0){

      for(occaInnerId2 = 0; occaInnerId2 < occaInnerDim2; ++occaInnerId2){
        for(occaInnerId1 = 0; occaInnerId1 < occaInnerDim1; ++occaInnerId1){
          for(occaInnerId0 = 0; occaInnerId0 < occaInnerDim0; ++occaInnerId0){
            // GPU kernel scope
      }}}
}}}
```

OpenCL

```
// GPU kernel scope
```

NVIDIA CUDA

```
// GPU kernel scope
```

The concept of iterating over groups and items is simple

OCCA Intermediate Representation (IR)

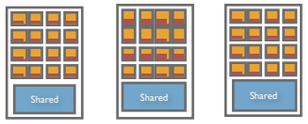
Device Kernel Language

- Composed of macros which mask supported languages
- Uses the GPU programming model (work-groups / work-items)

Exposes implicit work-groups / work-items

IR

```
occaOuterFor2{          // Work-group implicit loop
  occaOuterFor1{
    occaOuterFor0{

      occaInnerFor2{ // Work-item implicit loop
        occaInnerFor1{
          occaInnerFor0{
            // GPU kernel scope
      }}}
}}}
```



OpenMP

```
_Pragma("omp parallel for
        firstprivate(occaInnerId0, occaInnerId1, occaInnerId2,
                     occaDims0   , occaDims1   , occaDims2)")
for(occaOuterId2 = 0; occaOuterId2 < occaOuterDim2; ++occaOuterId2){
  for(occaOuterId1 = 0; occaOuterId1 < occaOuterDim1; ++occaOuterId1){
    for(occaOuterId0 = 0; occaOuterId0 < occaOuterDim0; ++occaOuterId0){

      for(occaInnerId2 = 0; occaInnerId2 < occaInnerDim2; ++occaInnerId2){
        for(occaInnerId1 = 0; occaInnerId1 < occaInnerDim1; ++occaInnerId1){
          for(occaInnerId0 = 0; occaInnerId0 < occaInnerDim0; ++occaInnerId0){
            // GPU kernel scope
      }}}
}}}
```

OpenCL

```
// GPU kernel scope
```

NVIDIA CUDA

```
// GPU kernel scope
```

The concept of iterating over groups and items is simple

OCCA Intermediate Representation (IR)

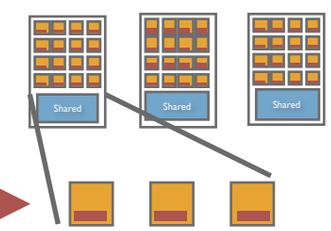
Device Kernel Language

- Composed of macros which mask supported languages
- Uses the GPU programming model (work-groups / work-items)

Exposes implicit work-groups / work-items

IR

```
occaOuterFor2{          // Work-group implicit loop
  occaOuterFor1{
    occaOuterFor0{

      occaInnerFor2{ // Work-item implicit loop
        occaInnerFor1{
          occaInnerFor0{
            // GPU kernel scope
      }}}
}}}
```



OpenMP

```
_Pragma("omp parallel for
        firstprivate(occaInnerId0, occaInnerId1, occaInnerId2,
                     occaDims0   , occaDims1   , occaDims2)")
for(occaOuterId2 = 0; occaOuterId2 < occaOuterDim2; ++occaOuterId2){
  for(occaOuterId1 = 0; occaOuterId1 < occaOuterDim1; ++occaOuterId1){
    for(occaOuterId0 = 0; occaOuterId0 < occaOuterDim0; ++occaOuterId0){

      for(occaInnerId2 = 0; occaInnerId2 < occaInnerDim2; ++occaInnerId2){
        for(occaInnerId1 = 0; occaInnerId1 < occaInnerDim1; ++occaInnerId1){
          for(occaInnerId0 = 0; occaInnerId0 < occaInnerDim0; ++occaInnerId0){
            // GPU kernel scope
      }}}
}}}
```

OpenCL

```
// GPU kernel scope
```

NVIDIA CUDA

```
// GPU kernel scope
```

The concept of iterating over groups and items is simple

OCCA Intermediate Representation (IR)

OCCA-for loop iterator expansions

IR

```
occaOuterFor0{
  occaInnerFor0{
    const int i = (occaOuterId0 * occaInnerDim0) + occaInnerId0;
    AB[i] = A[i] + B[i];
  }
}
```

OCCA	OpenMP	OpenCL	CUDA
occaInnerId0		get_local_id(0)	threadIdx.x
occaInnerId1		get_local_id(1)	threadIdx.y
occaInnerId2		get_local_id(2)	threadIdx.z
occaOuterId0		get_group_id(0)	blockIdx.x
occaOuterId1		get_group_id(1)	blockIdx.y
occaGlobalId0	occaInnerId0 + occaInnerDim0*occaOuterId0	get_global_id(0)	threadIdx.x + blockIdx.x*blockDim.x
occaGlobalId1	occaInnerId1 + occaInnerDim1*occaOuterId1	get_global_id(1)	threadIdx.y + blockIdx.y*blockDim.y
occaGlobalId2	occaInnerId2	get_global_id(2)	threadIdx.z

… and many more macros

OCCA Intermediate Representation (IR)

Shared Memory

- Shared memory is crucial for GPU-modes (requires barrier)
- Manual pre-fetching on CPU-mode
- Barriers are used to synchronize between work-items

IR

```
occaOuterFor0{  // Work-group implicit loops
  occaShared int sharedVar[16];

  occaInnerFor0{  // Work-item implicit loops
    sharedVar[itemX] = itemX;
  }

  occaBarrier(occaLocalMemFence);

  occaInnerFor0{  // Work-item implicit loops
    int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
  }
}
```

Finished inner-loops = all work-items finished their work and can synchronize

OCCA Intermediate Representation (IR)

Shared Memory

- Shared memory is crucial for GPU-modes (requires barrier)
- Manual pre-fetching on CPU-mode
- Barriers are used to synchronize between work-items

IR

```
occaOuterFor0{  // Work-group implicit loops
  occaShared int sharedVar[16];

   occaInnerFor0{  // Work-item implicit loops
     sharedVar[itemX] = itemX;
   }

   occaBarrier(occaLocalMemFence);

   occaInnerFor0{  // Work-item implicit loops
     int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
   }
}
```

Finished inner-loops = all work-items finished their work and can synchronize

OCCA Intermediate Representation (IR)

Register Memory

- Fastest memory on GPUs
- Another issue with for-loop scopes … requires thread-local-storage memory
- Good for prefetching data

IR

```
occaOuterFor0{  // Work-group implicit loops
  occaPrivate(int, exclusiveVar);            // int exclusiveVar;
  occaPrivateArray(int, exclusiveArray, 10); // int exclusiveArray[10];

   occaInnerFor0{  // Work-item implicit loops
     exclusiveVar = itemX;  // Pre-fetch
   }

   boccaBarrier(occaLocalMemFence);

   occaInnerFor0{  // Work-item implicit loops
     int i = exclusiveVar;  // Use pre-fetched data
   }
}
```

?

Optimizations in OCCA IR can be very verbose ...

Prefetches or variable reuse can be declared occaPrivate to "jump scopes"

OCCA Intermediate Representation (IR)

Register Memory

- Fastest memory on GPUs
- Another issue with for-loop scopes … requires thread-local-storage memory
- Good for prefetching data

IR

```
exclusiveVar = 0

occaOuterFor0{  // Work-group implicit loops
  occaPrivate(int, exclusiveVar);            // int exclusiveVar;
  occaPrivateArray(int, exclusiveArray, 10); // int exclusiveArray[10];

  occaInnerFor0{  // Work-item implicit loops
    exclusiveVar = itemX;  // Pre-fetch
  }

  occaBarrier(occaLocalMemFence);

  occaInnerFor0{  // Work-item implicit loops
    int i = exclusiveVar;  // Use pre-fetched data
  }
}
```

?

Optimizations in OCCA IR can be very verbose ...

Prefetches or variable reuse can be declared occaPrivate to "jump scopes"

OCCA Intermediate Representation (IR)

Register Memory

- Fastest memory on GPUs
- Another issue with for-loop scopes … requires thread-local-storage memory
- Good for prefetching data

IR

```
exclusiveVar = 0
exclusiveVar = 1
```

```
occaOuterFor0{  // Work-group implicit loops
  occaPrivate(int, exclusiveVar);            // int exclusiveVar;
  occaPrivateArray(int, exclusiveArray, 10); // int exclusiveArray[10];

  occaInnerFor0{  // Work-item implicit loops
    exclusiveVar = itemX;  // Pre-fetch
  }

  occaBarrier(occaLocalMemFence);

  occaInnerFor0{  // Work-item implicit loops
    int i = exclusiveVar;  // Use pre-fetched data
  }
}
```

?

Optimizations in OCCA IR can be very verbose ...

Prefetches or variable reuse can be declared occaPrivate to "jump scopes"

OCCA Intermediate Representation (IR)

Register Memory

- Fastest memory on GPUs
- Another issue with for-loop scopes … requires thread-local-storage memory
- Good for prefetching data

IR

```
exclusiveVar = 0
exclusiveVar = 1
exclusiveVar = 2
.
.
.
```

```
occaOuterFor0{  // Work-group implicit loops
  occaPrivate(int, exclusiveVar);            // int exclusiveVar;
  occaPrivateArray(int, exclusiveArray, 10); // int exclusiveArray[10];

  occaInnerFor0{  // Work-item implicit loops
    exclusiveVar = itemX;  // Pre-fetch
  }

  occaBarrier(occaLocalMemFence);

  occaInnerFor0{  // Work-item implicit loops
    int i = exclusiveVar;  // Use pre-fetched data
  }
}
```

?

Optimizations in OCCA IR can be very verbose ...

Prefetches or variable reuse can be declared occaPrivate to "jump scopes"

OKL: OCCA Kernel Language

Description

- Minimal extensions to C, familiar for regular programmers
- Translates to OCCA IR with code transformations
- 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
        }}}
  }}}

  ...
}
```

The concept of iterating over groups and items is simple

OKL: OCCA Kernel Language

Description

- Minimal extensions to C, familiar for regular programmers
- Translates to OCCA IR with code transformations
- 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

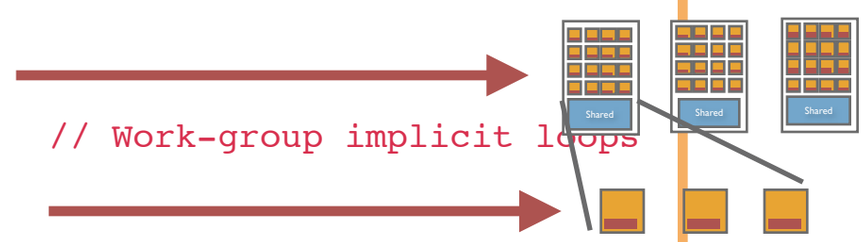
The concept of iterating over groups and items is simple

OKL: OCCA Kernel Language

Description

- Minimal extensions to C, familiar for regular programmers
- Translates to OCCA IR with code transformations
- 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
        }}}
  }}}

  ...
}
```



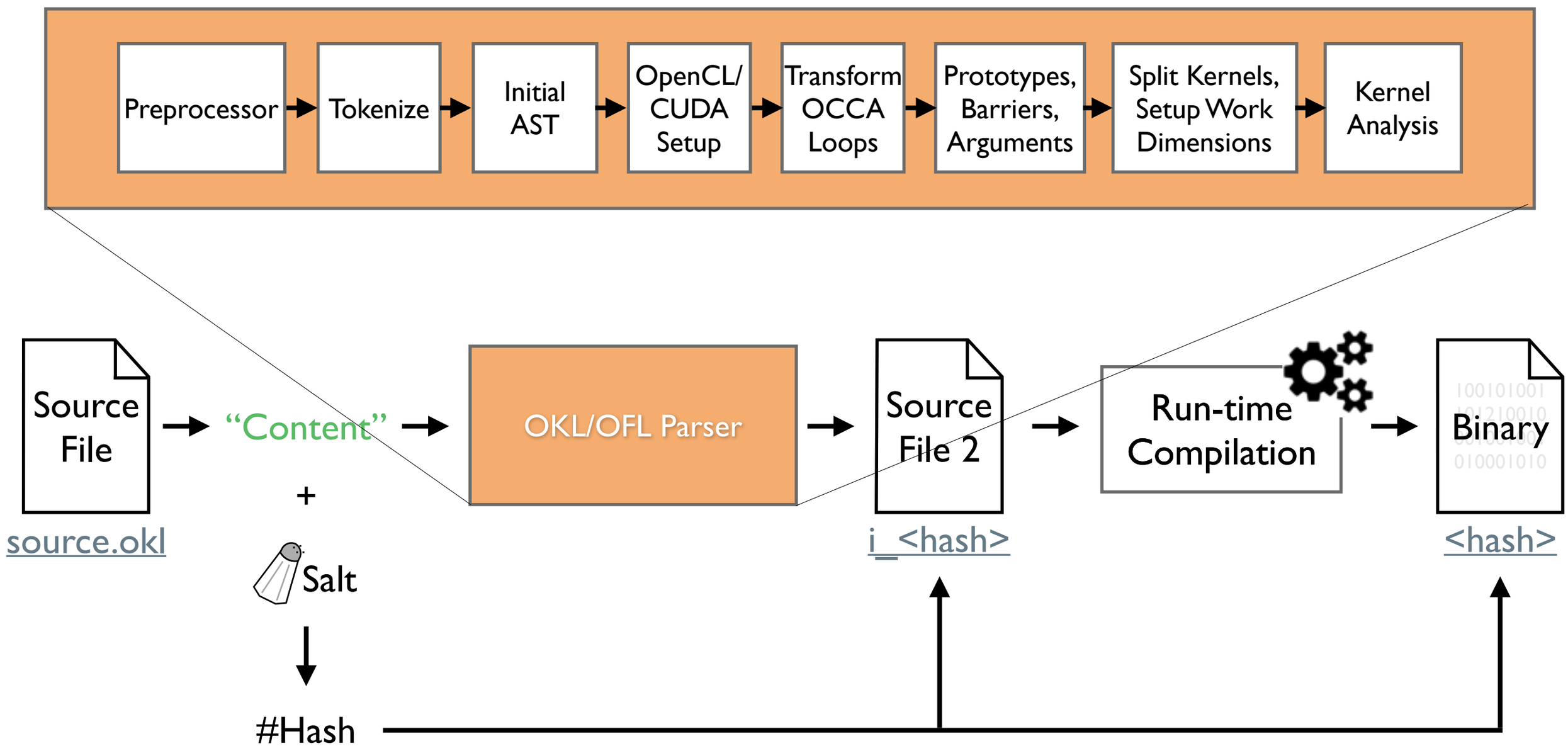
OKL

The concept of iterating over groups and items is simple

OKL: OCCA Kernel Language

Source-to-Source Compilation

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

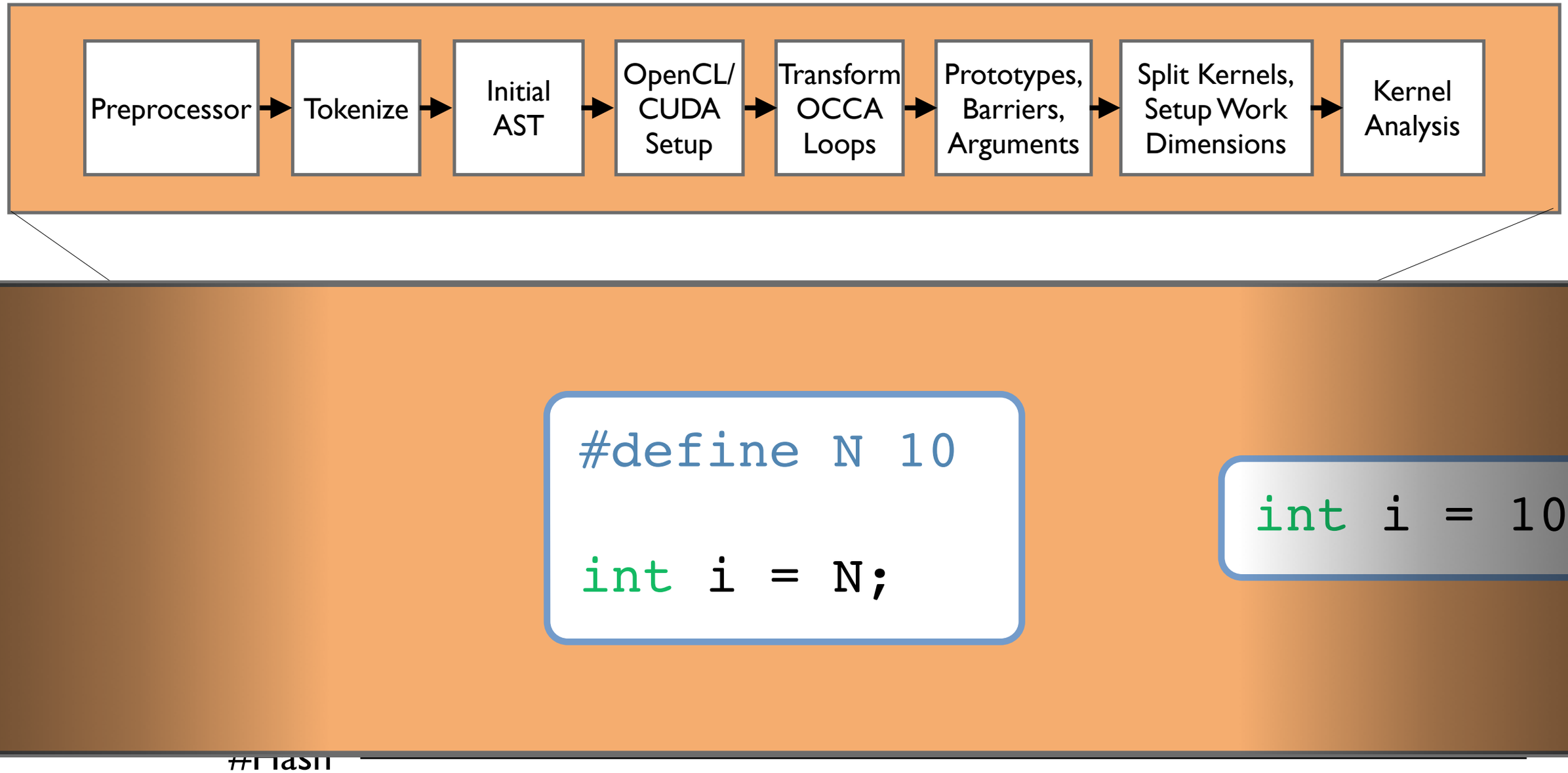


Custom compilation tools tailored for code manipulation and analysis

OKL: OCCA Kernel Language

Source-to-Source Compilation

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

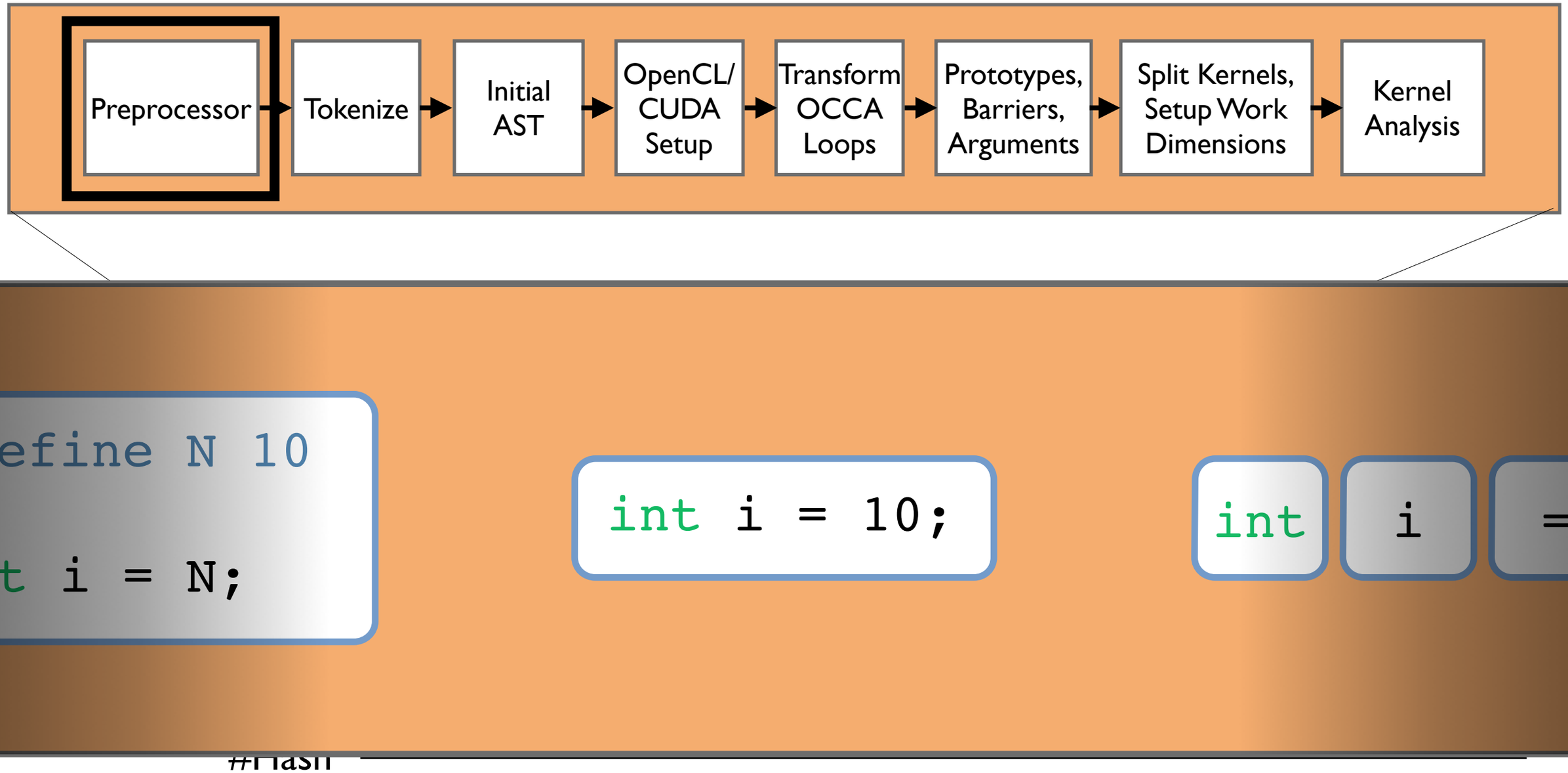


Custom compilation tools tailored for code manipulation and analysis

OKL: OCCA Kernel Language

Source-to-Source Compilation

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

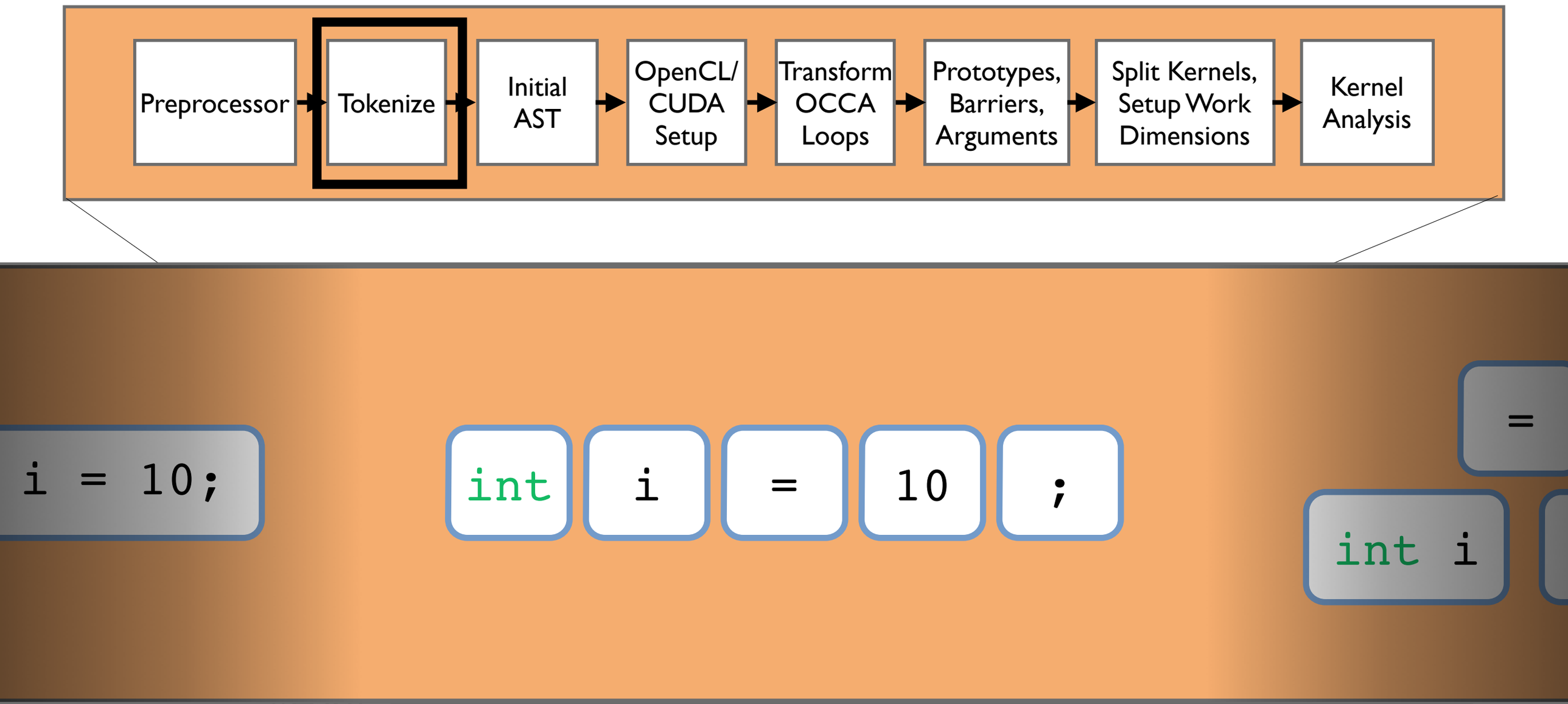


Custom compilation tools tailored for code manipulation and analysis

OKL: OCCA Kernel Language

Source-to-Source Compilation

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

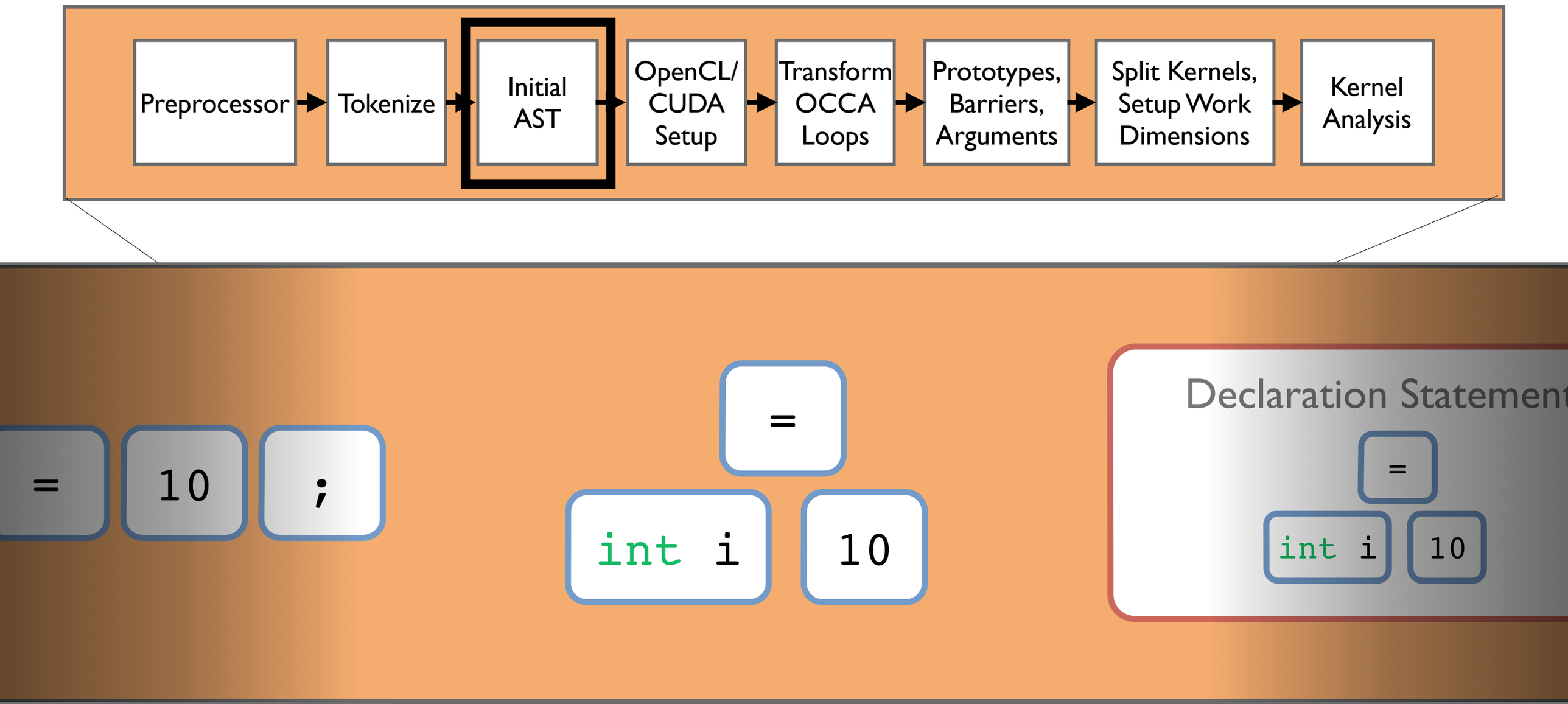


Custom compilation tools tailored for code manipulation and analysis

OKL: OCCA Kernel Language

Source-to-Source Compilation

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

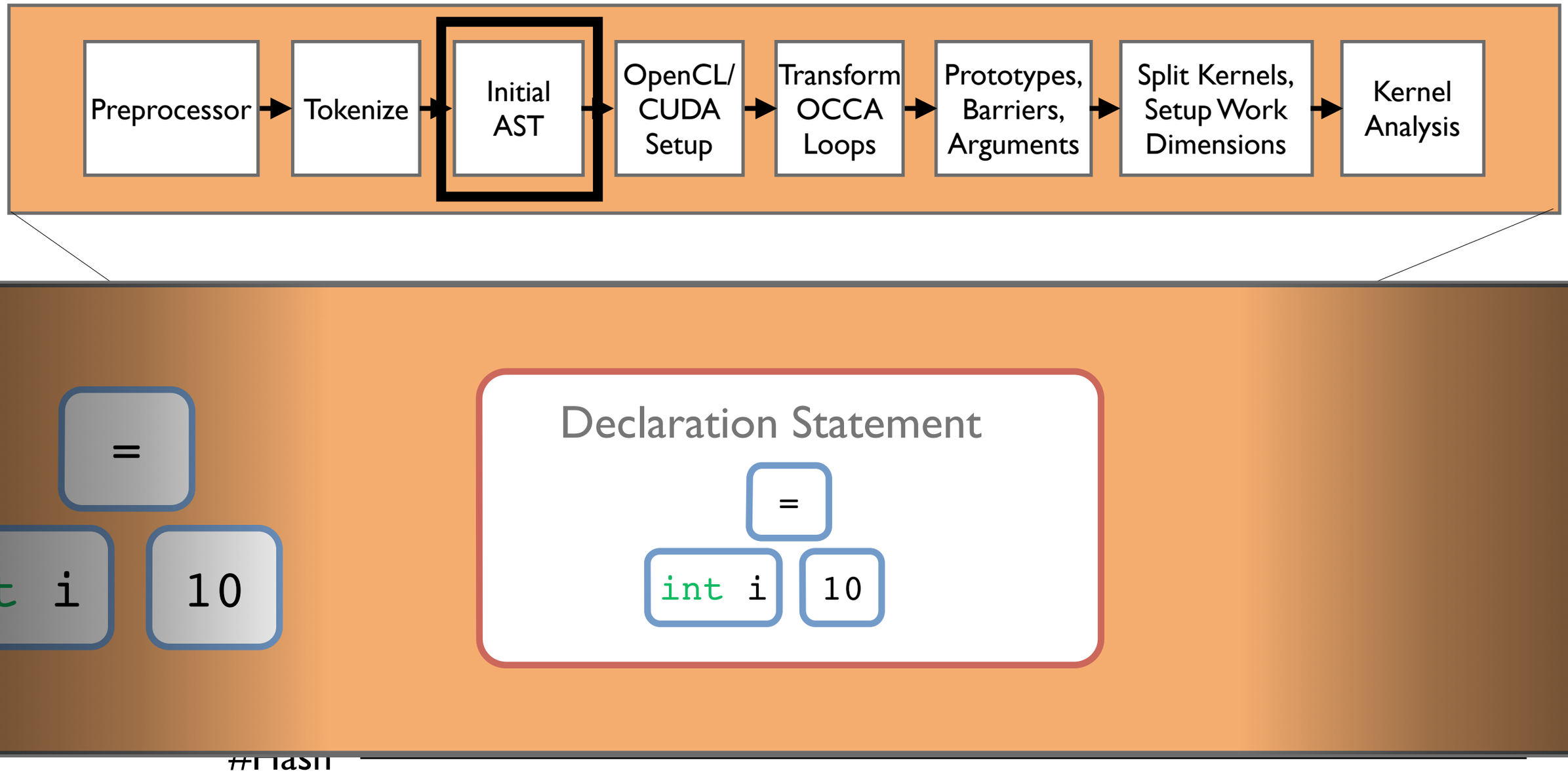


Custom compilation tools tailored for code manipulation and analysis

OKL: OCCA Kernel Language

Source-to-Source Compilation

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

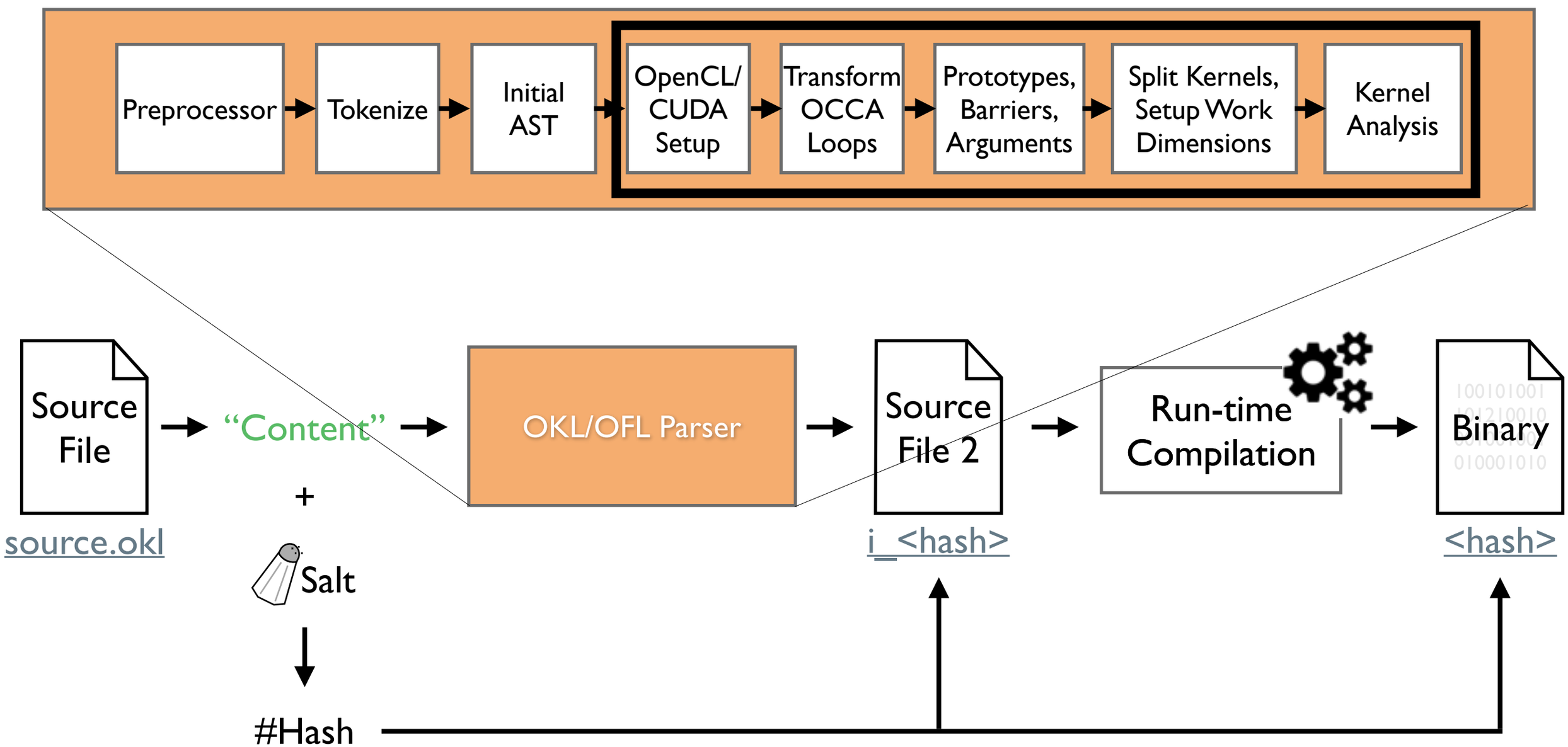


Custom compilation tools tailored for code manipulation and analysis

OKL: OCCA Kernel Language

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

OKL: OCCA Kernel Language

Multiple outer-loops

- Sets of 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

Execute a set of outer-loops is equivalent to launching a CUDA/OpenCL kernel

OKL: OCCA Kernel Language

Multiple outer-loops

- Sets of 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

NVIDIA CUDA

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

Execute a set of outer-loops is equivalent to launching a CUDA/OpenCL kernel

OKL: OCCA Kernel Language

Multiple outer-loops

- Sets of 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

NVIDIA CUDA

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

Execute a set of outer-loops is equivalent to launching a CUDA/OpenCL kernel

OKL: OCCA Kernel Language

Multiple outer-loops

- Sets of 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

Execute a set of outer-loops is equivalent to launching a CUDA/OpenCL kernel

OKL: OCCA Kernel Language

Multiple outer-loops

- Sets of 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

Execute a set of outer-loops is equivalent to launching a CUDA/OpenCL kernel

OKL: OCCA Kernel Language

Multiple outer-loops

- Sets of 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

Execute a set of outer-loops is equivalent to launching a CUDA/OpenCL kernel

OKL: OCCA Kernel Language

Multiple outer-loops

- Sets of 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

Execute a set of outer-loops is equivalent to launching a CUDA/OpenCL kernel

OKL: OCCA 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

Register Memory (SIMD Variables)

