OCCA2: extensible many-core programming

Tim Warburton
Professor of Computational & Applied Mathematics @ Rice University

OCCA2: slides & repos

git clone https://github.com/tcew/OCCA2Tutorial

git clone https://github.com/tcew/OCCA2

Overview

Part 1: Brief review of CUDA

- Progression in NVIDIA GPU hardware.
- Abstract GPU architecture.
- Natural map between GPU architecture and CUDA thread model.
- CUDA kernel language and implicit parallel for-loop construct.

Part 2: Interlude on CUDA optimization.

Part 3: OpenCL (optional)

- Open Computing Language

Part 4: OCCA

- Unified and extensible many-core programming model.