```
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 (gives a warning)

OKL: OCCA 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

Register Memory (SIMD Variables)

```
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 (gives a warning)

OFL: OCCA Fortran Language

Description

- Translates to OKL and then to OCCA IR with code transformations
- Parallel loops are explicit through the inner and outer DO-labels

```
kernel subroutine kernelName(...)
  ...

  DO groupY = 1, yGroups, outer1
   DO groupX = 1, xGroups, outer0 // Work-group implicit loops
    DO itemY = 1, yItems, inner1
     DO itemX = 1, xItems, inner0 // Work-item implicit loops
      // GPU Kernel Scope
     END DO
    END DO
   END DO
  END DO

  ...
end subroutine kernelName
```

OFL

Shared and Exclusive Memory

```
integer(4), shared    :: sharedVar(16,30)
integer(4), exclusive :: exclusiveVar, exclusiveArray(10)
```

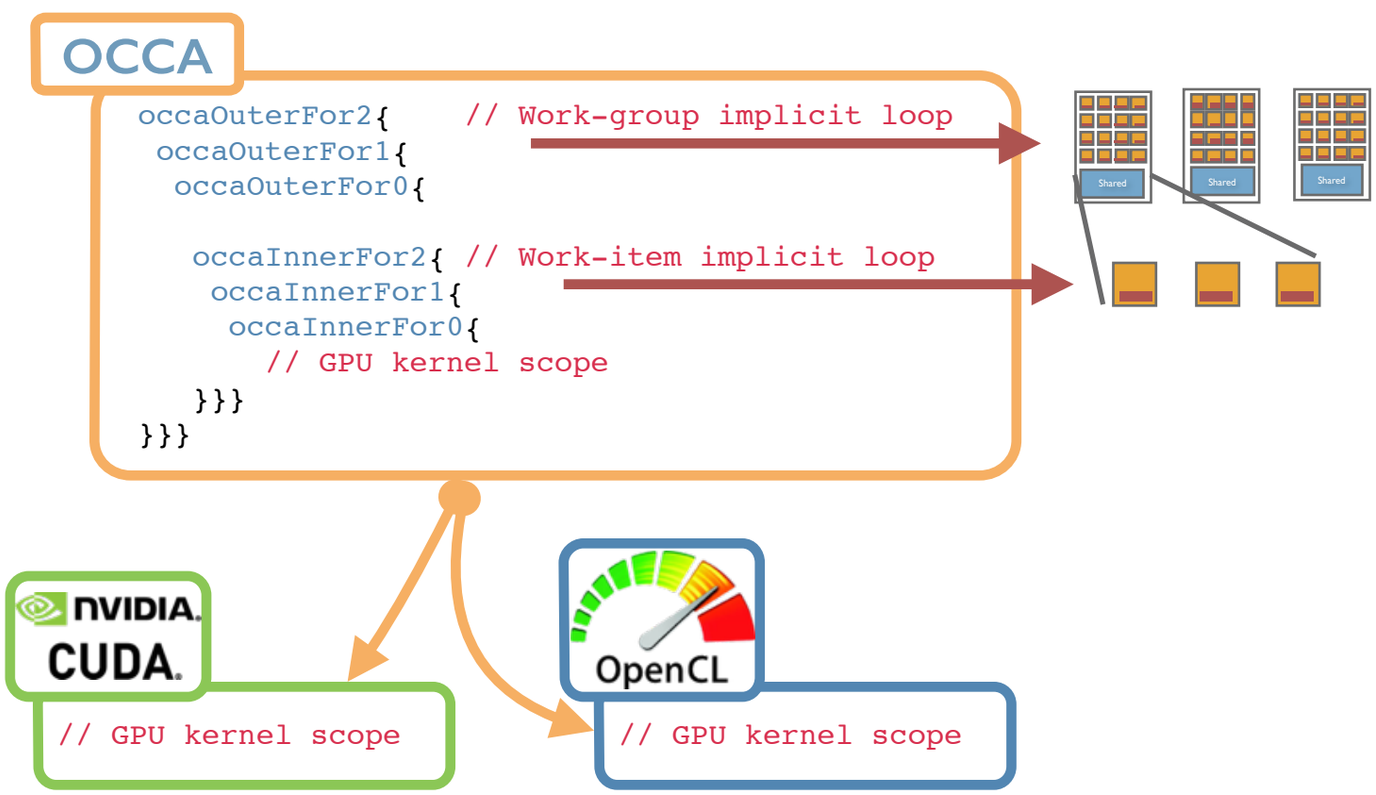
OFL

Because [OFL -> OKL], all features added to OKL are inherintly added to OFL

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

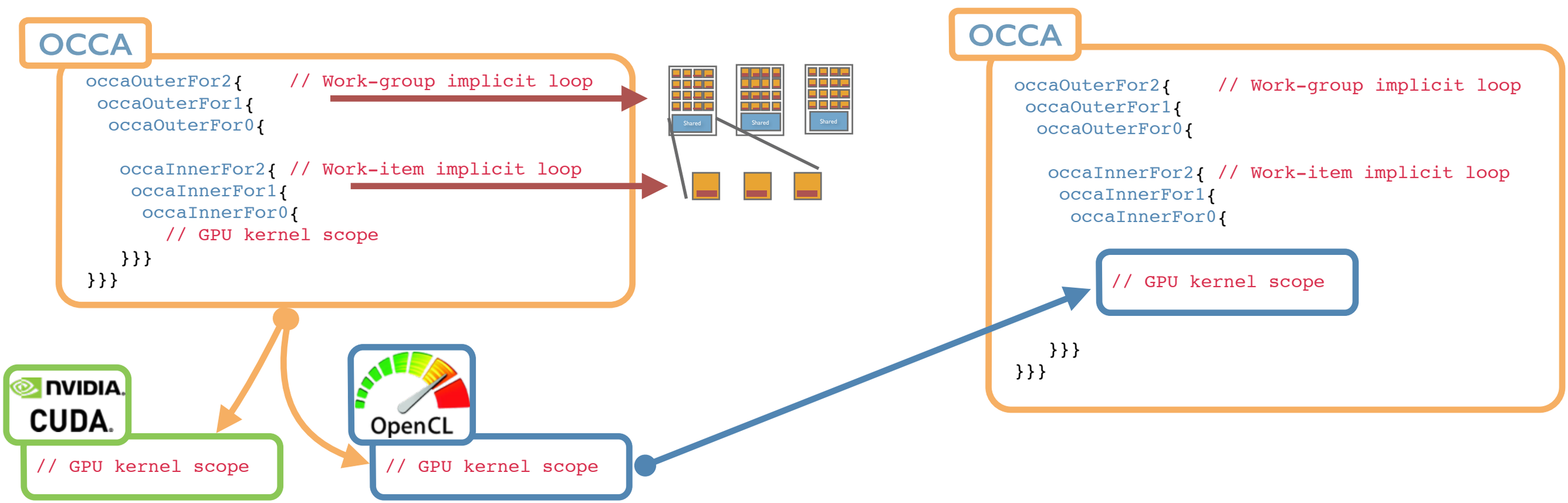


Since we derived OCCA IR from the GPU model, the inverse should be easy … right?

OpenCL/CUDA to OCCA IR

Description

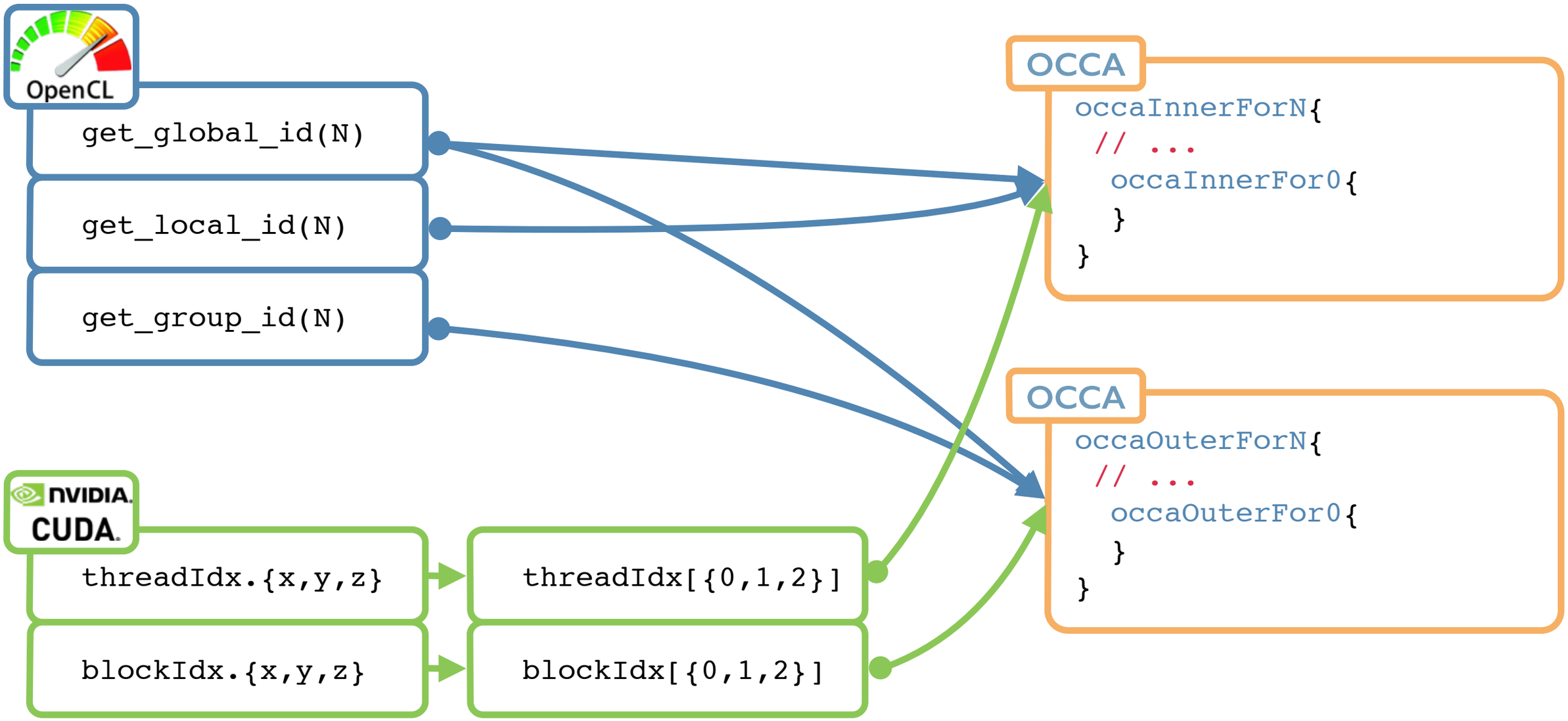
- Parser can translate OpenCL/CUDA kernels to OCCA IR*
- Although OCCA IR was derived from the GPU model, there are complexities



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

OpenCL/CUDA to OCCA

Kernel Work Dimensions



*Working thread dimensions are deduced through the use of these built-ins**

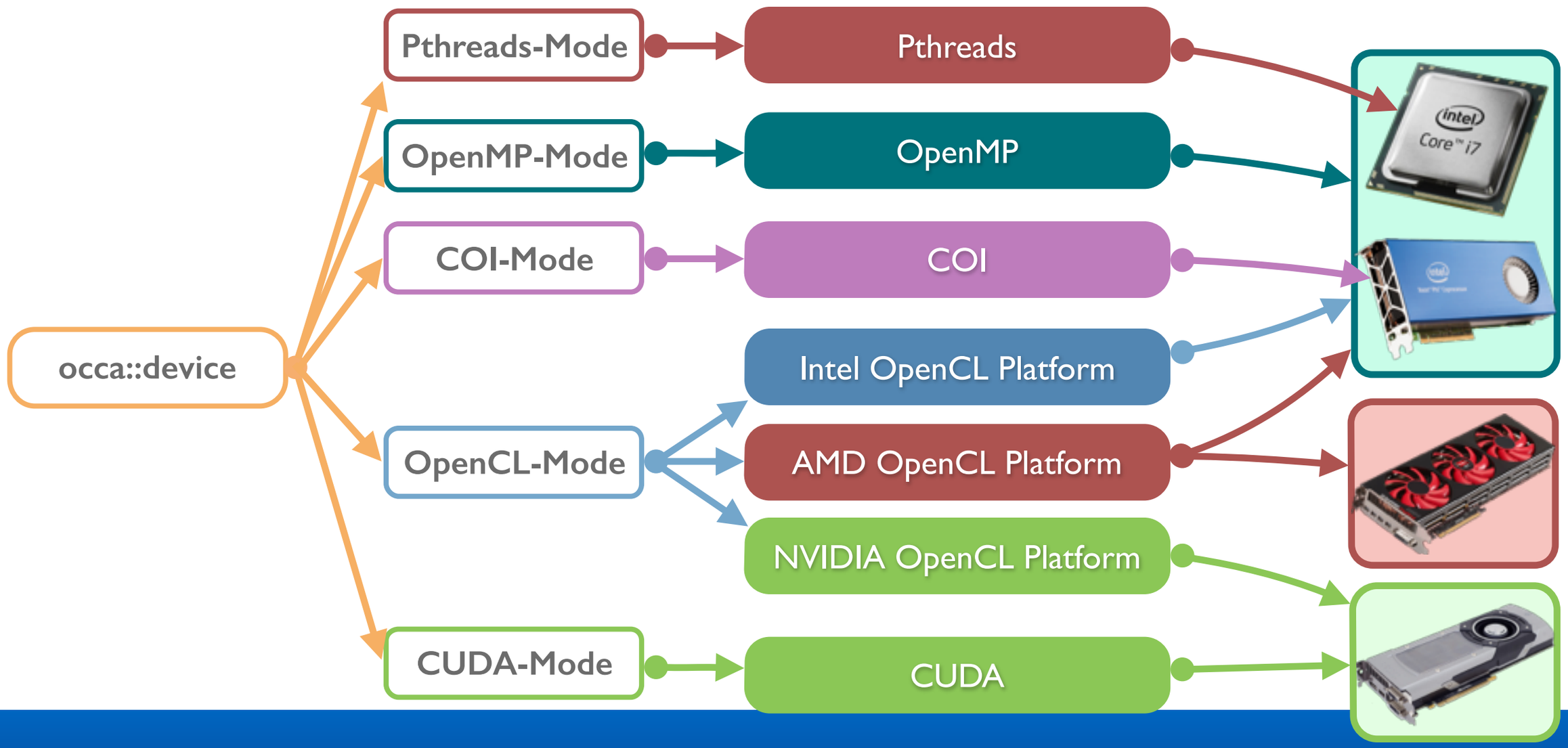
OCCA Application Programming Interface (API)

1. `occa::device` (C++ Class)
2. `occa::memory` (C++ Class)
3. `occa::kernel` (C++ Class)

OCCA Application Programming Interface (API)

`occa::device` (C++ Class)

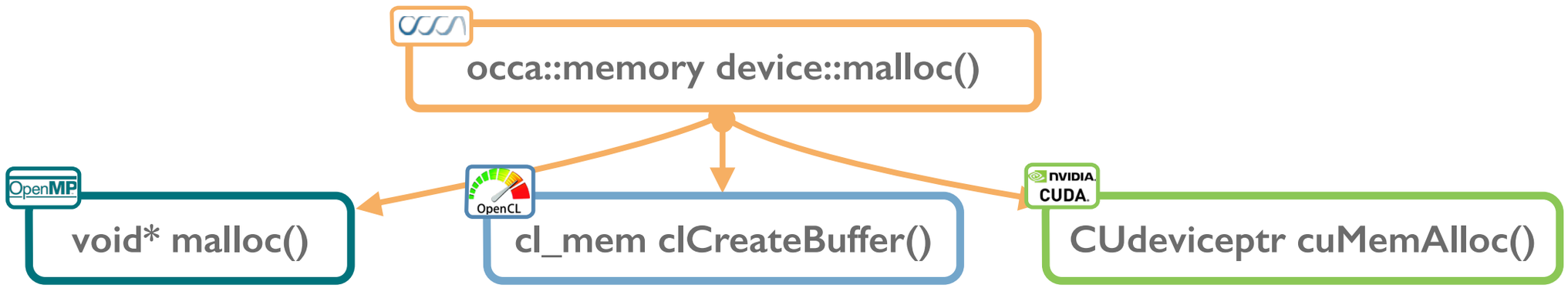
- Choose between using the CPU or available accelerators
- In charge of allocating memory and compiling kernels



OCCA API

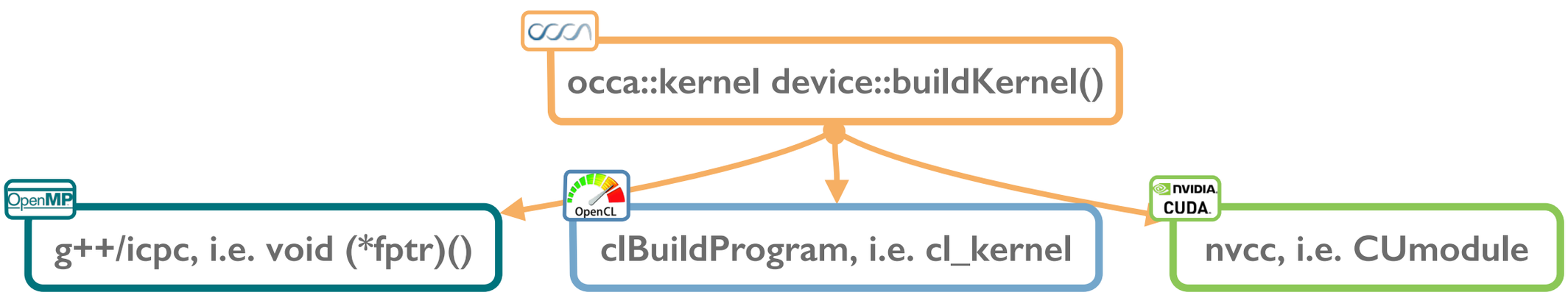
`occa::memory` (C++ Class)

- Abstracts the memory handles found in each language
- Asynchronous memory transfers are supported



`occa::kernel` **(C++ Class)**

- Uses run-time compilation
- Kernel binaries are cached to prevent re-compiling



OCCA API: Adding Two Vectors

```
#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("mode = OpenCL, platformID = 0, deviceID = 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.okl",
                                            "addVectors");

  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';
```

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OCCA API: Adding Two Vectors

```
#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("mode = OpenCL, platformID = 0, deviceID = 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.okl",
                                            "addVectors");

  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';
```

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OCCA API: Adding Two Vectors

```
#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("mode = OpenCL, platformID = 0, deviceID = 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.okl",
                                            "addVectors");

  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';
```

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OCCA API: Adding Two Vectors

```
#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("mode = OpenCL, platformID = 0, deviceID = 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.okl",
                                            "addVectors");

  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';
```

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OCCA API: Adding Two Vectors

```
#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("mode = OpenCL, platformID = 0, deviceID = 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.okl",
                                            "addVectors");

  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';
```

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OCCA API: Adding Two Vectors

```
#include "occa.hpp"

int main(int argc, char
  float *a  = new float
  float *b  = new float
  float *ab = new float

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

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

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

  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';
```

```
kernel void addVectors(const int entries,
                       const float *a,
                       const float *b,
                       float *ab){
  for(int group = 0; group < ((entries + 15)/16); ++group; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      const int N = (item + (16 * group));

      if(N < entries)
        ab[N] = a[N] + b[N];
    }
  }
}
```

OKL

OCCA API: Adding Two Vectors

```
#include "occa.hpp"

int main(int argc, char
  float *a  = new float
  float *b  = new float
  float *ab = new float

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

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

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

  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';
```

```
kernel void addVectors(const int entries,
                       const float *a,
                       const float *b,
                       float *ab){
  for(int group = 0; group < ((entries + 15)/16); ++group; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      const int N = (item + (16 * group));

      if(N < entries)
        ab[N] = a[N] + b[N];
    }
  }
}
```

OKL

OCCA API: Adding Two Vectors

```
#include "occa.hpp"

int main(int argc, char
  float *a  = new float
  float *b  = new float
  float *ab = new float

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

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

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

  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';
```

```
kernel void addVectors(const int entries,
                       const float *a,
                       const float *b,
                       float *ab){
  for(int group = 0; group < ((entries + 15)/16); ++group; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      const int N = (item + (16 * group));

      if(N < entries)
        ab[N] = a[N] + b[N];
    }
  }
}
```

OKL

OCCA API: Adding Two Vectors

```
#include "occa.hpp"

int main(int argc, char
  float *a  = new float
  float *b  = new float
  float *ab = new float

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

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

  device.setup("mode =

  o_a  = device.malloc
  o_b  = device.malloc
  o_ab = device.malloc

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

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

  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';
```

```
kernel void addVectors(const int entries,
                       const float *a,
                       const float *b,
                       float *ab){
  for(int group = 0; group < ((entries + 15)/16); ++group; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      const int N = (item + (16 * group));

      if(N < entries)
        ab[N] = a[N] + b[N];
    }
  }
}
```

OKL

```
kernel void addVectors(const int entries,
                       const float *a,
                       const float *b,
                       float *ab){
  for(int N = 0; N < entries; ++N; tile(16)){
    if(N < entries)
      ab[N] = a[N] + b[N];
  }
}
```

OKL

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OFL: Adding Two Vectors

```
program main
  use occa
  implicit none

  character(len=1024) :: deviceInfo = “mode = OpenMP”

  real(4), allocatable :: a(:), b(:), ab(:)

  type(occaDevice) :: device
  type(occaKernel) :: addVectors
  type(occaMemory) :: o_a, o_b, o_ab

  allocate(a(1:5), b(1:5), ab(1:5))

  do i = 1, 5
    a(i)  = i
    b(i)  = 1-i
    ab(i) = 0
  end do

  device = occaGetDevice(deviceInfo)

  o_a  = occaDeviceMalloc(device, int(5,8)*4_8, a(1))
  o_b  = occaDeviceMalloc(device, int(5,8)*4_8, b(1))
  o_ab = occaDeviceMalloc(device, int(5,8)*4_8)

  addVectors = occaBuildKernelFromSource(device, &
                                         "addVectors.ofl", "addVectors")

  call occaKernelRun(addVectors, occaTypeMem_t(5), o_a, o_b, o_ab)

  call occaCopyMemToPtr(ab(1), o_ab);

  print *, "a  = ", a(:)
  print *, "b  = ", b(:)
  print *, "ab = ", ab(:)

end program main
```

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OFL: Adding Two Vectors

```
program main
  use occa
  implicit none

  character(len=1024)

  real(4), allocatabl

  type(occaDevice) ::
  type(occaKernel) ::
  type(occaMemory) ::

  allocate(a(1:5), b(

  do i = 1, 5
    a(i)  = i
    b(i)  = 1-i
    ab(i) = 0
  end do

  device = occaGetDevi

  o_a  = occaDeviceMalloc(device, int(5,8)*4_8, a(1))
  o_b  = occaDeviceMalloc(device, int(5,8)*4_8, b(1))
  o_ab = occaDeviceMalloc(device, int(5,8)*4_8)

  addVectors = occaBuildKernelFromSource(device, &
                                         "addVectors.ofl", "addVectors")

  call occaKernelRun(addVectors, occaTypeMem_t(5), o_a, o_b, o_ab)

  call occaCopyMemToPtr(ab(1), o_ab);

  print *, "a  = ", a(:)
  print *, "b  = ", b(:)
  print *, "ab = ", ab(:)

end program main
```

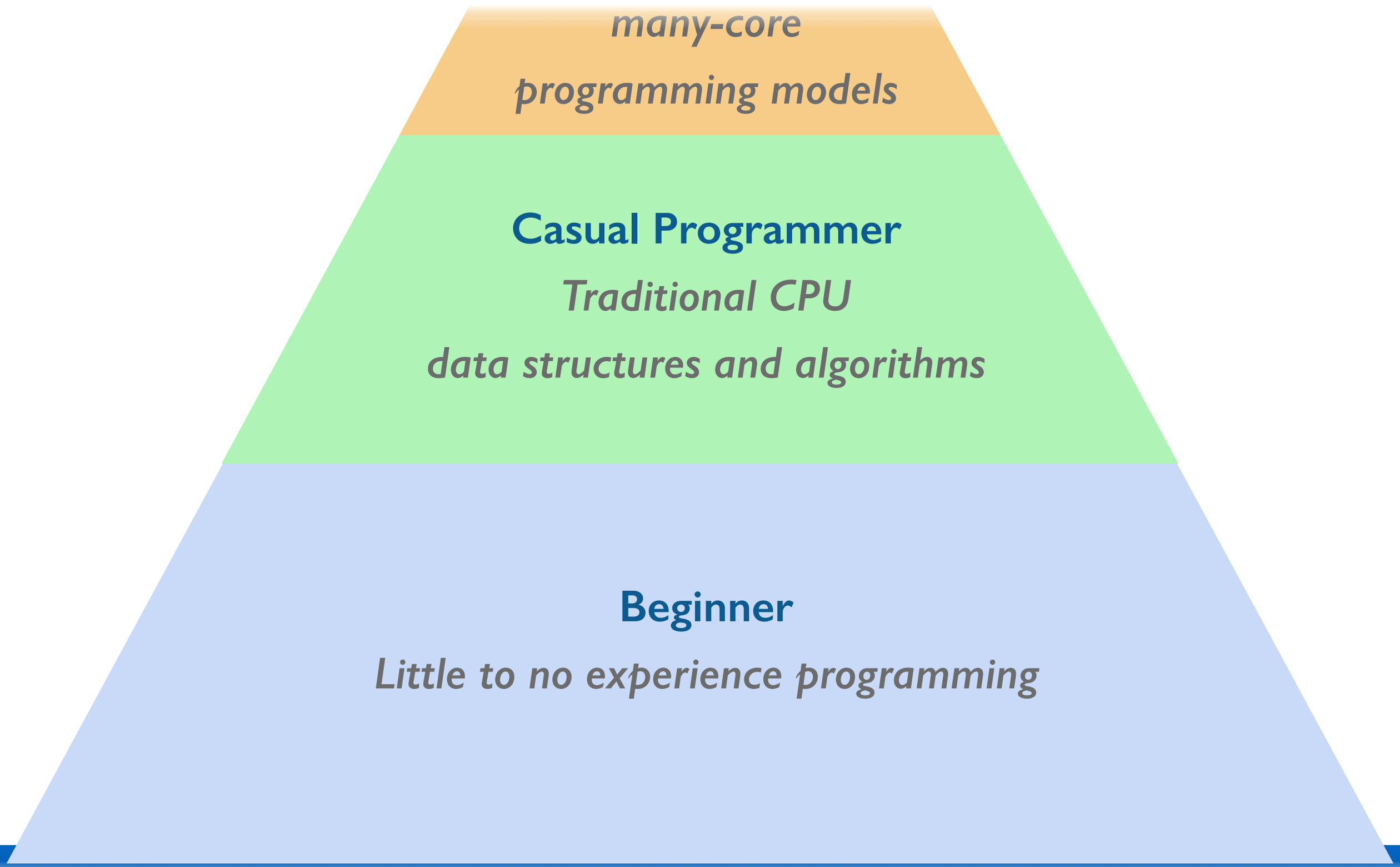
```
kernel subroutine addVectors(entries, a, b, ab)
  implicit none

  integer(4), intent(in)  :: entries
  real(4)   , intent(in)  :: a(:), b(:)
  real(4)   , intent(out) :: ab(:)

  do group = 1, ((entries + 15) / 16), outer0
     do item = 1, 16, inner0
        N = (item + (16 * (group - 1)))

        if (N < entries) then
           ab(i) = a(i) + b(i)
        end if
     end do
  end do

end subroutine addVectors
```

OFL

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

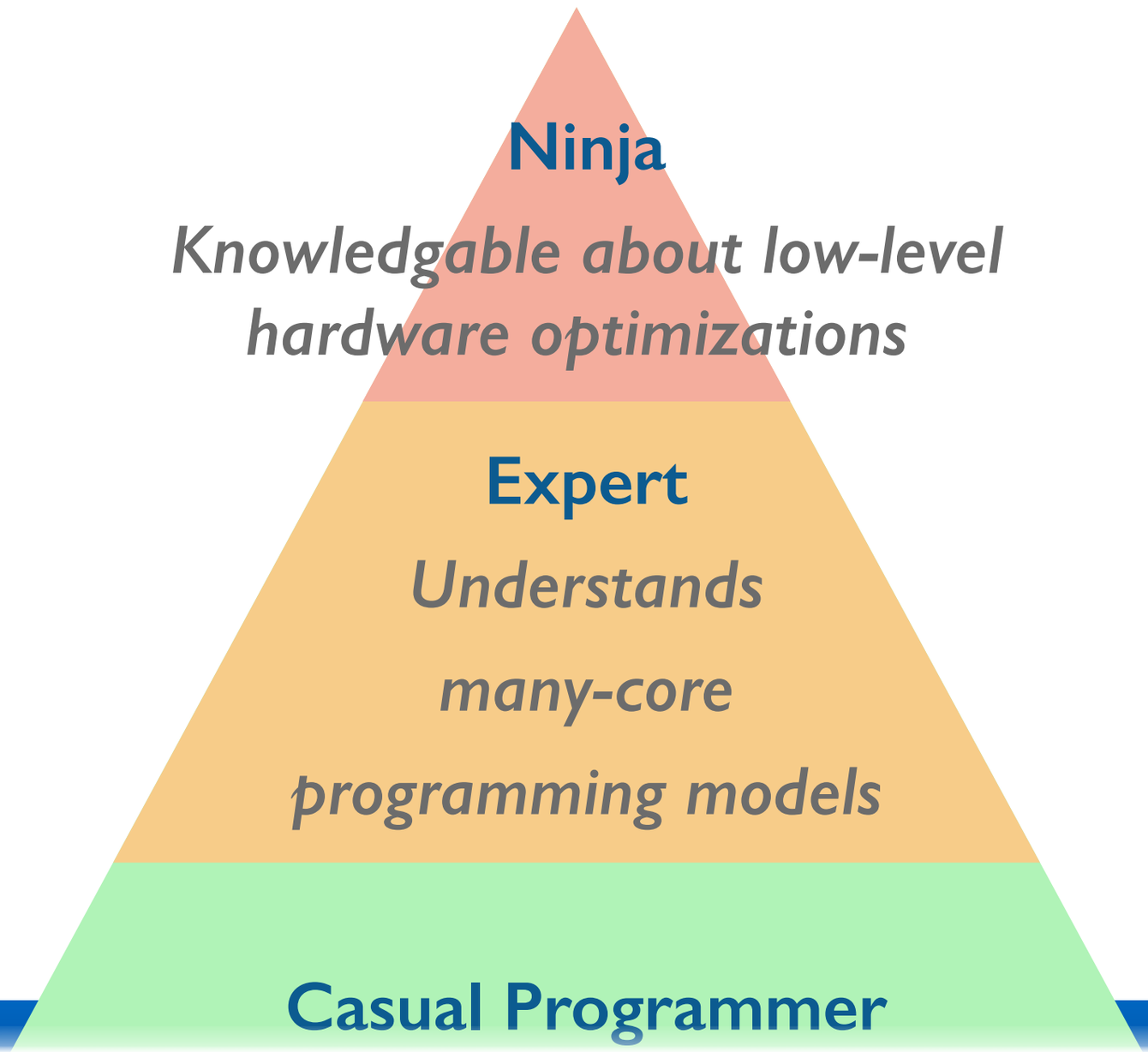
OCCA Automation

Programmer Expertise Hierarchy



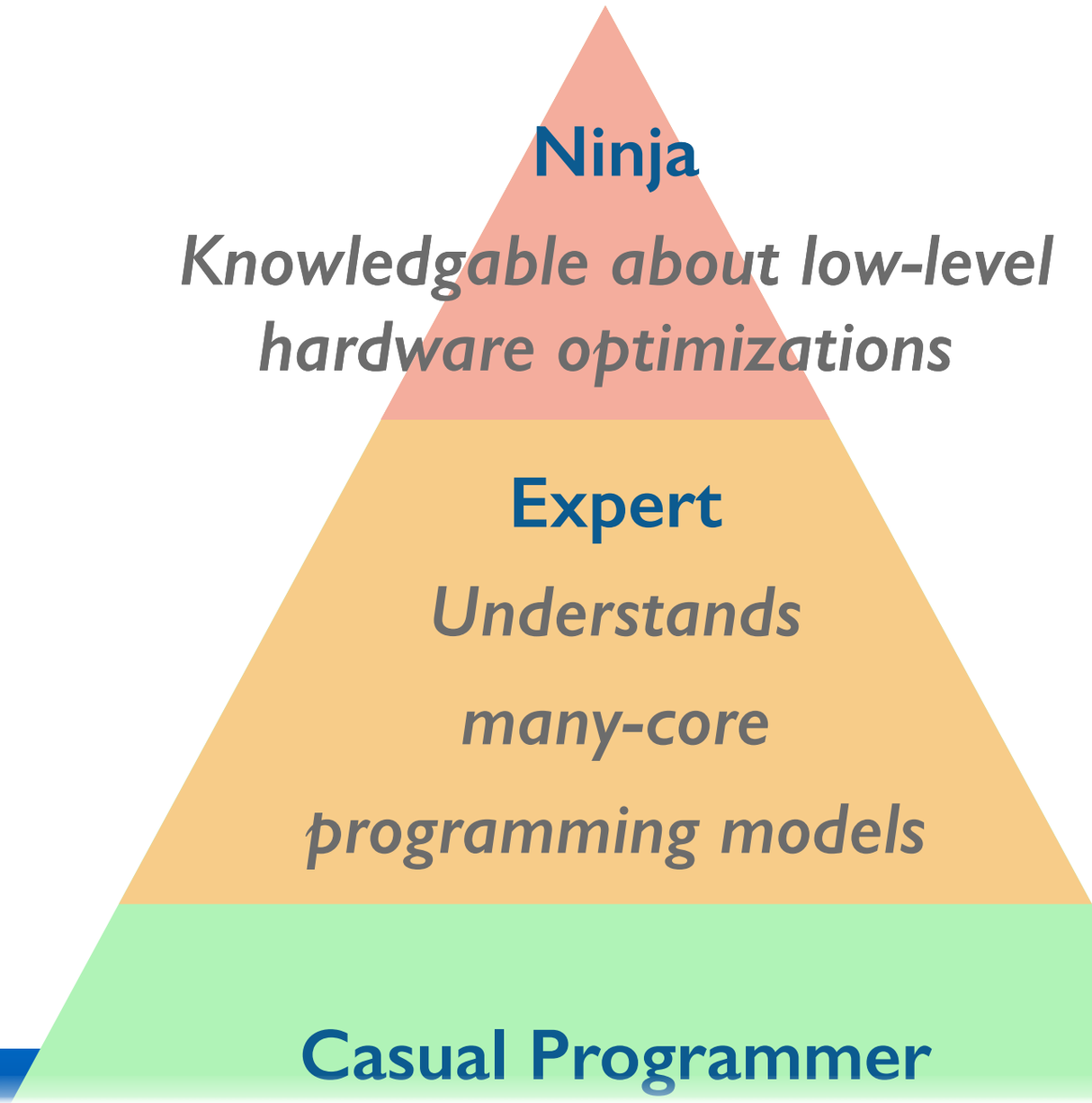
We chose to target developers in each proficiency layer

Programmer Expertise Hierarchy



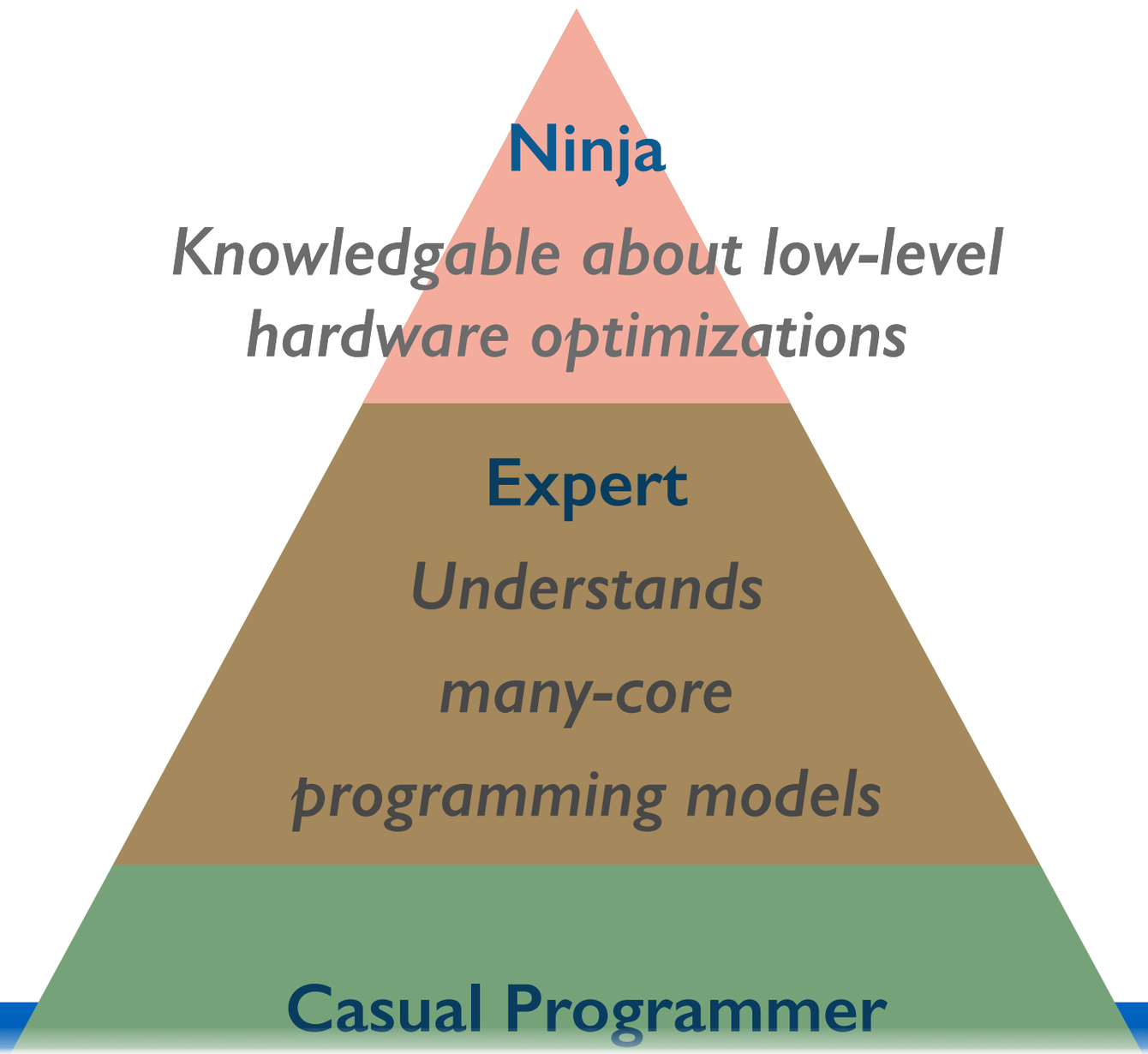
We chose to target developers in each proficiency layer

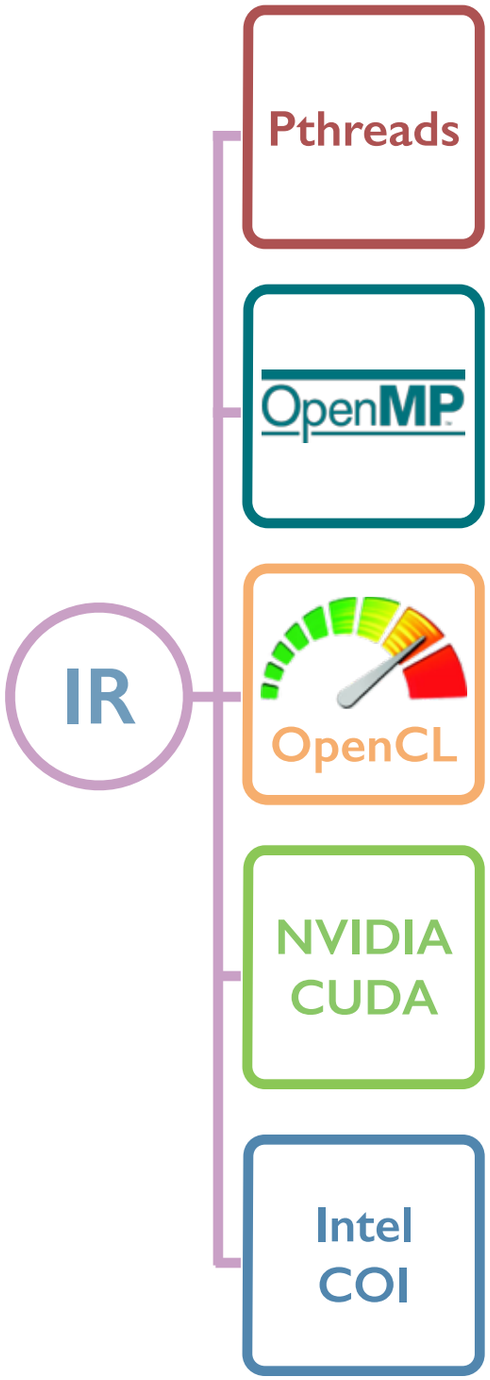
Programmer Expertise Hierarchy



I chose to target developers in each proficiency layer

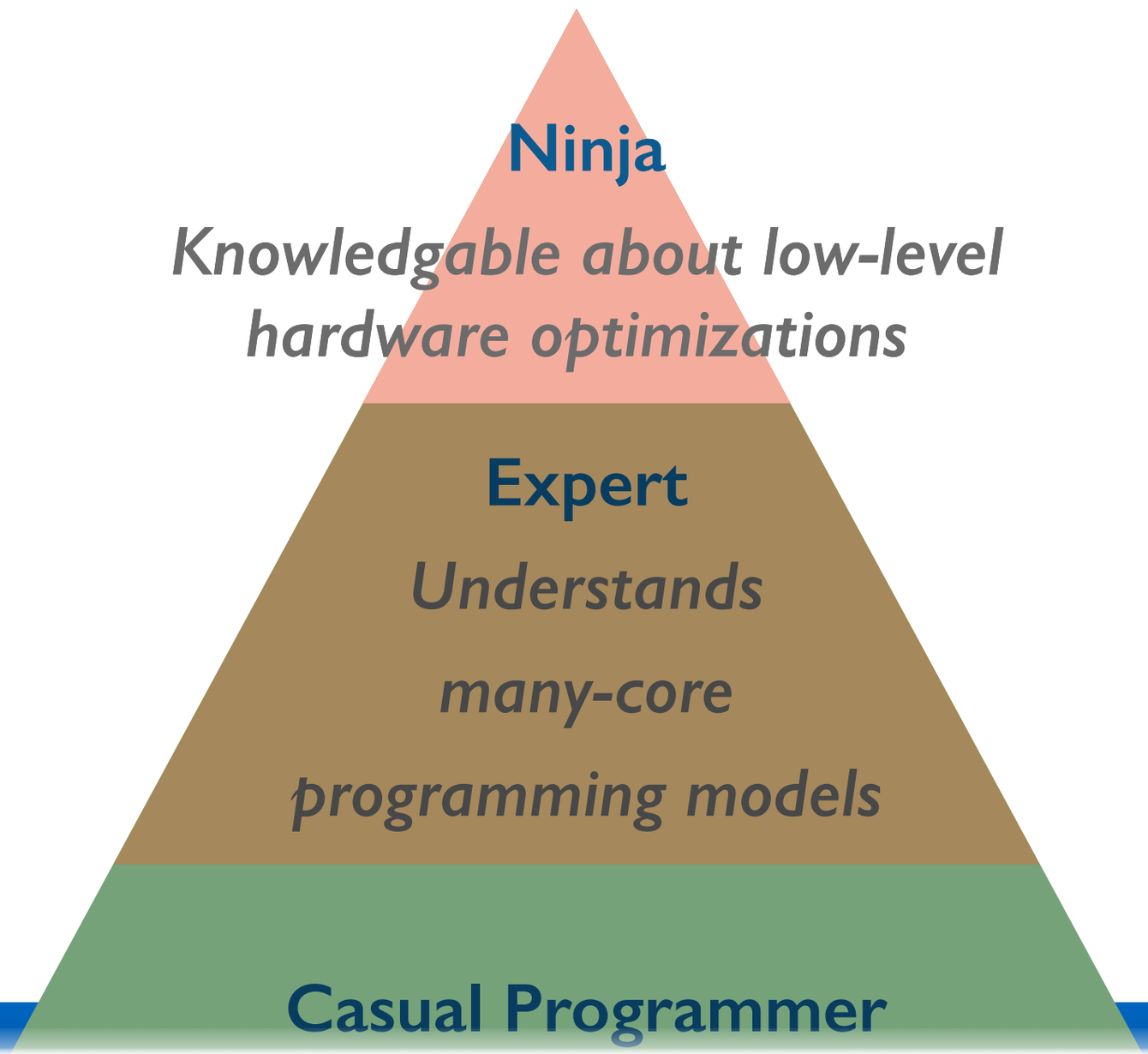
Programmer Expertise Hierarchy

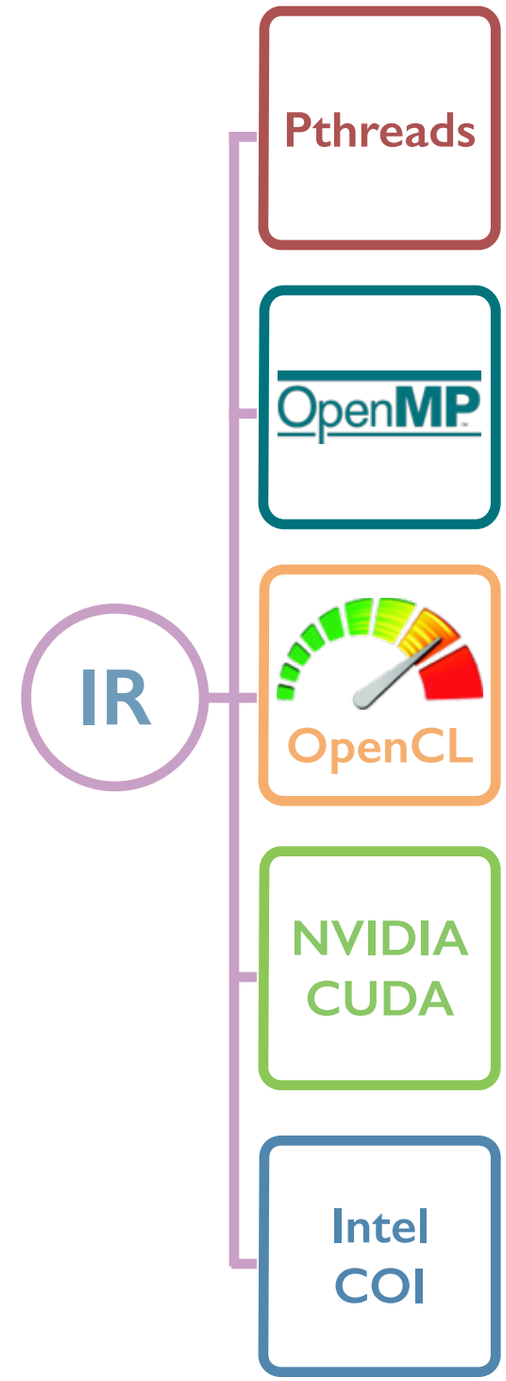




I chose to target developers in each proficiency layer

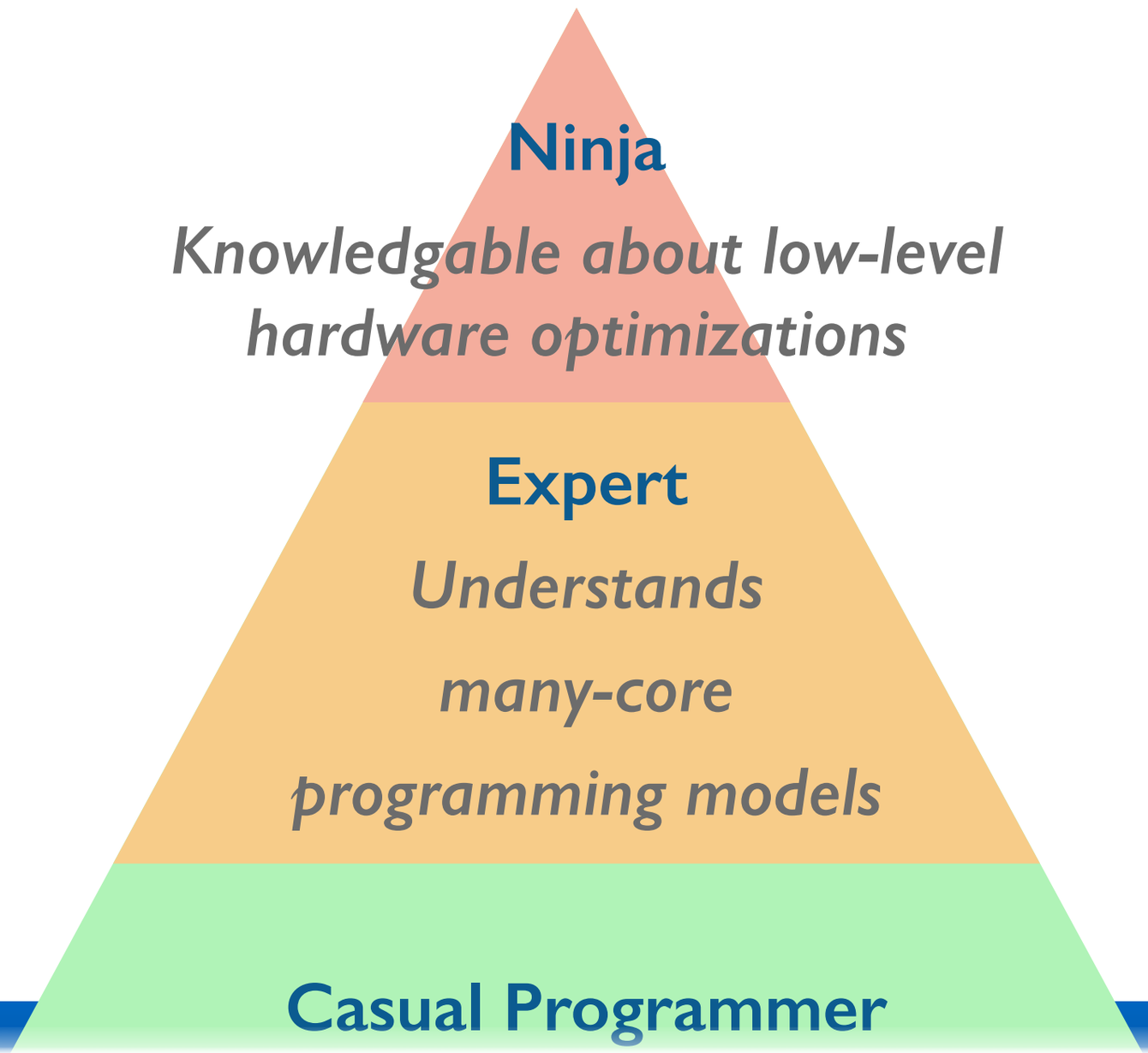
Programmer Expertise Hierarchy





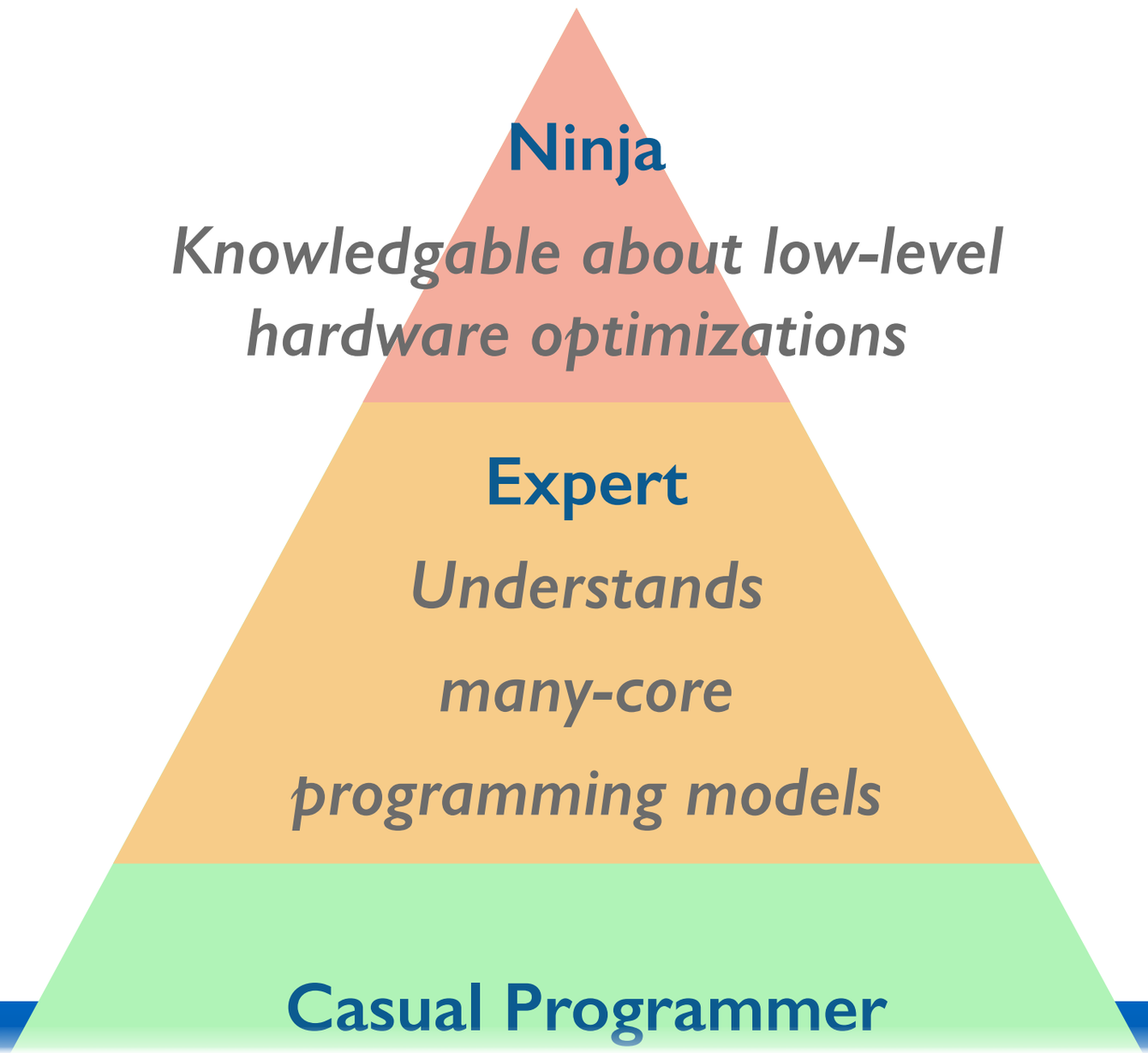
I chose to target developers in each proficiency layer

Programmer Expertise Hierarchy



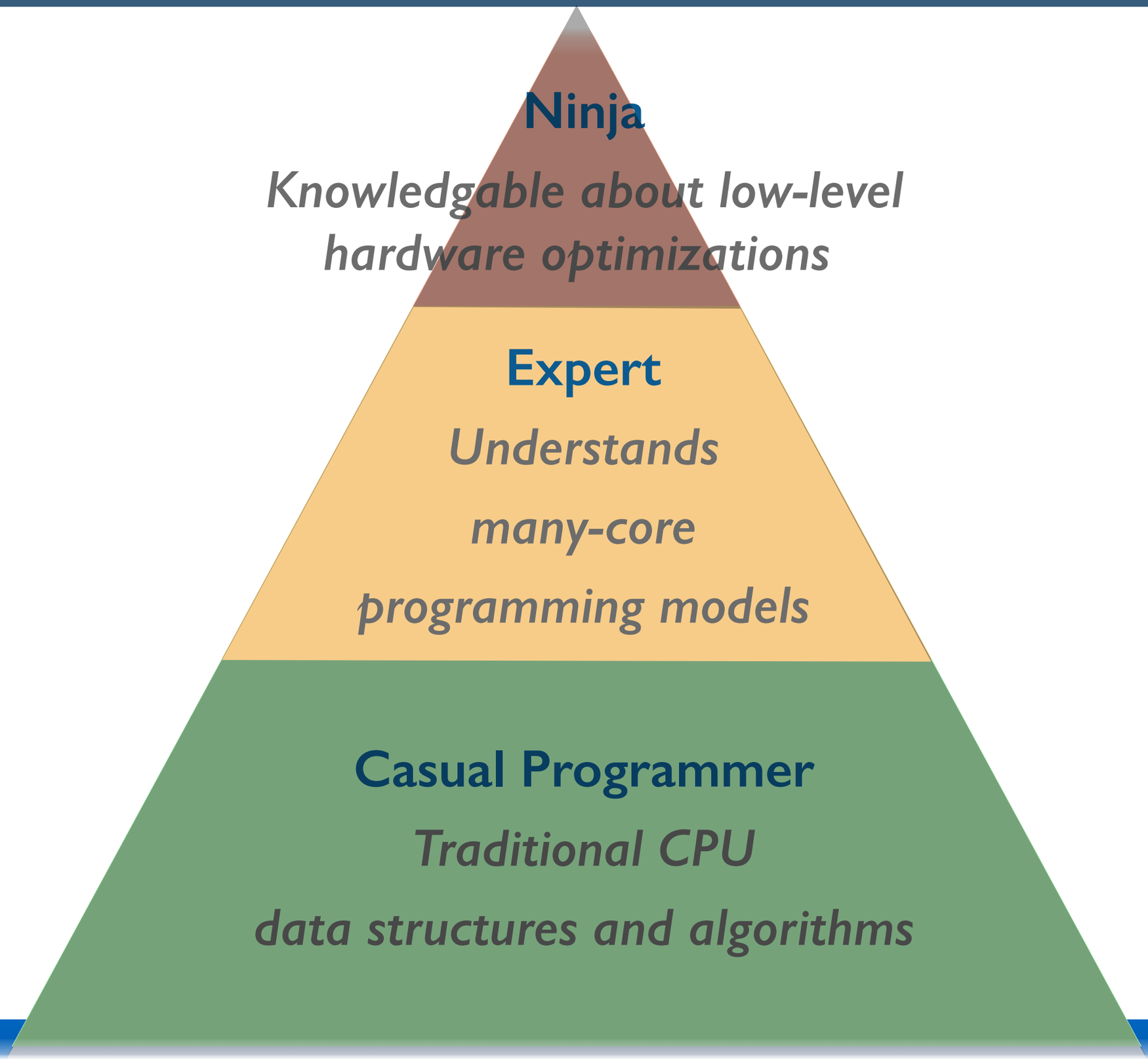
I chose to target developers in each proficiency layer

Programmer Expertise Hierarchy



I chose to target developers in each proficiency layer

Programmer Expertise Hierarchy

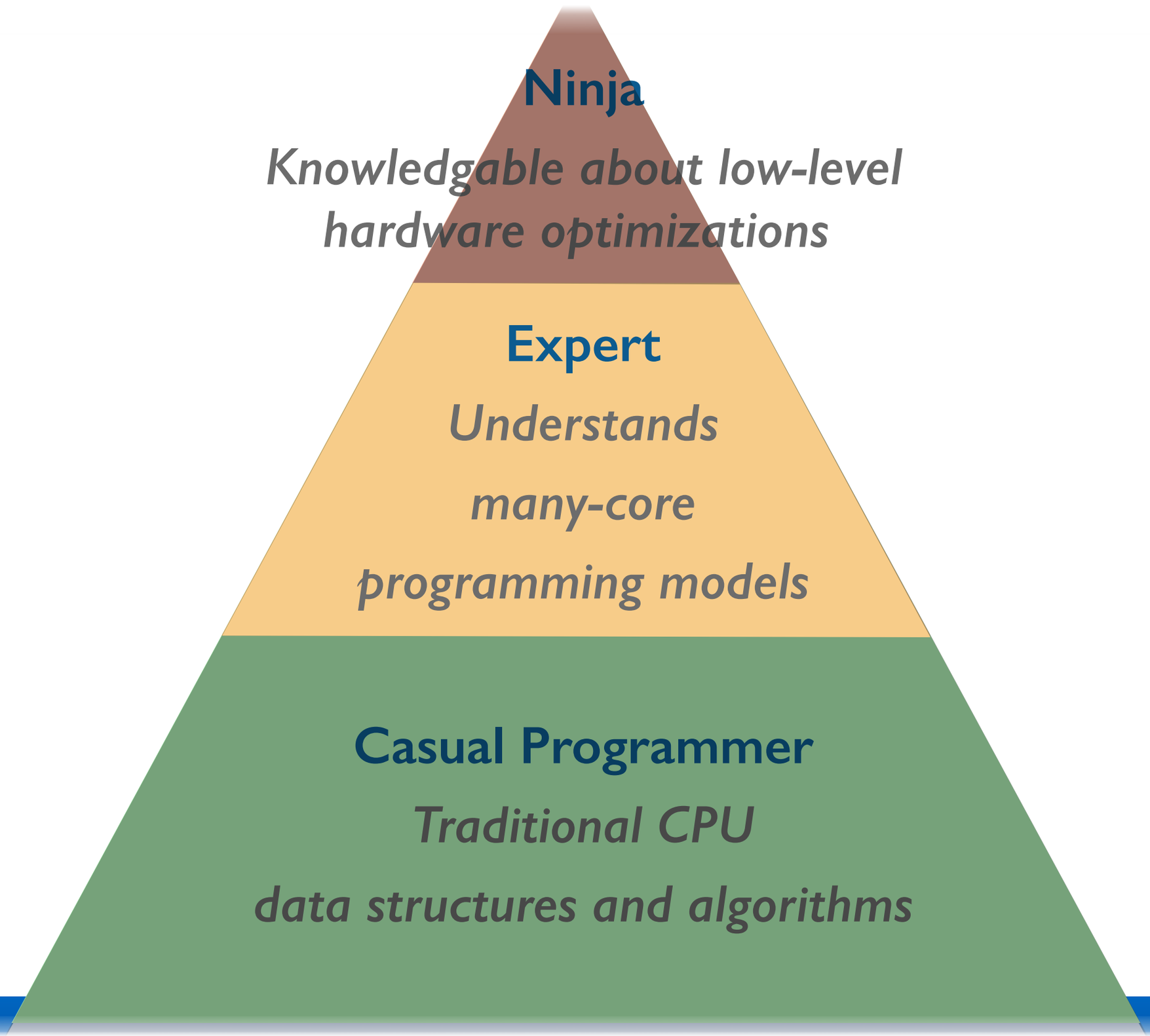


OKL

OFL

I chose to target developers in each proficiency layer

Programmer Expertise Hierarchy

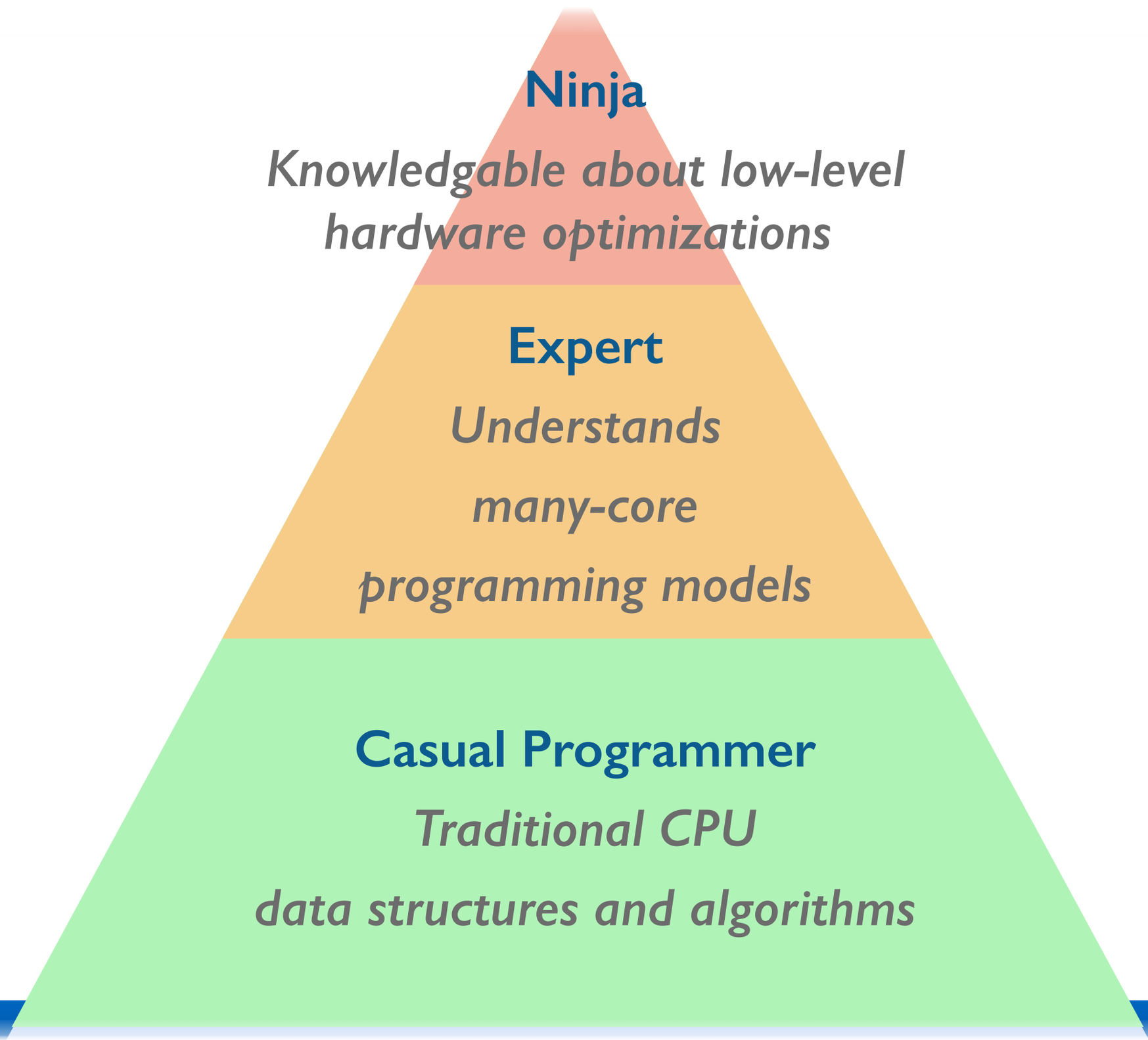


OKL

OFL

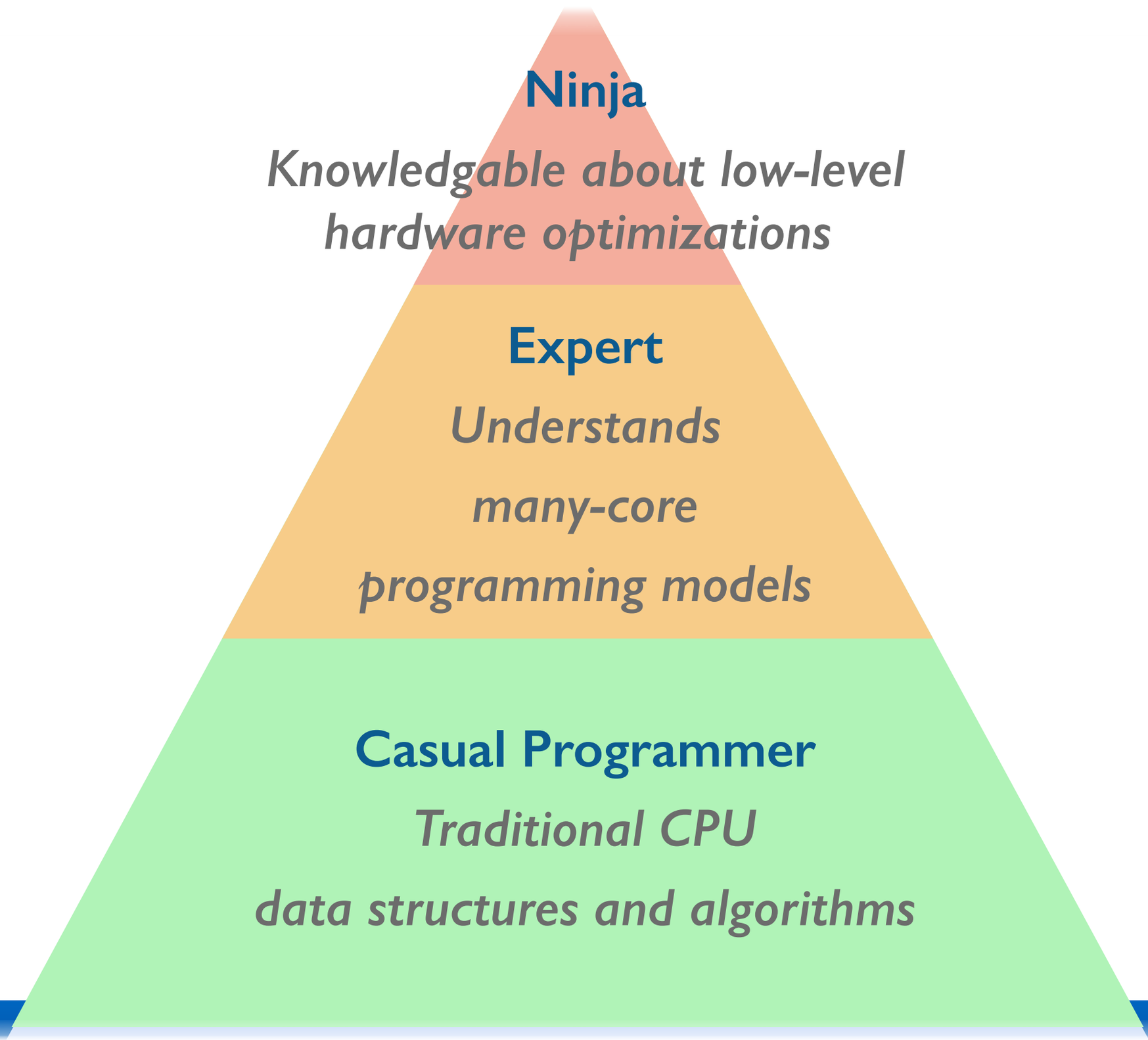
I chose to target developers in each proficiency layer

Programmer Expertise Hierarchy



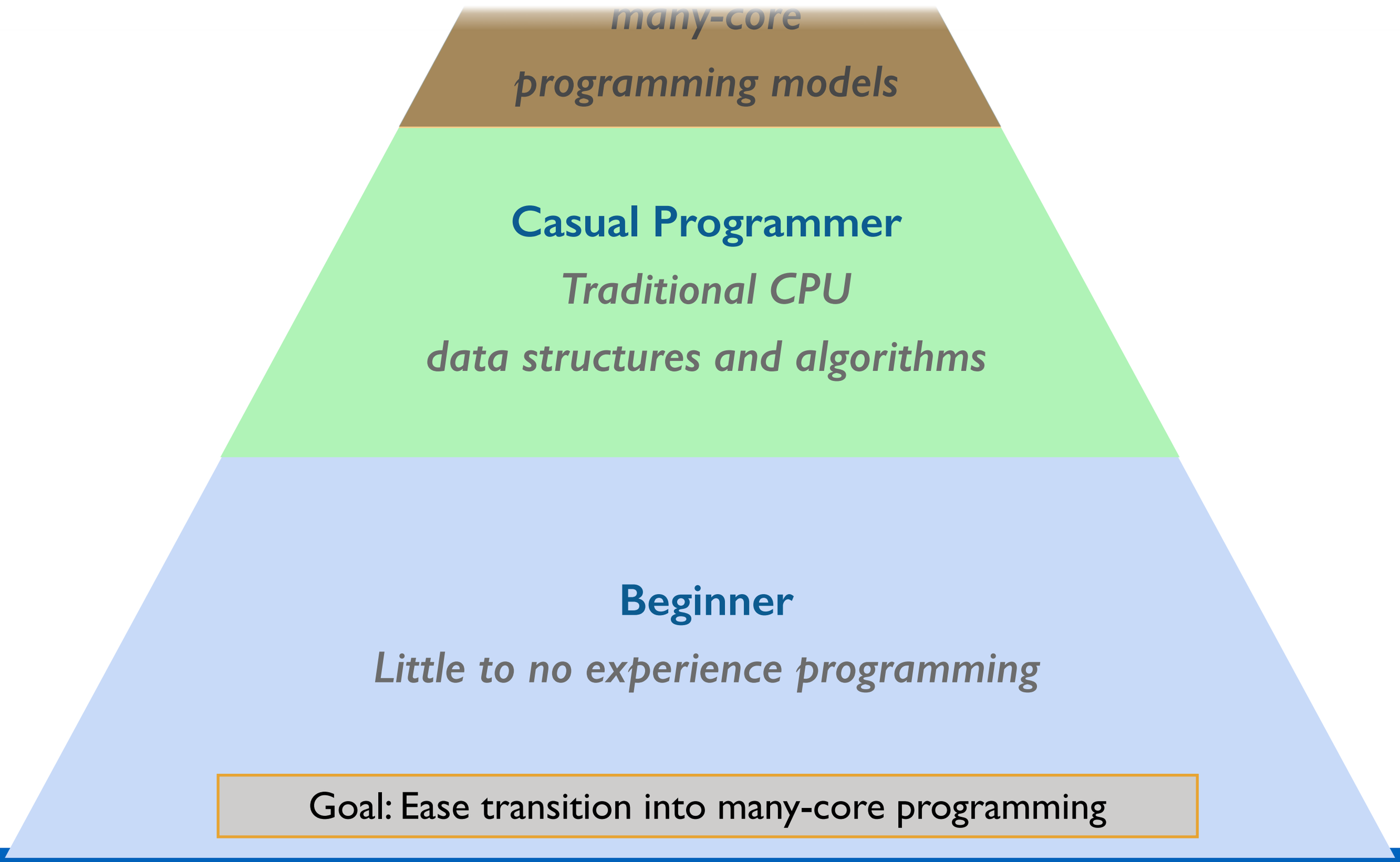
I chose to target developers in each proficiency layer

Programmer Expertise Hierarchy



I chose to target developers in each proficiency layer

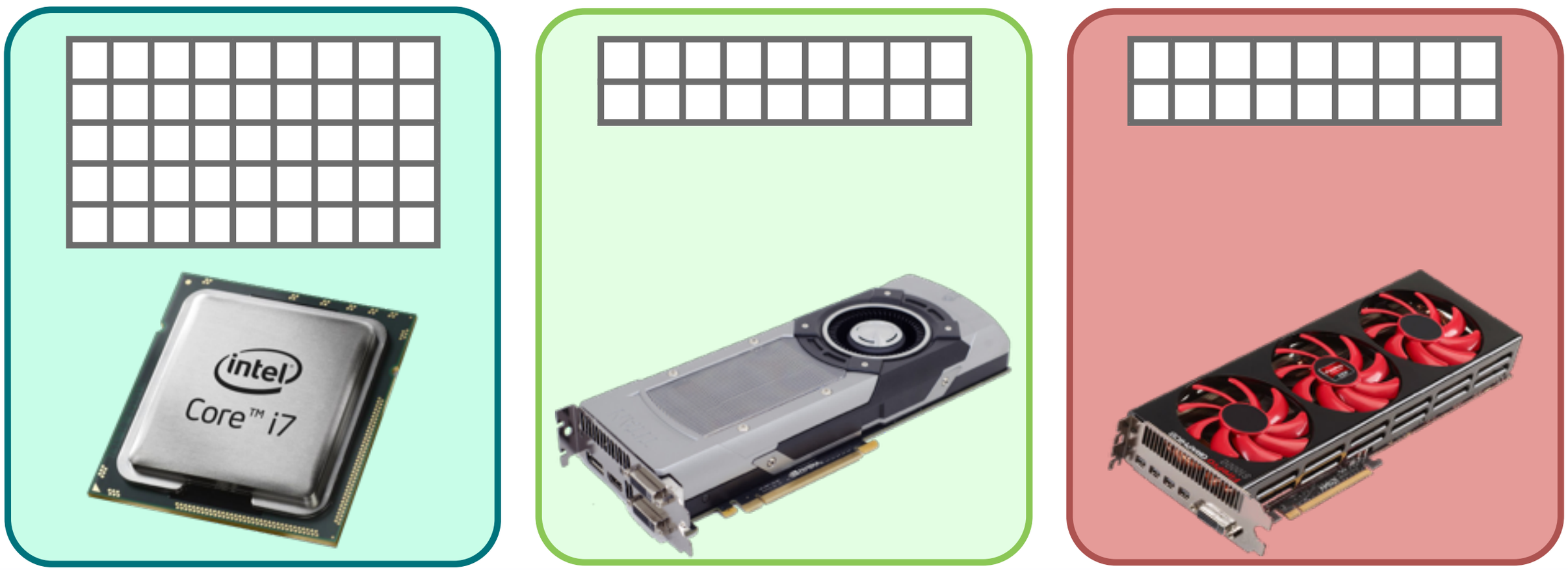
Programmer Expertise Hierarchy



Purpose of Automation Layers

Emulating UVA (Unified Virtual Address)

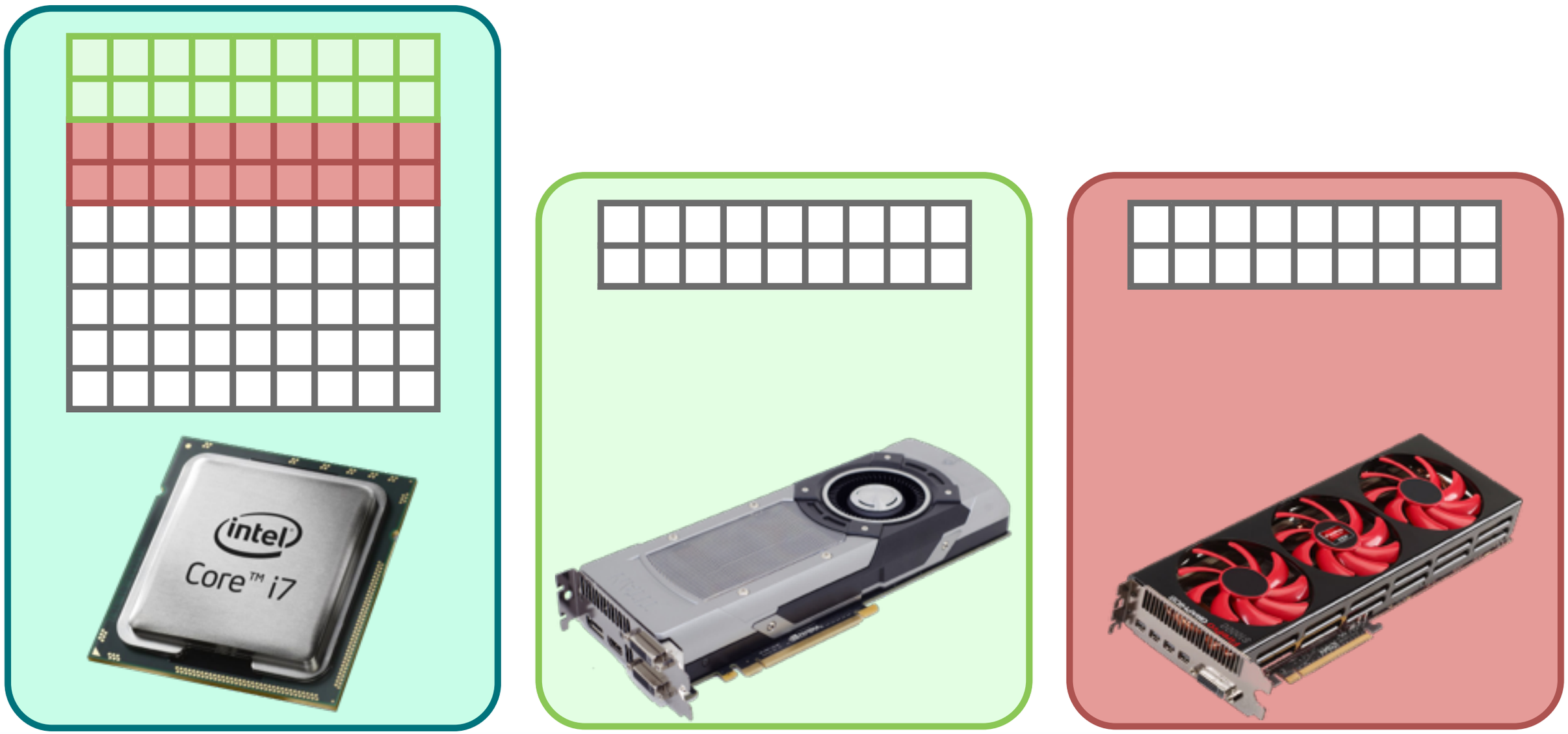
- Emulates one large memory addressing space across devices
- Hides device memory and data transfers



Purpose of Automation Layers

Emulating UVA (Unified Virtual Address)

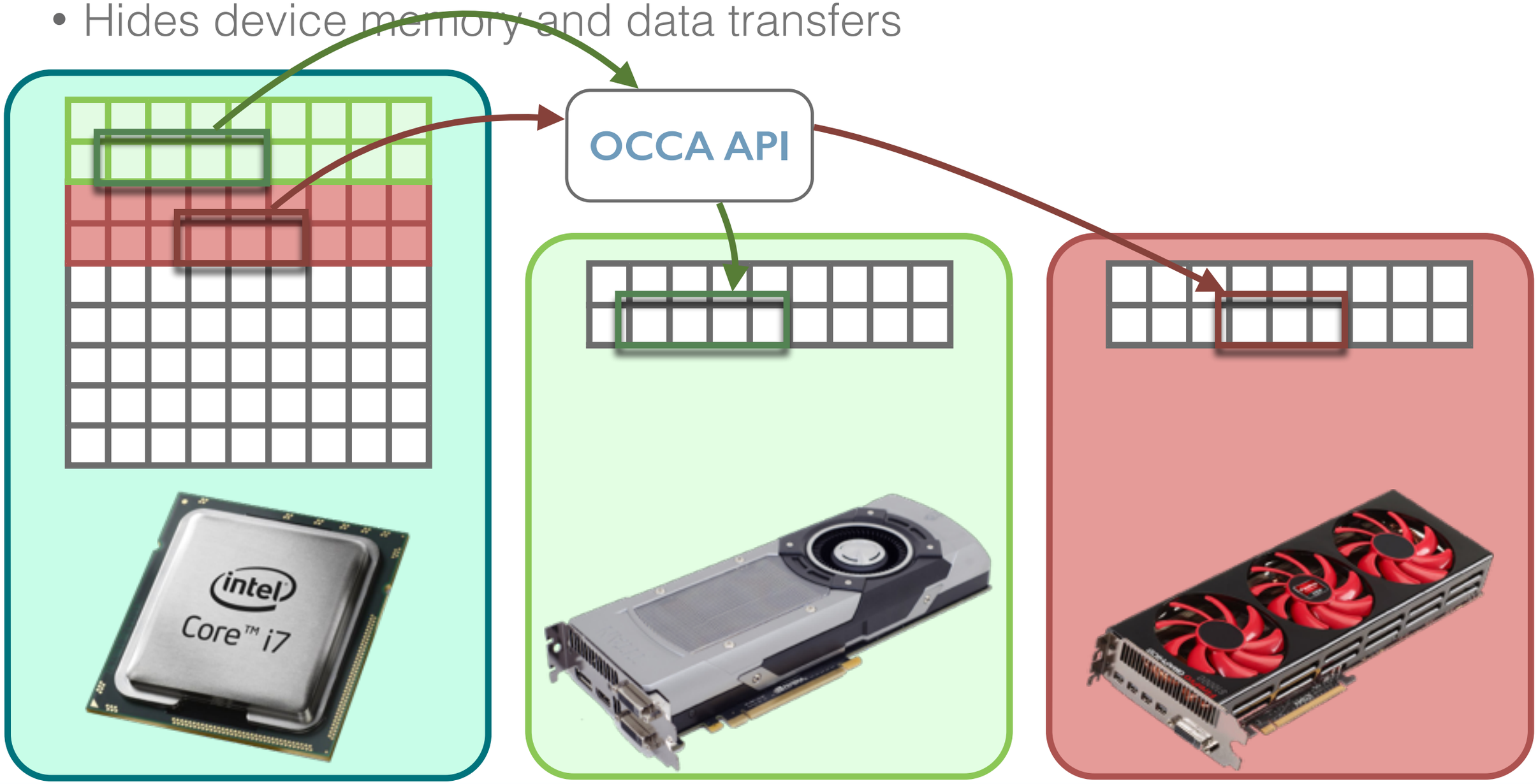
- Emulates one large memory addressing space across devices
- Hides device memory and data transfers



Purpose of Automation Layers

Emulating UVA (Unified Virtual Address)

- Emulates one large memory addressing space across devices
- Hides device memory and data transfers



OCCA API: Managed Memory

Automating Memory Transfers

- Eases introduction to heterogeneous computing
- Simplifies data movement
- Proper use of [`const`] and optional array descriptors for optimality

```
float *a  = new float[5];
float *b  = new float[5];
float *ab = new float[5];

occa::memory o_a, o_b, o_ab;

// o_a + 1 = ?
```

```
float *a  = (float*) device.managedUvaAlloc(5 * sizeof(float));
float *b  = (float*) device.managedUvaAlloc(5 * sizeof(float));
float *ab = (float*) device.managedUvaAlloc(5 * sizeof(float));
```

Additional API calls can be used to manually synchronize with device

OCCA API: Original

```
#include "occa.hpp"

int main(int argc, char **argv){
  occa::device device;
  occa::kernel addVectors;
  occa::memory o_a, o_b, o_ab;

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

  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;
  }

  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.okl",
                                            "addVectors");

  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';
```

https://github.com/tcew/OCCA2/tree/master/examples/addVectors

OCCA API: Original + UVA + Managed Memory

```
#include "occa.hpp"

int main(int argc, char **argv){
  occa::device device;
  occa::kernel addVectors;
  float *o_a, *o_b, *o_ab;

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

  float *a  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *b  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *ab = (float*) device.managedUvaAlloc(5 * sizeof(float));

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

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

  occa::memcpy(o_a, a, 5*sizeof(float));
  occa::memcpy(o_b, b, 5*sizeof(float));

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

  addVectors(5, a, b, ab);

  occa::finish() // occa::memcpy(ab, o_ab, 5*sizeof(float));

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

https://github.com/tcew/OCCA2/tree/master/examples/uvaAddVectors

OCCA API: Original + UVA + Managed Memory

```
#include "occa.hpp"

int main(int argc, char **argv){
  occa::device device;
  occa::kernel addVectors;

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

  float *a  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *b  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *ab = (float*) device.managedUvaAlloc(5 * sizeof(float));

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

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

  addVectors(5, a, b, ab);

  occa::finish()

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

https://github.com/tcew/OCCA2/tree/master/examples/uvaAddVectors

OCCA API: Original + Managed Memory + OAK

```
#include "occa.hpp"

int main(int argc, char
  occa::device device;
  occa::kernel addVecto

  device.setup("mode =

  float *a  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *b  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *ab = (float*) device.managedUvaAlloc(5 * sizeof(float));

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

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

  addVectors(5, a, b, ab);

  occa::finish()

  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 N = 0; N < entries; ++N)
    ab[N] = a[N] + b[N];
}
```

OAK

https://github.com/tcew/OCCA2/tree/master/examples/uvaAddVectors

OCCA API: Original + Managed Memory + OAK

```
#include "occa.hpp"

int main(int argc, char
  occa::device device;
  occa::kernel addVecto

  device.setup("mode =

  float *a  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *b  = (float*) device.managedUvaAlloc(5 * sizeof(float));
  float *ab = (float*) device.managedUvaAlloc(5 * sizeof(float));

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

  addVectors = device.l


  addVectors(5, a, b, ab);

  occa::finish()

  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 N = 0; N < entries; ++N)
    ab[N] = a[N] + b[N];
}
```

OAK

```
kernel void addVectors(const int entries,
                       const float *a,
                       const float *b,
                       float *ab){
  for(int N = 0; N < entries; ++N; tile(16)){
    if(N < entries)
      ab[N] = a[N] + b[N];
  }
}
```

OKL

https://github.com/tcew/OCCA2/tree/master/examples/uvaAddVectors

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  for(int z = r; z < (pointsZ - r); ++z){
    for(int y = r; y < (pointsY - r); ++y){
      for(int x = r; x < (pointsX - r); ++x){
        float lap = 0;

        for(int i = -r; i <= r; ++i)
          lap += w[i]*(u0[(z  )*pointsXY + (y  )*pointsX + (x+i)] +
                       u0[(z  )*pointsXY + (y+i)*pointsX + (x  )] +
                       u0[(z+i)*pointsXY + (y  )*pointsX + (x  )]);

        const int id = (z*pointsXY + y*pointsX + x);

        u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }

  // ...
}
```

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  fdKernel = device.buildKernelFromSource("fdKernel.oak",
                                          "fdKernel");

  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float *u0,
                     float *u1,
                     float *u2,
                     float *w,
                     float *c2,
                     float dt2){

  for(int z = r; z < (pointsZ - r); ++z){
    for(int y = r; y < (pointsY - r); ++y){
      for(int x = r; x < (pointsX - r); ++x){
        float lap = 0;

        for(int i = -r; i <= r; ++i)
          lap += w[i]*(u0[(z  )*pointsXY + (y  )*pointsX + (x+i)] +
                       u0[(z  )*pointsXY + (y+i)*pointsX + (x  )] +
                       u0[(z+i)*pointsXY + (y  )*pointsX + (x  )]);

        const int id = (z*pointsXY + y*pointsX + x);

        u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }
}
```

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*pointsY));
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);

  fdKernel = device.buildKernelFromSource("fdKernel.oak",
                                          "fdKernel");
                                          fdKInfo);

  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
*u0,
*u1,
*u2,
*w,
*c2,
dt2){

sZ - r); ++z){
ntsY - r); ++y){
ointsX - r); ++x){


= r; ++i)
  lap += w[i]*(u0[(z  )*pointsXY + (y  )*pointsX + (x+i)] +
               u0[(z  )*pointsXY + (y+i)*pointsX + (x  )] +
               u0[(z+i)*pointsXY + (y  )*pointsX + (x  )]);

  const int id = (z*pointsXY + y*pointsX + x);

  u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }
}
```

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*poin
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);


  fdKernel = device.buildKernelFromSource("
                                           "
                                           f


  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float *u0,
                     float *u1,
                     float *u2,
                     float *w,
                     float *c2,
                     float dt2){

  for(int z = 4; z < (100 - 4); ++z){
    for(int y = 4; y < (100 - 4); ++y){
      for(int x = 4; x < (100 - 4); ++x){
        float lap = 0;

        for(int i = -4; i <= 4; ++i)
          lap += w[i]*(u0[(z  )*10000 + (y  )*100 + (x+i)] +
                       u0[(z  )*10000 + (y+i)*100 + (x  )] +
                       u0[(z+i)*10000 + (y  )*100 + (x  )]);

        const int id = (z*10000 + y*100 + x);

        u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }
}
```

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*poin
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);


  fdKernel = device.buildKernelFromSource("


  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float *u0,
                     float *u1,
                     float *u2,
                     float *w,
                     float *c2,
                     float dt2){

  for(int z = 4; z < (100 - 4); ++z){
    for(int y = 4; y < (100 - 4); ++y){
      for(int x = 4; x < (100 - 4); ++x){
        float lap = 0;

        for(int i = -4; i <= 4; ++i)
          lap += w[i]*(u0[(z  )*10000 + (y  )*100 + (x+i)] +
                       u0[(z  )*10000 + (y+i)*100 + (x  )] +
                       u0[(z+i)*10000 + (y  )*100 + (x  )]);

        const int id = (z*10000 + y*100 + x);

        u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }
}
```

```
float * restrict u0,
float * restrict u1,
float * restrict u2,
float * restrict w,
float * restrict c2
```

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*poin
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);


  fdKernel = device.buildKernelFromSource("
                                           f


  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float * restrict u0,
                     float * restrict u1,
                     float * restrict u2,
                     float * restrict w,
                     float * restrict c2,
                     float dt2){

  for(int z = 4; z < (100 - 4); ++z){
    for(int y = 4; y < (100 - 4); ++y){
      for(int x = 4; x < (100 - 4); ++x){
        float lap = 0;

        for(int i = -4; i <= 4; ++i){
          lap += w[i]*(u0[(z  )*10000 + (y  )*100 + (x+i)] +
                       u0[(z  )*10000 + (y+i)*100 + (x  )] +
                       u0[(z+i)*10000 + (y  )*100 + (x  )]);

        const int id = (z*10000 + y*100 + x);

        u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }
}
```

Bounds: [4, 96)
Stride: 1

Bounds: [-4, 4]
Stride: 1

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*poin
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);


  fdKernel = device.buildKernelFromSou



  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float * restrict u0,
                     float * restrict u1,
                     float * restrict u2,
                     float * restrict w,
                     float * restrict c2,
                     float dt2){

  for(int z = 4; z < (100 - 4); ++z){
    for(int y = 4; y < (100 - 4); ++y){
      < (100 - 4); ++x){


      ; i <= 4; ++i)
      (u0[(z  )*10000 + (y  )*100 + (x+i)] +
       u0[(z  )*10000 + (y+i)*100 + (x  )] +
       u0[(z+i)*10000 + (y  )*100 + (x  )]);

      = (z*10000 + y*100 + x);

      c2[id]*lap + u0[id] + u1[id];


    }
  }
}
```

```
Iterator [Stride]
    z    [1000]
    y    [100]
  (x+i)  [1]
```

```
Iterator [Stride]
    z    [1000]
  (y+i)  [100]
    x    [1]
```

```
Iterator [Stride]
  (z+i)  [1000]
    y    [100]
    x    [1]
```

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*poin
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);


  fdKernel = device.buildKernelFromSou



  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float * restrict u0,
                     float * restrict u1
                     float * res
                     float * res
                     float * res
                     float dt2){

                     (100 - 4);
    for(int y = 4; y < (100 - 4); ++y){
                     < (100 -

                     ; i <= 4;
                     (u0[(z  )*
                      u0[(z  )*
                      u0[(z+i)*10000 + (y  )*100 + (x  )]);

                     = (z*10000

                     c2[id]*lap

  }
}
```

Iterator	[Stride]
z	[10000]
y	[100]
x	[1]
i	[1]

Bounds	(Stride)
[40000,960000]	(10000)
[400 ,9600]	(100)
[0 ,100]	(1)

Iterator	[Stride]
z	[10000]
y	[100]
i	[100]
x	[1]

Bounds	(Stride)
[40000,960000]	(10000)
[0 ,10000]	(100)
[4 ,96]	(1)

Iterator	[Stride]
z	[10000]
i	[10000]
y	[100]
x	[1]

Bounds	(Stride)
[0 ,1000000]	(10000)
[400 ,9600]	(100)
[4 ,96]	(1)

OCCA Automagic Kernel (OAK)

```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*poin
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);


  fdKernel = device.buildKernelFromSource


  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float * restrict u0,
                     * restrict u1,
                     * restrict u2,
                     * restrict w,
                     * restrict c2,
                     dt2){

  for(int z = 4; z < (1
    for(int y = 4;
      for(int x = 4
        float lap =

        for(int i =
          lap += w[i]*(u0[(z   )*10000 + (y   )*100 + (x+i)] +
                       u0[(z   )*10000 + (y+i)*100 + (x   )] +
                       u0[(z+i)*10000 + (

        const int id = (z*10000 + y*100 +

        u2[id] = -dt2*c2[id]*lap + u0[id
      }
    }
  }
}
```

```
Iterator [Stride]
    id   [1]
```

```
Iterator [Stride]
    z    [10000]
    y    [100]
    x    [1]
```

Bounds	(Stride)
[40000,960000]	(10000)
[400 ,9600]	(100)
[4 ,96]	(1)

OCCA Automagic Kernel (OAK)

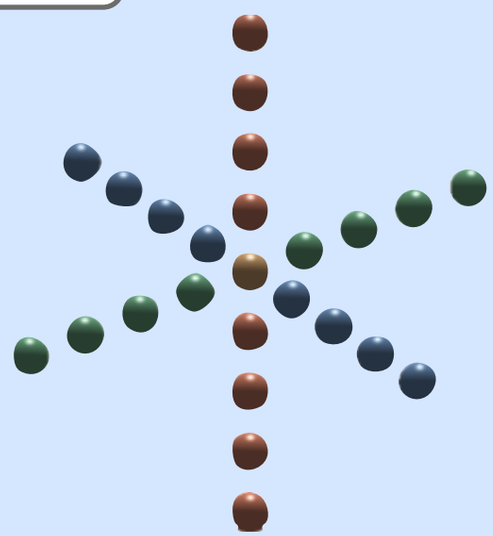
```
#include "occa.hpp"

int main(int argc, char **argv){
  // ...

  occa::kernelInfo fdKInfo;
  fdKInfo.addDefine("pointsX" , pointsX);
  fdKInfo.addDefine("pointsY" , pointsY);
  fdKInfo.addDefine("pointsXY", (pointsX*poin
  fdKInfo.addDefine("pointsZ" , pointsZ);
  fdKInfo.addDefine("r"       , r);


  fdKernel = device.buildKernelFromSource("
                                           "
                                           f


  fdKernel(u0, u1, u2, w, c, dt2);

  // ...
}
```

```
kernel void fdKernel(float * restrict u0,
                     float * restrict u1,
                     float * restrict u2,
                     float * restrict w,
                     float * restrict c2,
                     float dt2){

  for(int z = 4; z < (100 - 4); ++z){
    for(int y = 4; y < (100 - 4); ++y){
      for(int x = 4; x < (100 - 4); ++x){
        float lap = 0;

        for(int i = -4; i <= 4; ++i)
          lap += w[i]*(u0[(z  )*10000 + (y  )*100 + (x+i)] +
                       u0[(z  )*10000 + (y+i)*100 + (x  )] +
                       u0[(z+i)*10000 + (y  )*100 + (x  )]);

        const int id = (z*10000 + y*100 + x);

        u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }
}
```

Results

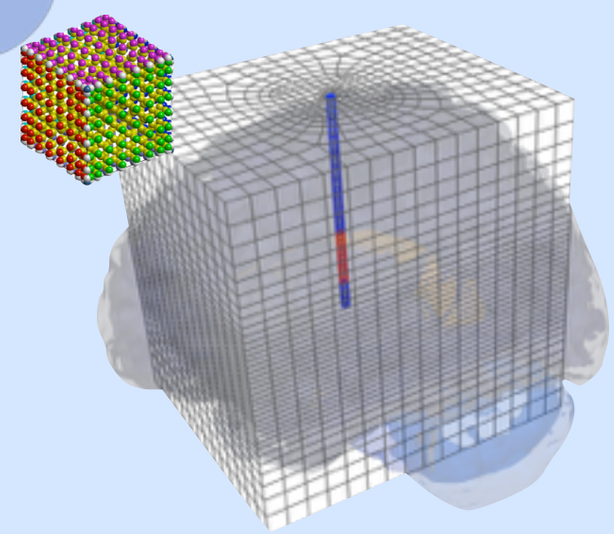
OCCA2 Applications

IR



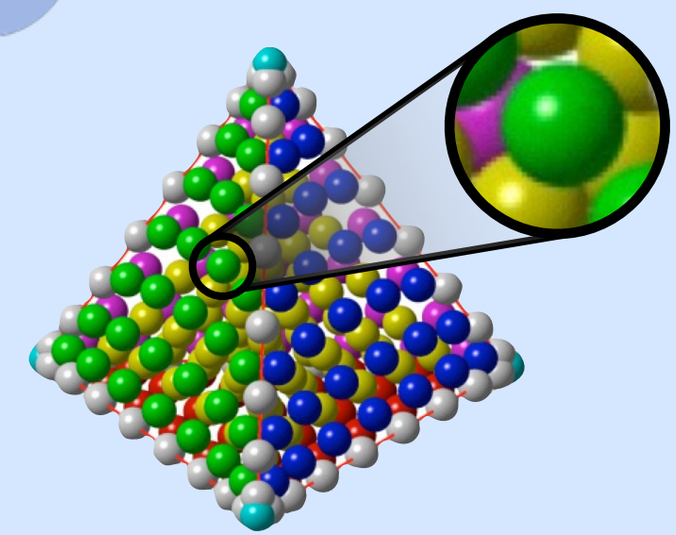
High-order Finite Difference

OKL



Spectral Element Method

OKL



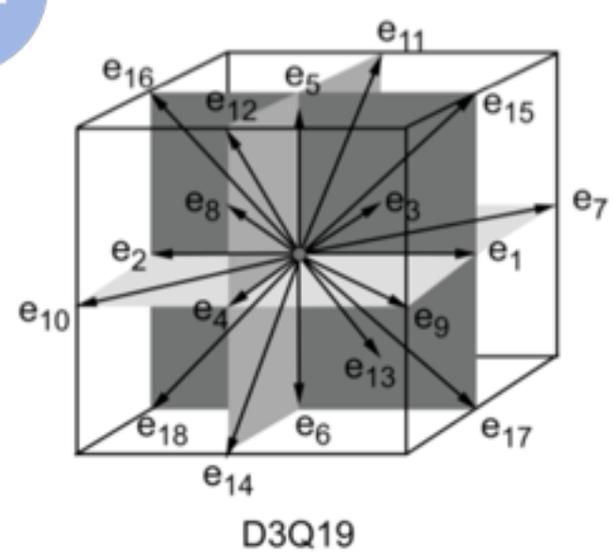
Discontinuous Galerkin

OKL

```
XSBench

Version 13

Contact Information

Organization:     Center for Exascale Simulation of Advanced Reactors (CESAR)
                  Argonne National Laboratory

Development Lead: John Tramm <jtramm@mcs.anl.gov>

What is XSBench?

XSBench is a mini-app representing a key computational kernel of the
Monte Carlo neutronics application OpenMC.

A full explanation of the theory and purpose of XSBench is provided in
docs/XSBench_Theory.pdf.

Quick Start Guide
```

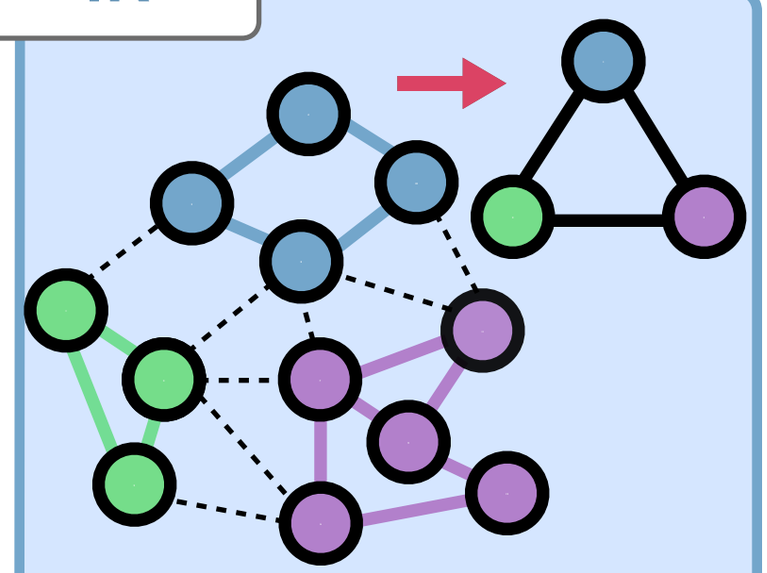
Monte Carlo

OKL



Lattice Boltzmann Method

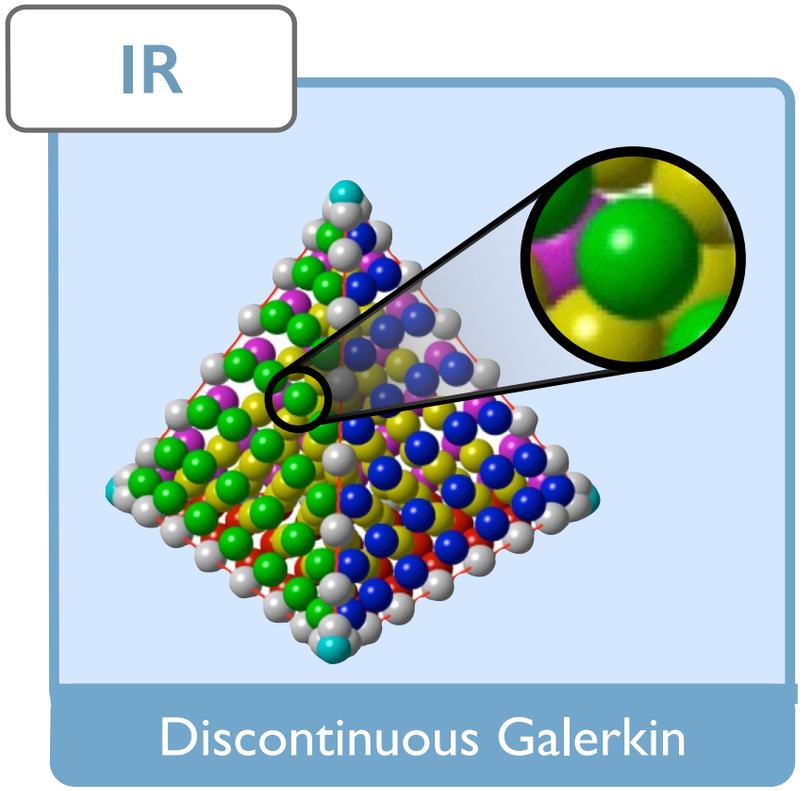
IR

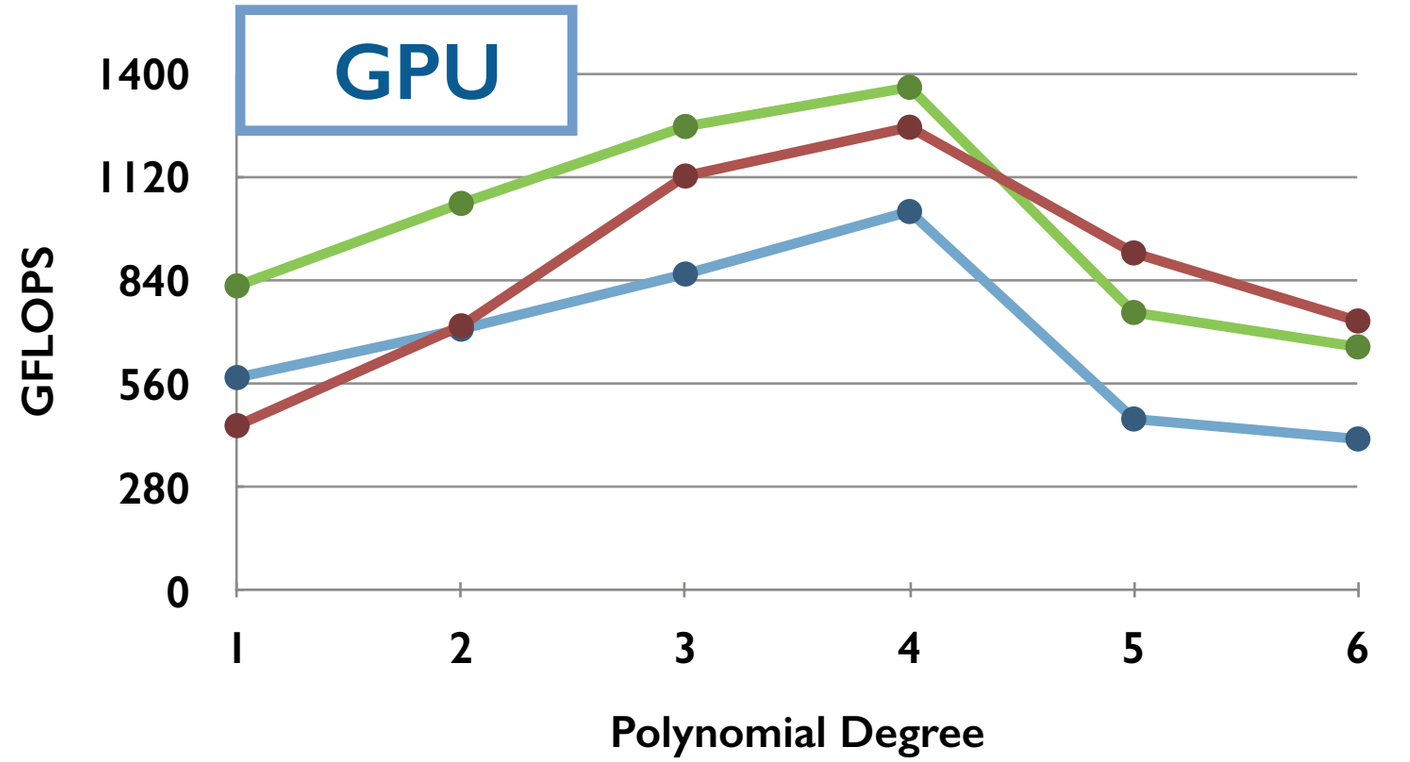


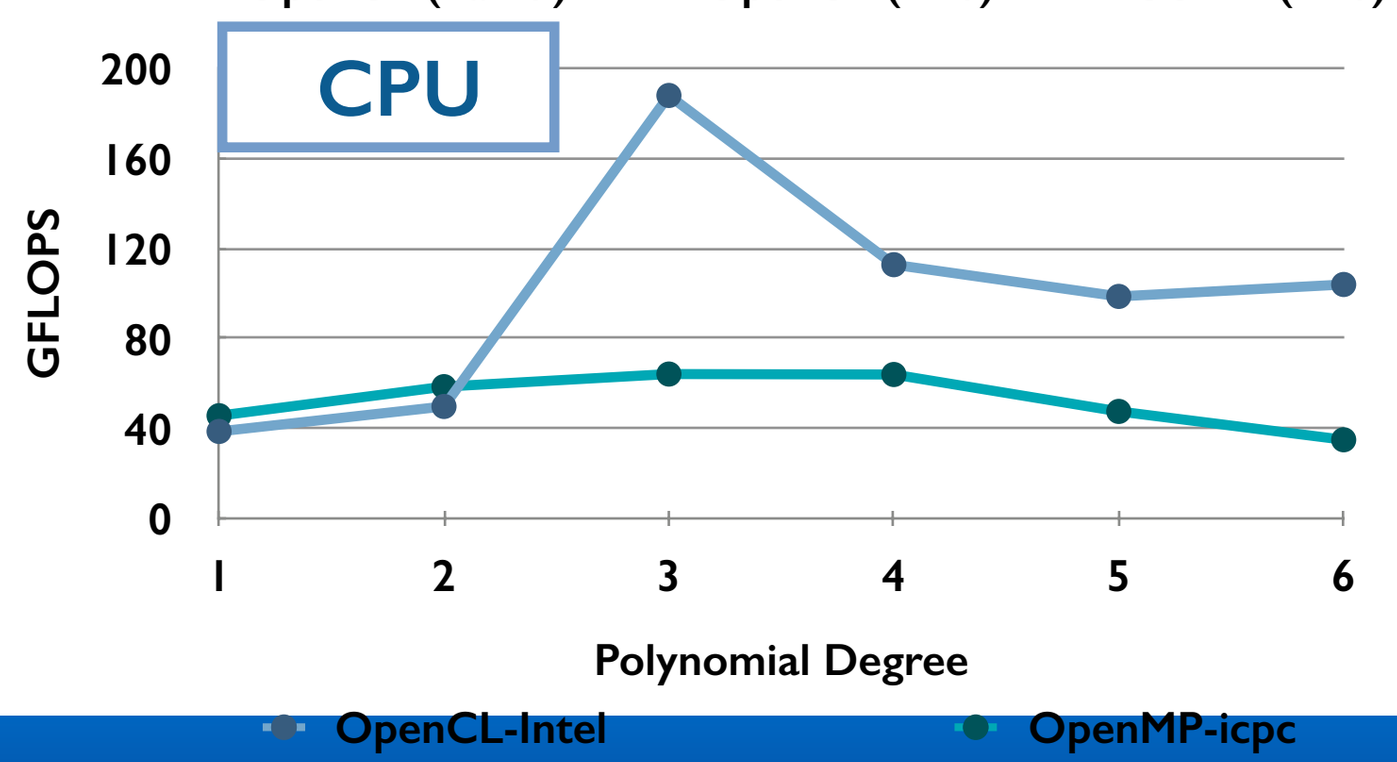
Algebraic Multigrid

A broad range of applications have been implemented using OCCA

OCCA2 Applications: Discontinuous Galerkin

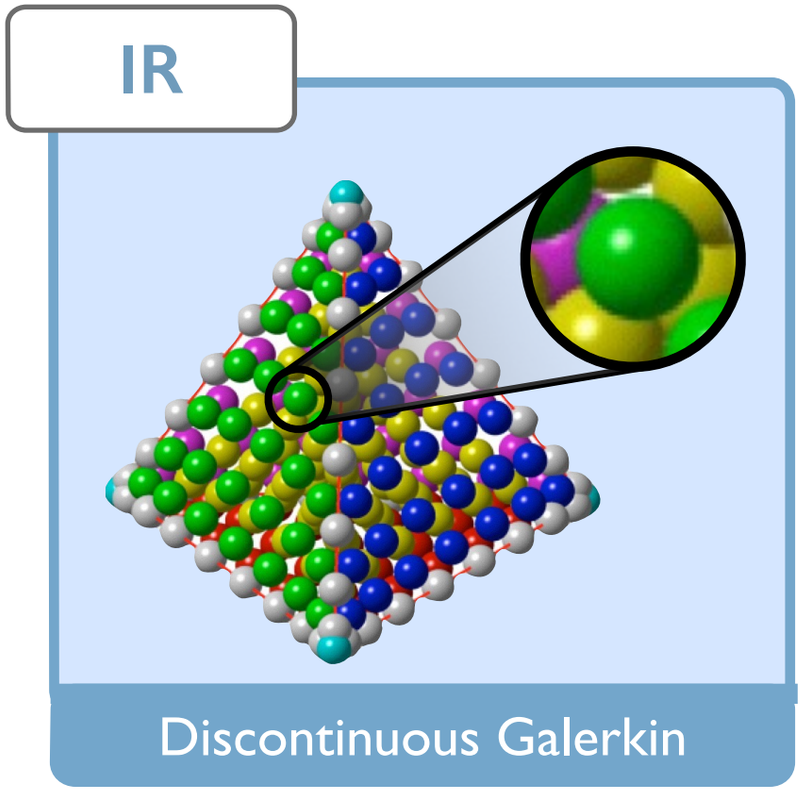


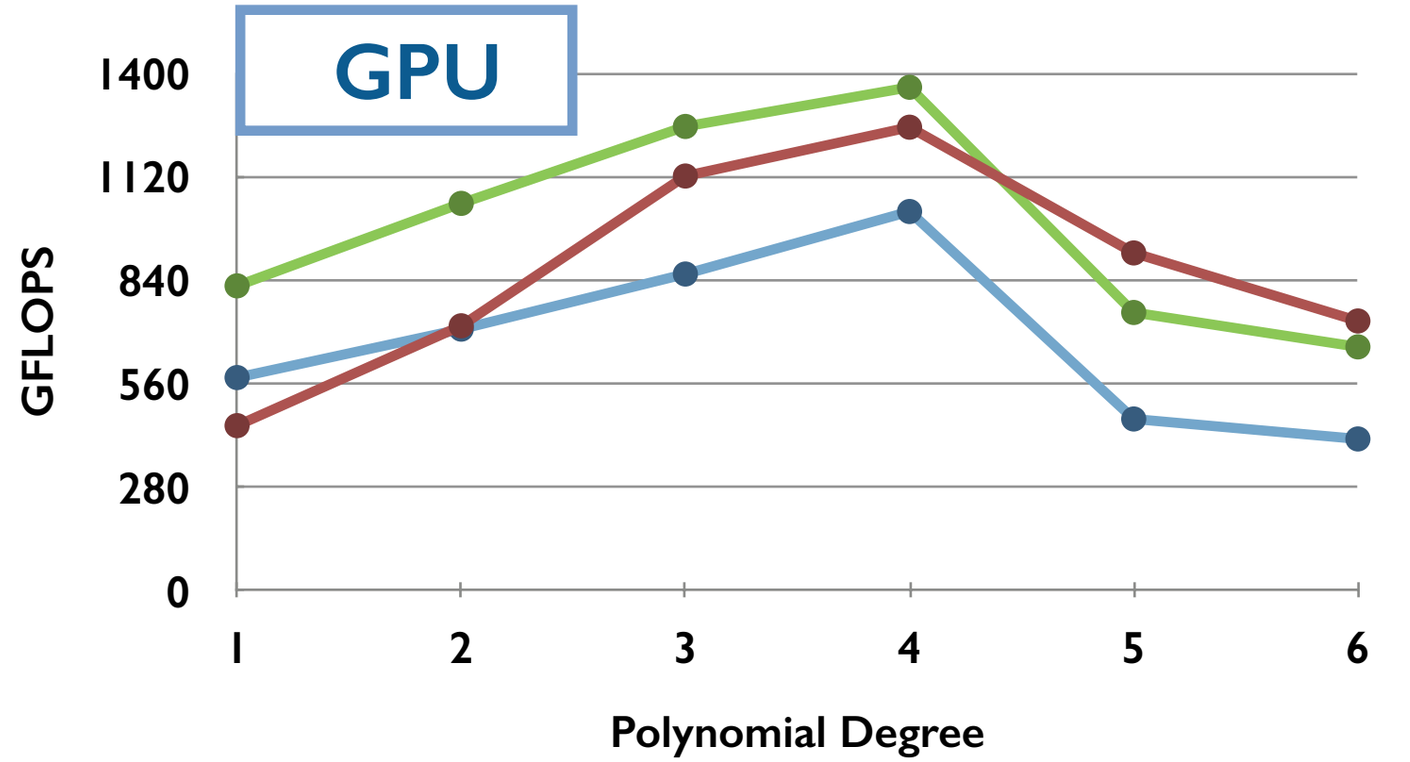


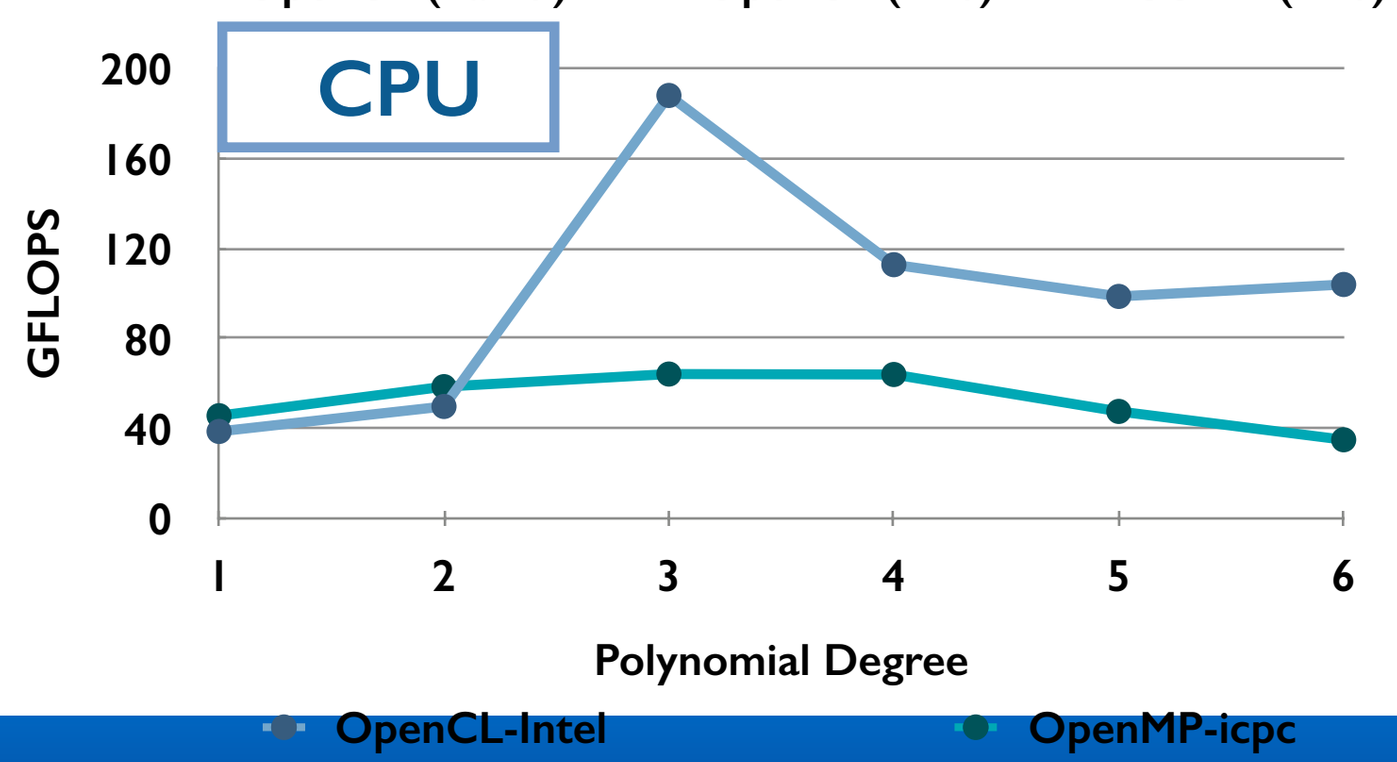


(Gandham, R., Medina, D. and Warburton, T. 2015)

OCCA2 Applications: Discontinuous Galerkin

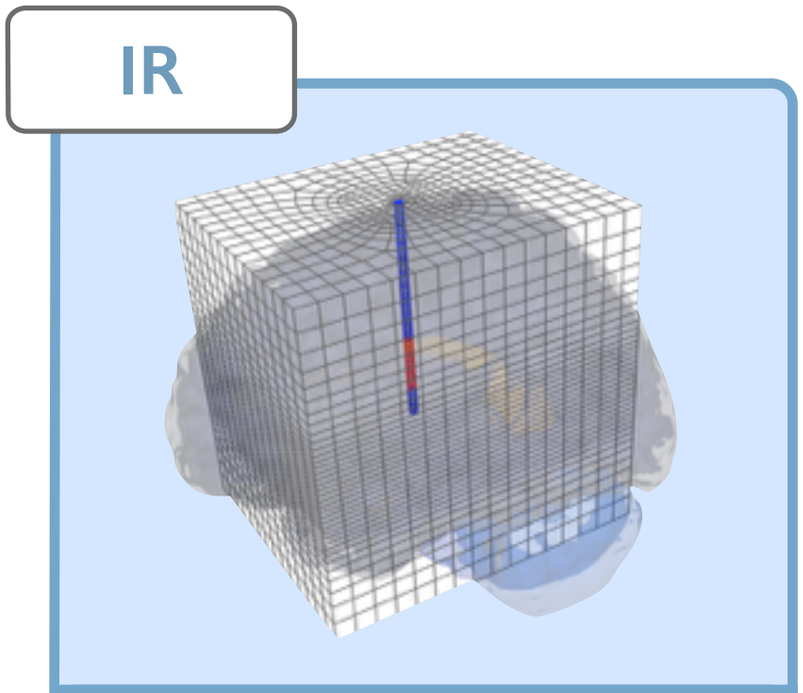




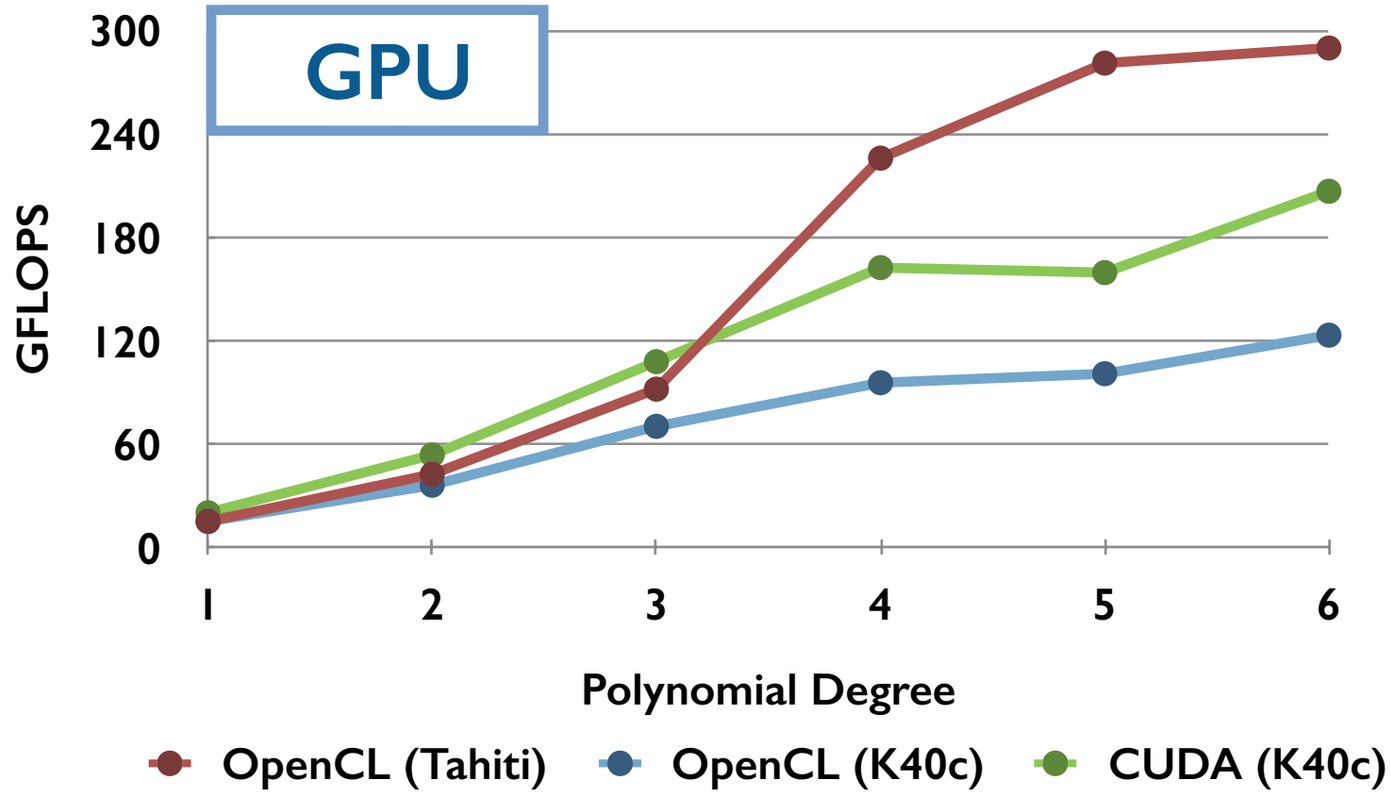


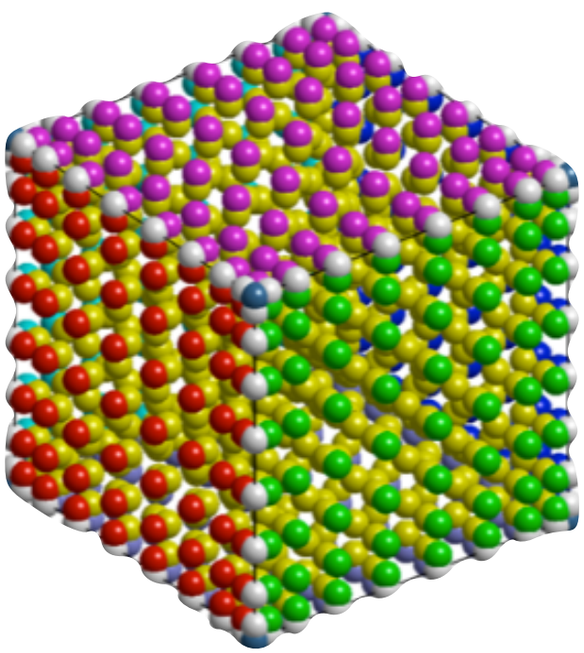
(Gandham, R., Medina, D. and Warburton, T. 2015)

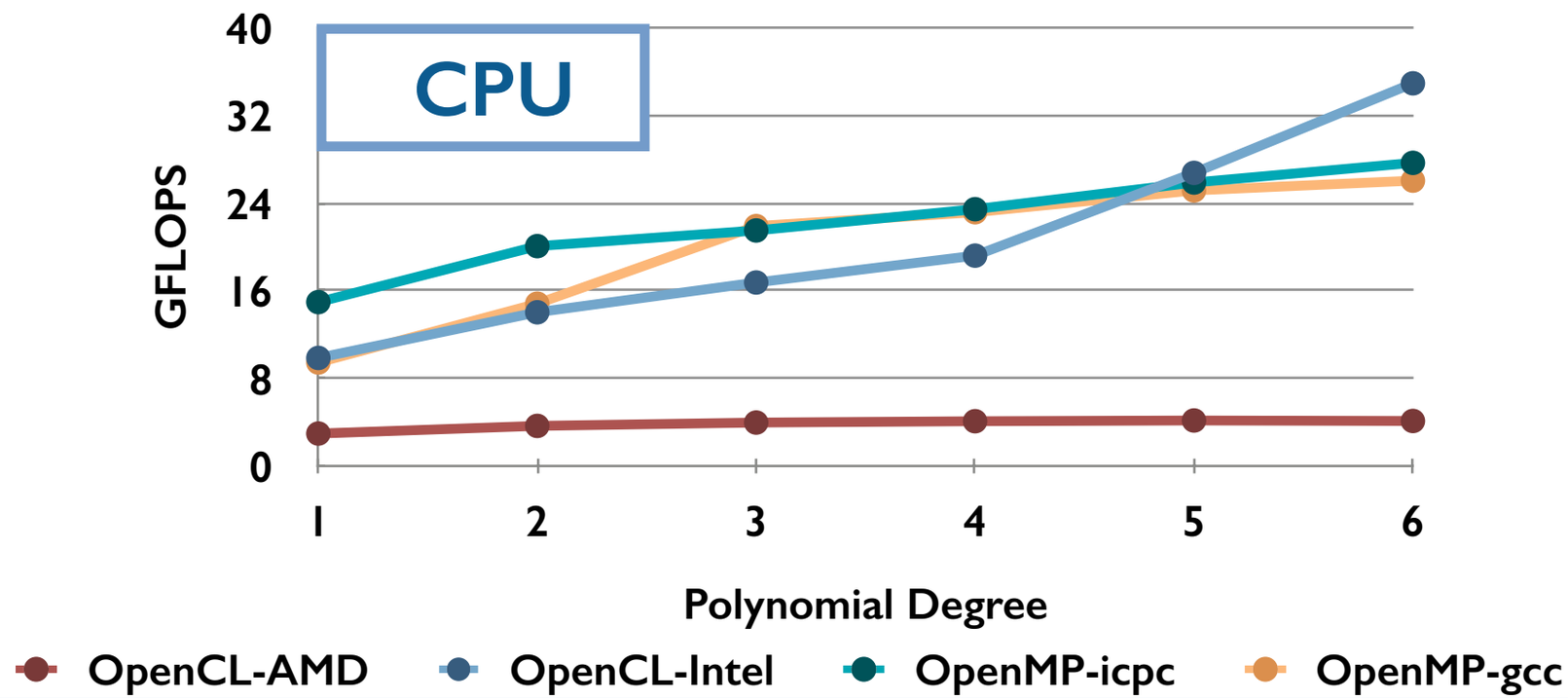
OCCA2 Applications: Spectral Element Method



Spectral Element Method



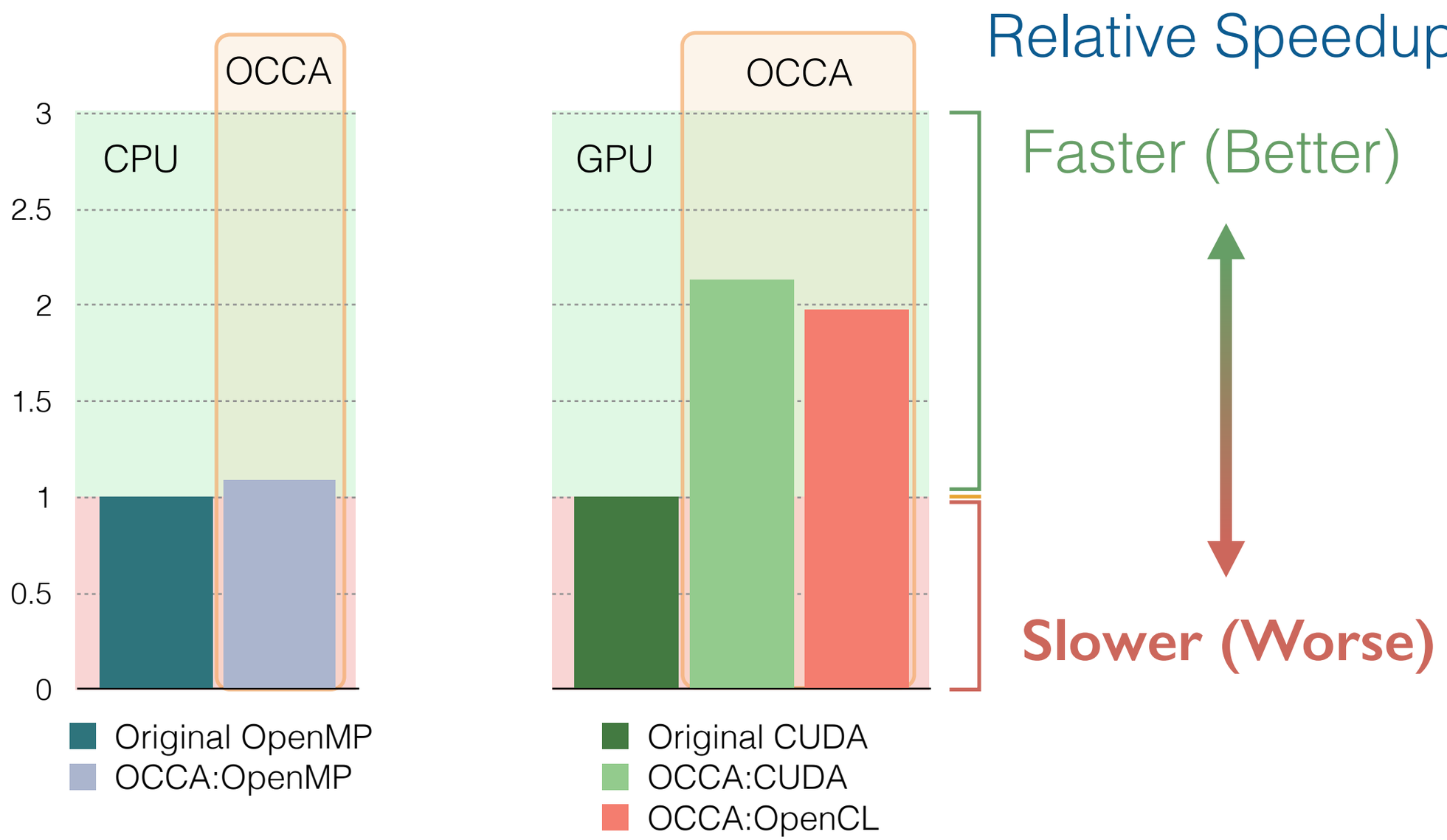




(J.-F. Remacle, R. Gandham, and T. Warburton: In Progress)

OCCA2 Rodinia Benchmarks

Back Propagation

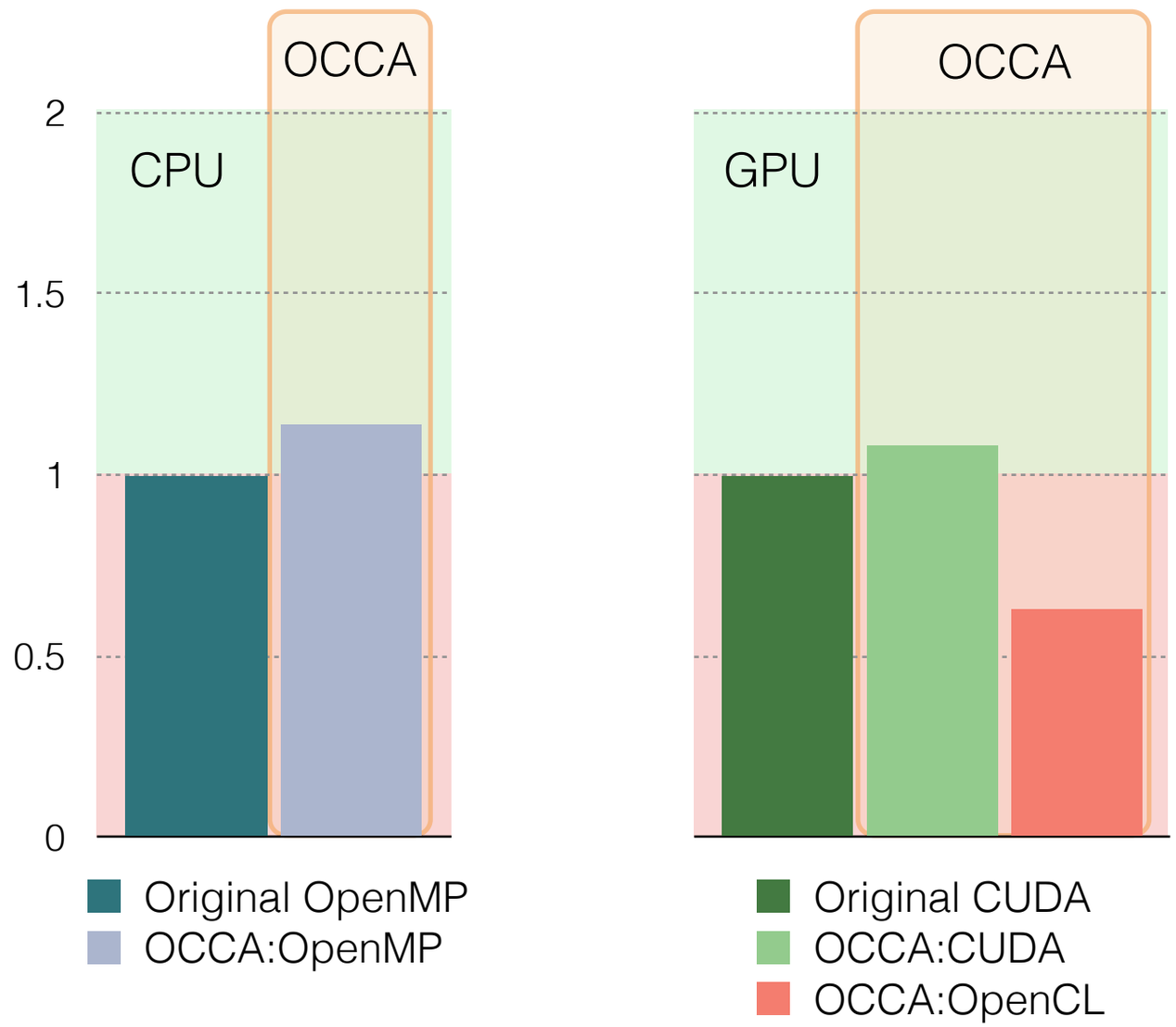


OpenMP : 2x Intel Xeon CPU E5-2650
OpenCL/CUDA : NVIDIA 980 GPU

(Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K. 2009)

OCCA2 Rodinia Benchmarks

Breadth-First Search

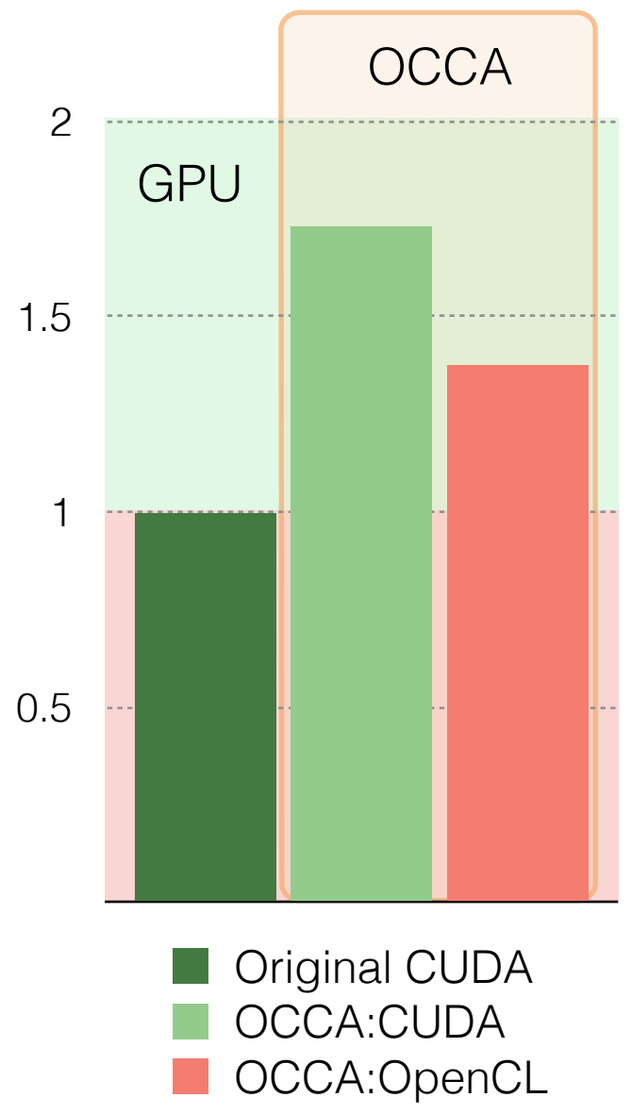


OpenMP : 2x Intel Xeon CPU E5-2650
OpenCL/CUDA : NVIDIA 980 GPU

(Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K. 2009)

OCCA2 Rodinia Benchmarks

Gaussian Elimination

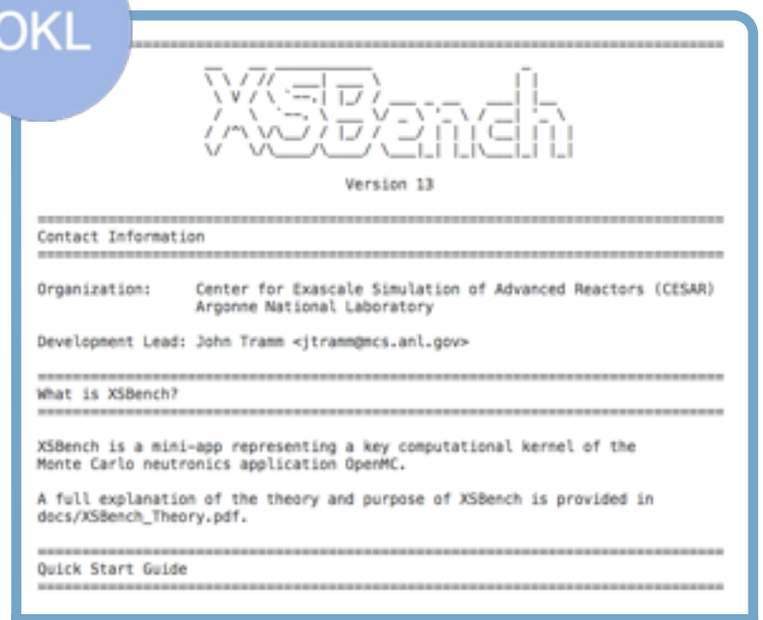


OpenMP : 2x Intel Xeon CPU E5-2650
OpenCL/CUDA : NVIDIA 980 GPU

(Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K. 2009)

OCCA2 Applications: Monte Carlo

OKL


Version 13

Contact Information

Organization: Center for Exascale Simulation of Advanced Reactors (CESAR)
Argonne National Laboratory

Development Lead: John Tramm <jtramm@mcs.anl.gov>

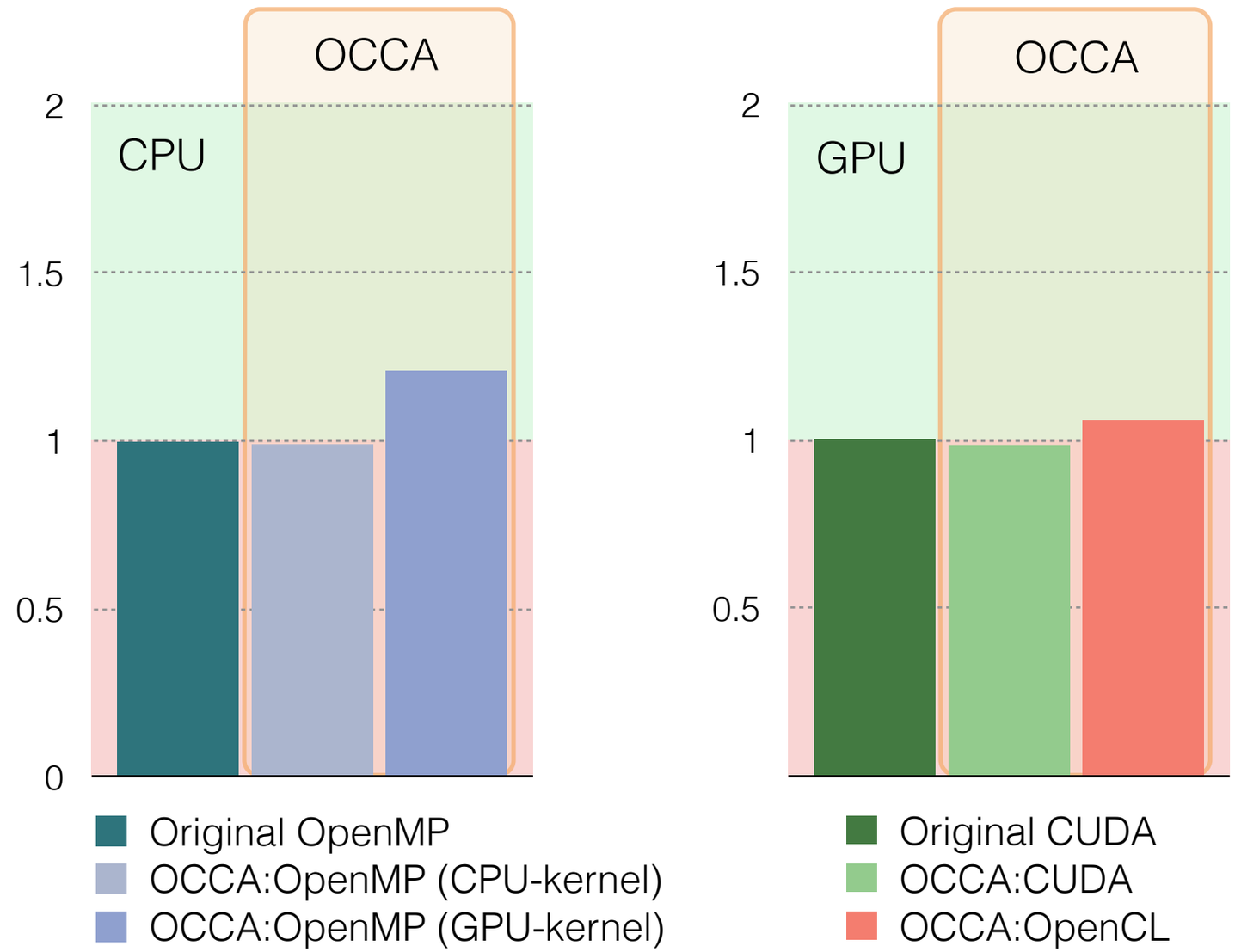
What is XSBench?

XSBench is a mini-app representing a key computational kernel of the Monte Carlo neutronics application OpenMC.

A full explanation of the theory and purpose of XSBench is provided in docs/XSBench_Theory.pdf.

Quick Start Guide

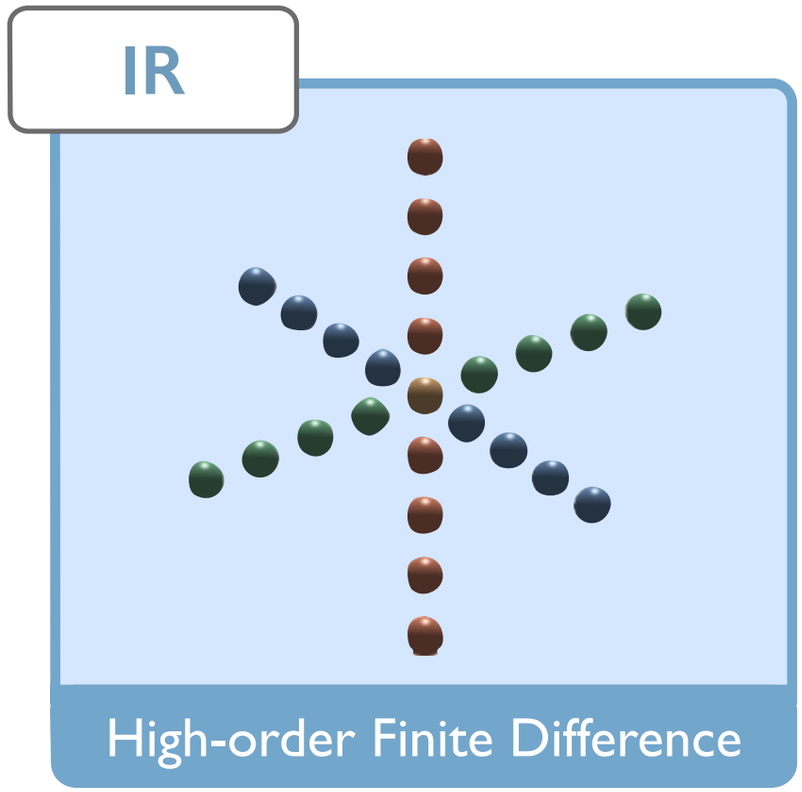
Monte Carlo

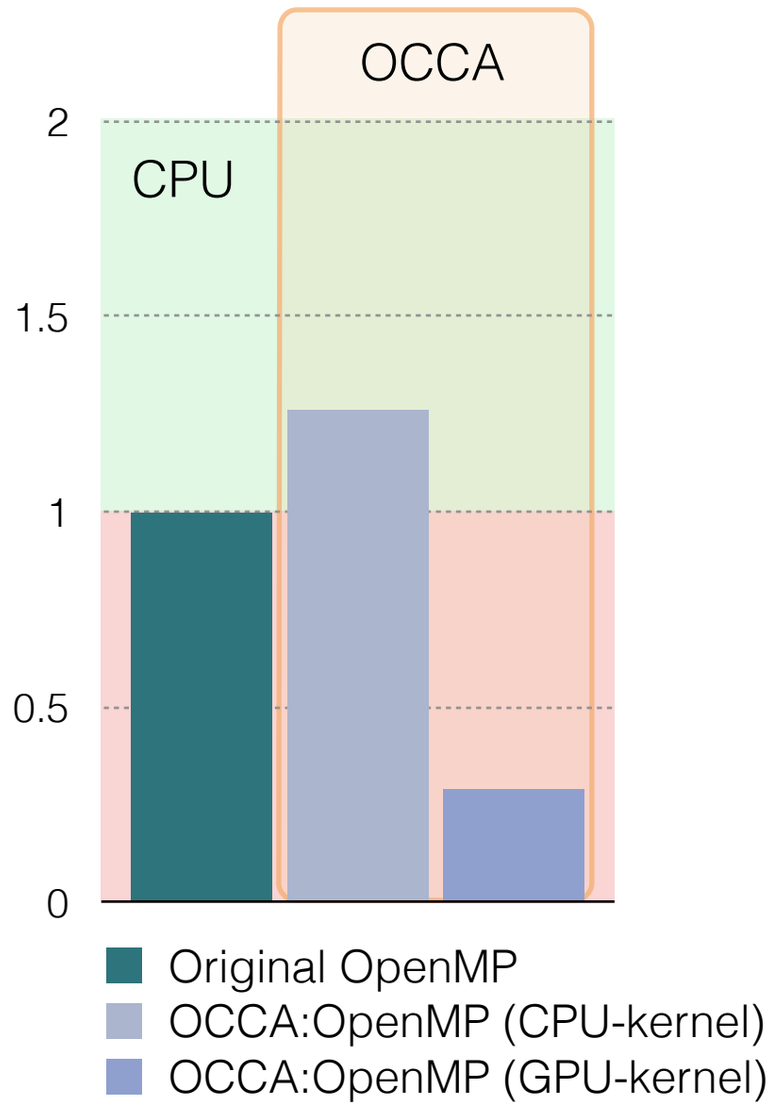


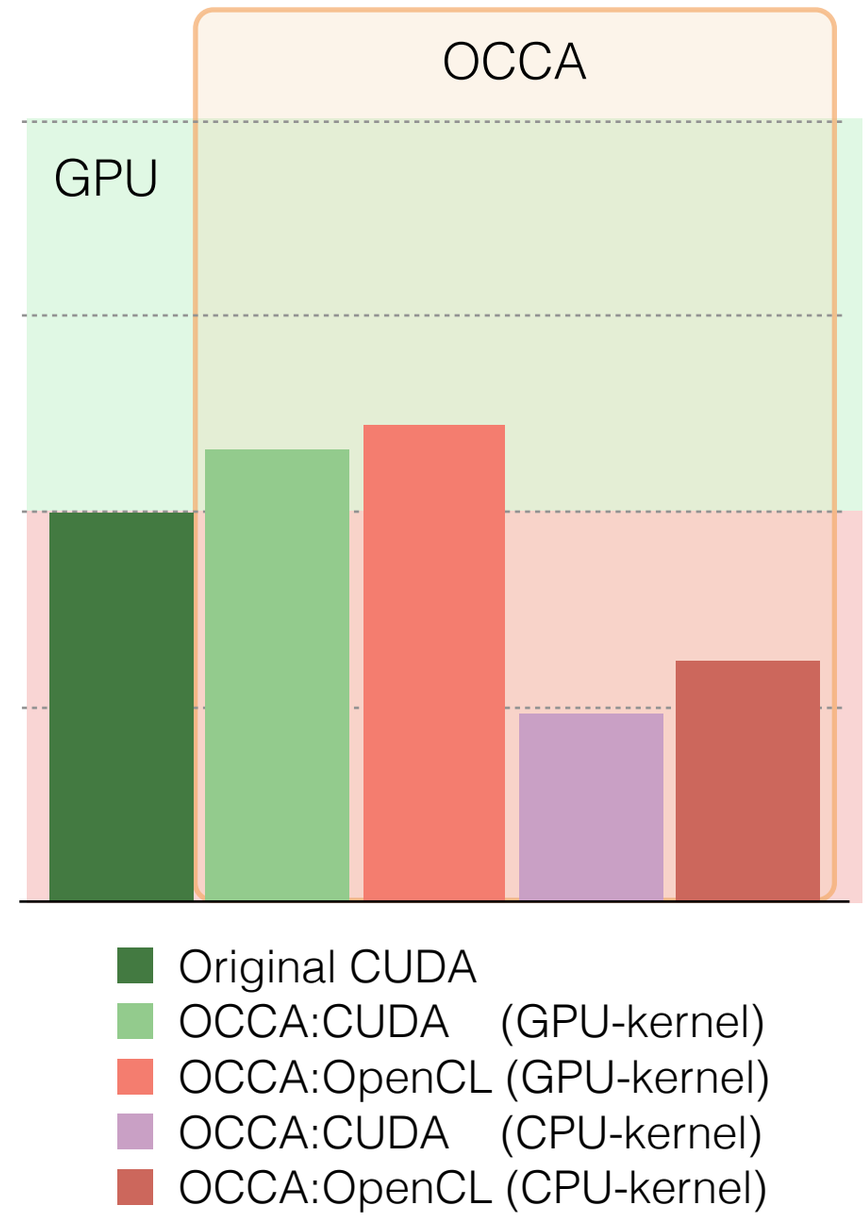
OpenMP : 2x Intel Xeon CPU E5-2650
OpenCL/CUDA : NVIDIA 980 GPU

3rd party use of OCCA at Argonne National Laboratory
(Rahaman, R., Medina, D., Lund, A., Tramm, J., Warburton, T. and Seigel, A. 2015)

OCCA2 Applications: Finite Difference







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

(Medina, D. S., St-Cyr, A. and Warburton, T. 2015)

OCCA2 Summary

Many-core Device Abstraction

- Abstract unified framework addressing portability and performance
- Implementation displays high performance across platforms

Unified Kernel Language

- Presented an approach for unifying programming language features (OCCA IR)
- Introduced language extensions for C and Fortran (OKL/OFL)

Automation

- Targeted difficulties in programming many-core devices
- Automate data transfers and kernel generation

Results

- Examined a wide range of numerical methods
- Compared performance with native counterparts through benchmarks

libocca github

An experimental version of OCCA is available from github

GitHub, Inc.

This repository Search

Pull requests Issues Gist

libocca / occa

Unwatch 7 Star 6 Fork 1

OCCA: Portable Approach for Parallel Architectures (http://libocca.org) — Edit

1,730 commits 2 branches 0 releases 4 contributors

Branch: master occa / +

[Base] Forgot to remove some printing outputs

dmed256 authored 2 days ago latest commit 7cc964a8d0

bin	[Bin] Added bin/occaInfo to print available devices	6 months ago
docs	[Web] Update	8 months ago
editorTools	[Editor] Updating okl-mode.el to handle @ attributes	2 months ago
examples	[bin] Can handle wildcards and stuff	13 days ago
include	[Base] Forgot to remove some printing outputs	2 days ago
lib	[Array] Changed to column-major for array. Fast strides look like: ar…	3 days ago
sandbox	[Array] Adding @ arrayArg attribute for passing multi-dimensional arr…	16 days ago
scripts	[Makefile] Need to push defines into additional file	5 days ago

Code

Issues 2

Pull requests 0

Wiki

Pulse

Graphs

Settings

HTTPS clone URL

https://github.com/libo

You can clone with HTTPS, SSH, or Subversion.

https://github.com/libocca/occa

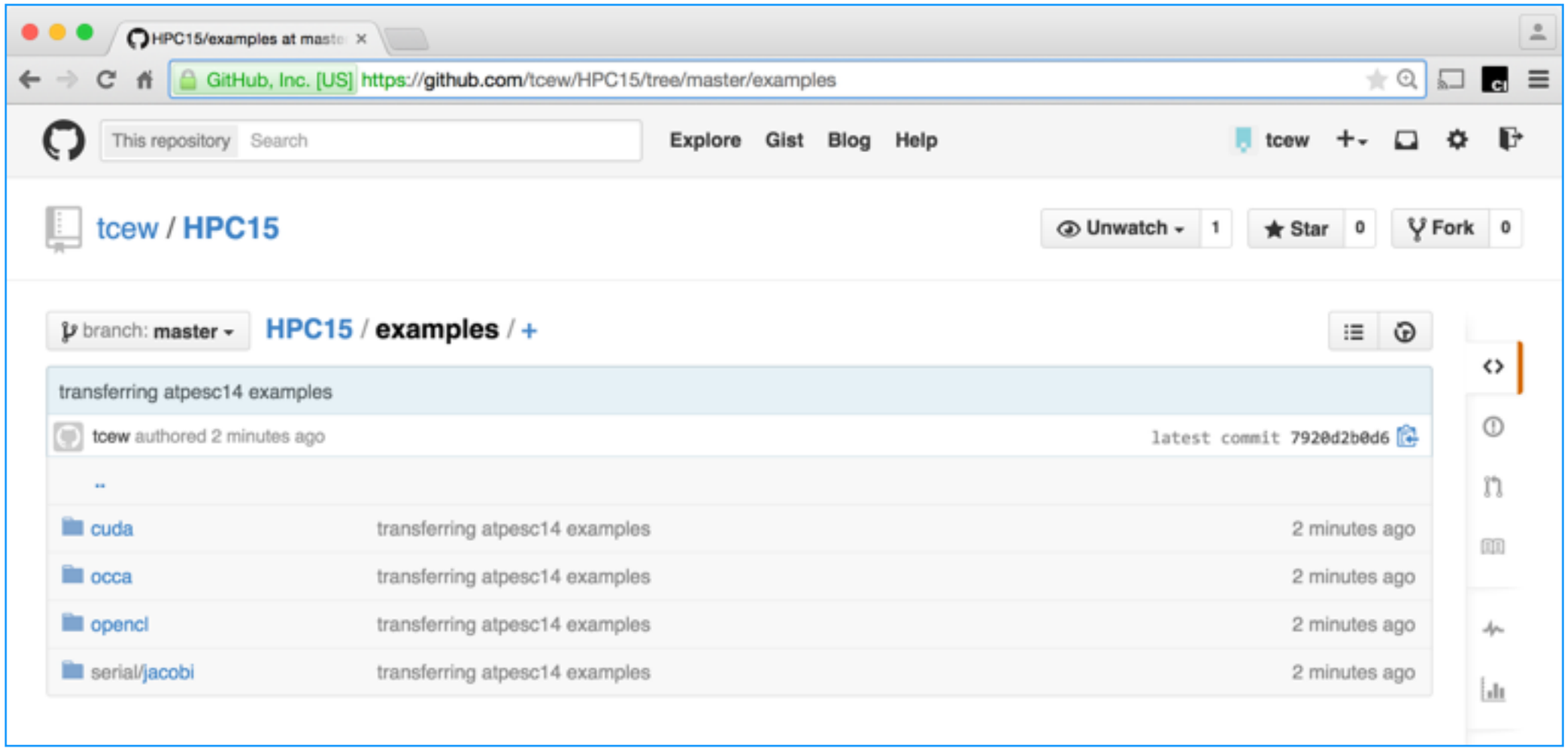
OCCA: elliptic solver sample code

See the ATPESC14 github for a full implementation [with some optional goodies]

1: git clone and build the OCCA library. 2: set up the environment variables for OCCA
*3: Let me know ** when ** you have any problems with this.*

OCCA: elliptic solver sample code

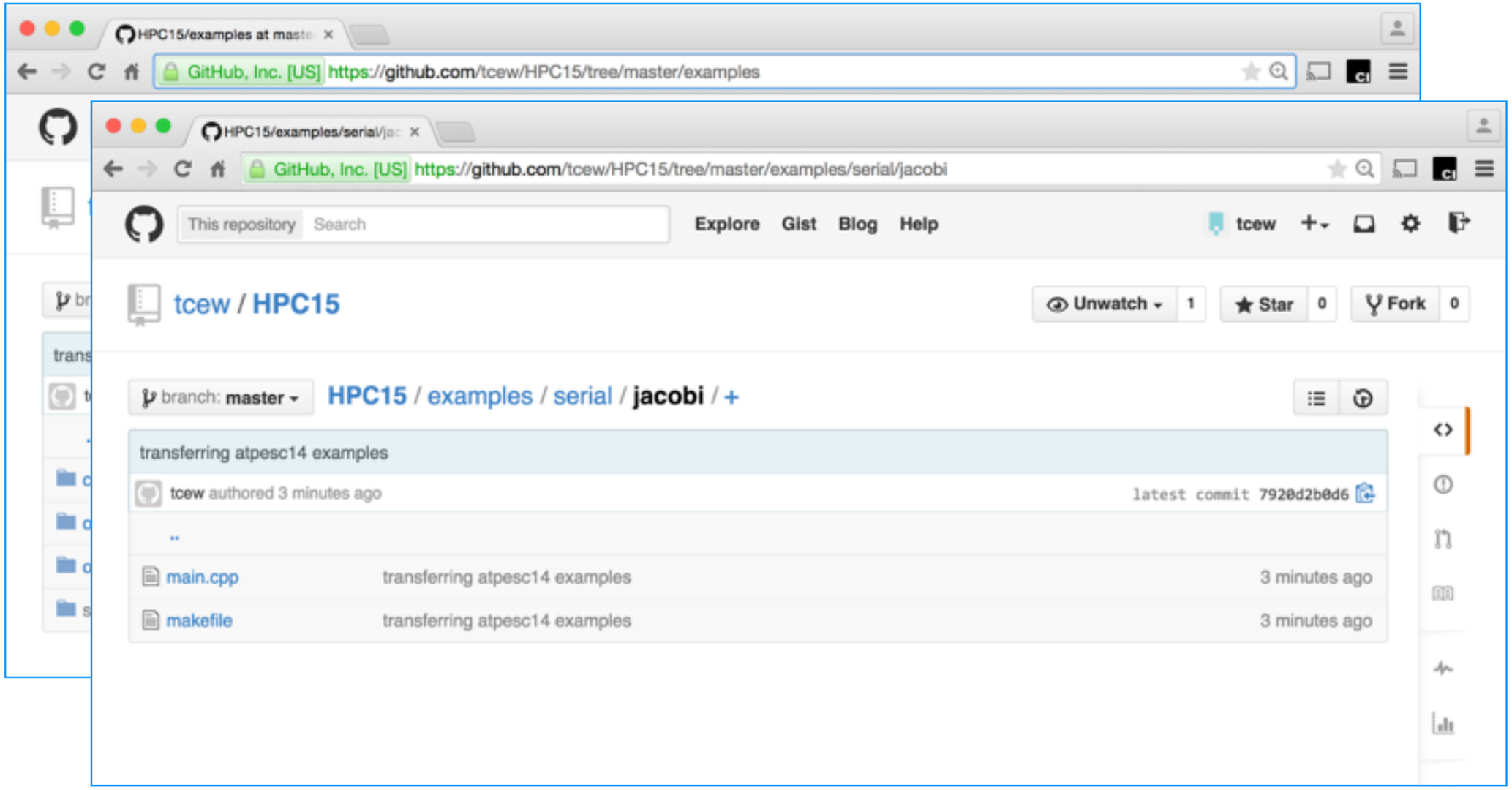
See the ATPESC14 github for a full implementation [with some optional goodies]



1: git clone and build the OCCA library. 2: set up the environment variables for OCCA
*3: Let me know ** when ** you have any problems with this.*

OCCA: elliptic solver sample code

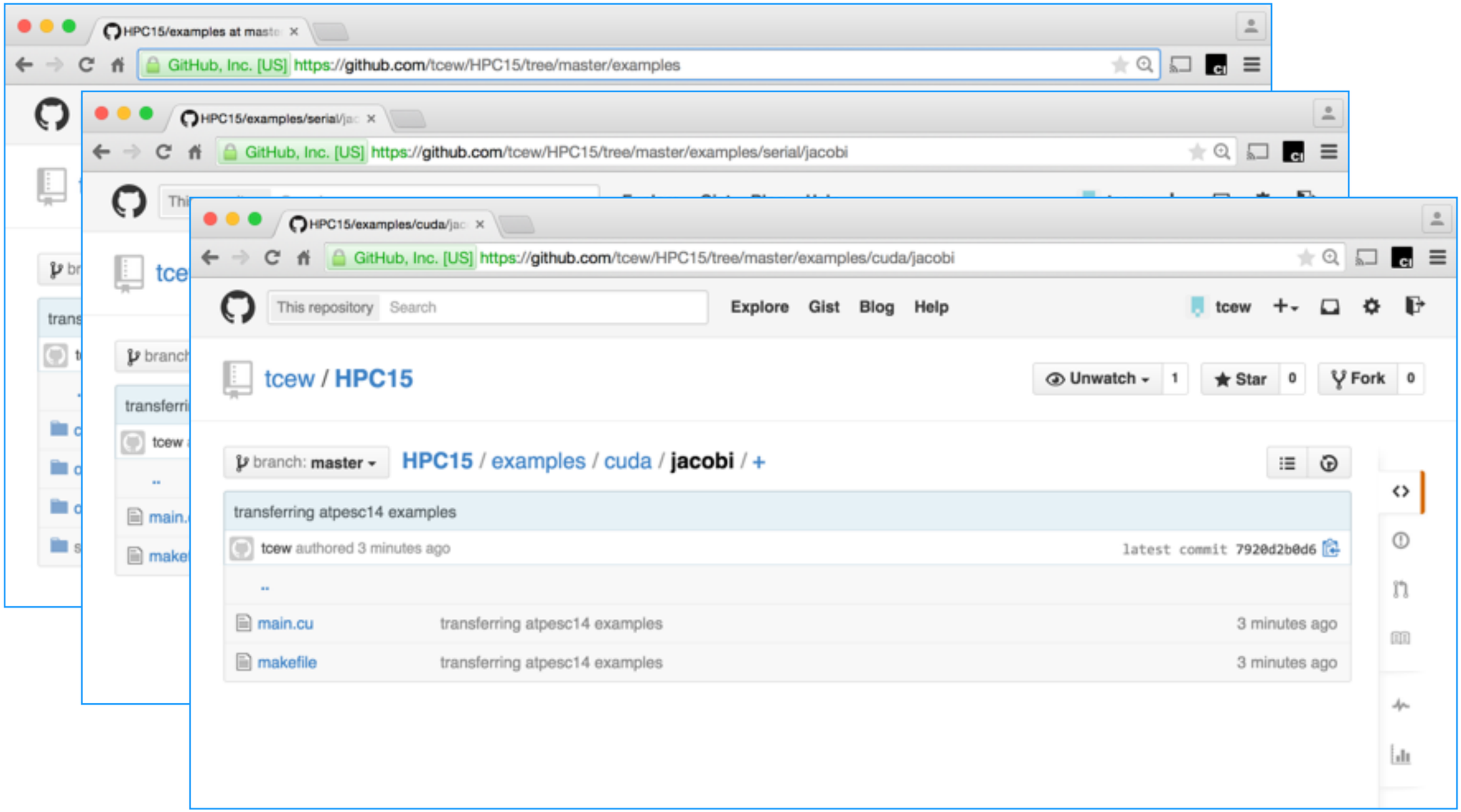
See the ATPESC14 github for a full implementation [with some optional goodies]



1: git clone and build the OCCA library. 2: set up the environment variables for OCCA
*3: Let me know ** when ** you have any problems with this.*

OCCA: elliptic solver sample code

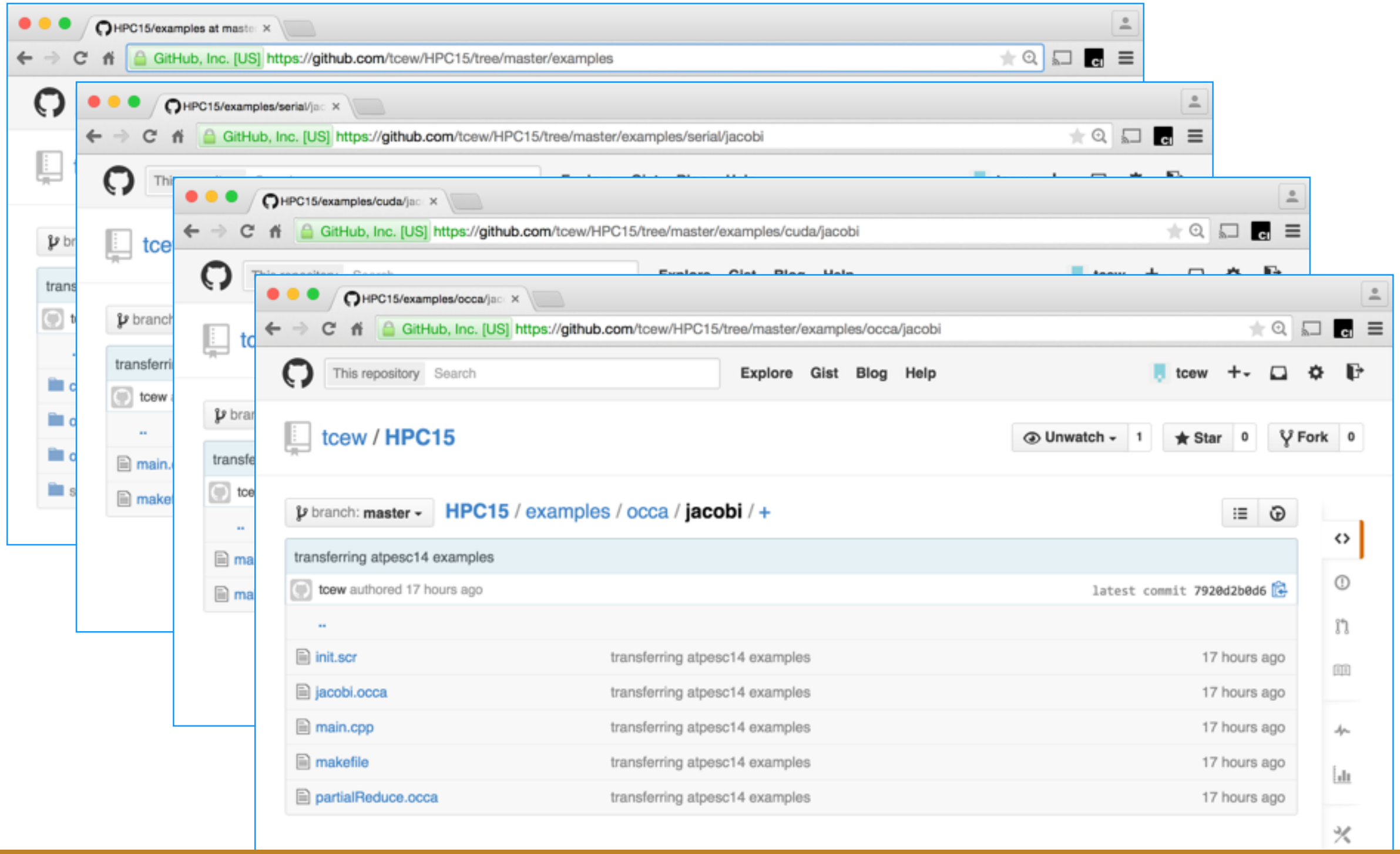
See the ATPESC14 github for a full implementation [with some optional goodies]



1: git clone and build the OCCA library. 2: set up the environment variables for OCCA
*3: Let me know ** when ** you have any problems with this.*

OCCA: elliptic solver sample code

See the ATPESC14 github for a full implementation [with some optional goodies]



1: git clone and build the OCCA library. 2: set up the environment variables for OCCA
*3: Let me know ** when ** you have any problems with this.*

OCCA: comparing Jacobi kernels

Recalling the 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

Recalling the 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

Recalling the 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)*(

s[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…)

OCCA: summary

OCCA:

- ✅ Lightweight API that hedges against shifts in programming models.
- ✅ Front ends: C++, C, Julia, Python, MATLAB, F90,…
- ✅ Currently works with OpenCL, CUDA, OpenMP, & pThreads back ends.

In progress:

- ✅ Version for Windows.
- ~ C# and Java front-ends.
- ~ Testing on the Xeon Phi.
- ~ New multi-threading languages will be added as they emerge and stabilize.

OCCA applications:

- ✅ Portability and performance for finite-difference, finite volume, & finite elements.
  - ~ Scalability of fully accelerated algebraic multi-grid preconditioner setup/cycling.
  - ~ ALMOND performance is a work in progress.

Comments:

- ✅ OCCA does reduce exposure to vendor-lock and churn in programming models.
- ✅ OCCA does provide an extra optimization direction for auto-tuning.
- ❌ OCCA does not provide high-level support for tuning and optimization, yet.

If time permits a quick demo of running a simple host code and OCCA kernel.

OCCA: arXiv report

A report describing OCCA is available from arXiv.org

[1403.0968] OCCA: A unif

arxiv.org/abs/1403.0968

Cornell University Library

We gratefully acknowledge support from the Simons Foundation and Rice University

arXiv.org > cs > arXiv:1403.0968

Search or Article-id (Help | Advanced search)

All papers Go!

Computer Science > Distributed, Parallel, and Cluster Computing

OCCA: A unified approach to multi-threading languages

David S Medina, Amik St-Cyr, T. Warburton

(Submitted on 4 Mar 2014)

The inability to predict lasting languages and architectures led us to develop OCCA, a C++ library focused on host-device interaction. Using run-time compilation and macro expansions, the result is a novel single kernel language that expands to multiple threading languages. Currently, OCCA supports device kernel expansions for the OpenMP, OpenCL, and CUDA platforms. Computational results using finite difference, spectral element and discontinuous Galerkin methods show OCCA delivers portable high performance in different architectures and platforms.

Comments: 25 pages, 6 figures, 9 code listings, 8 tables, Submitted to the SIAM Journal on Scientific Computing (SISC), presented at the Oil & Gas Workshop 2014 at Rice University
Subjects: **Distributed, Parallel, and Cluster Computing (cs.DC)**
Cite as: arXiv:1403.0968 [cs.DC]
(or arXiv:1403.0968v1 [cs.DC] for this version)

Submission history

From: David Medina [view email]
[v1] Tue, 4 Mar 2014 22:30:49 GMT (86kb,D)

Download:

- PDF
- Other formats

Current browse context:
cs.DC
< prev | next >
new | recent | 1403

Change to browse by:
cs

References & Citations
- NASA ADS

DBLP - CS Bibliography
listing | bibtex
David S. Medina
Amik St-Cyr
Amik St.-Cyr
T. Warburton

Bookmark (what is this?)

OCCA 2.0: github

Check out OCCA out on GitHub at https://github.com/libocca/occa

New contact info as of 08/10:

Mathematics @ Virginia Tech
tim.warburton@vt.edu

VirginiaTech
Invent the Future®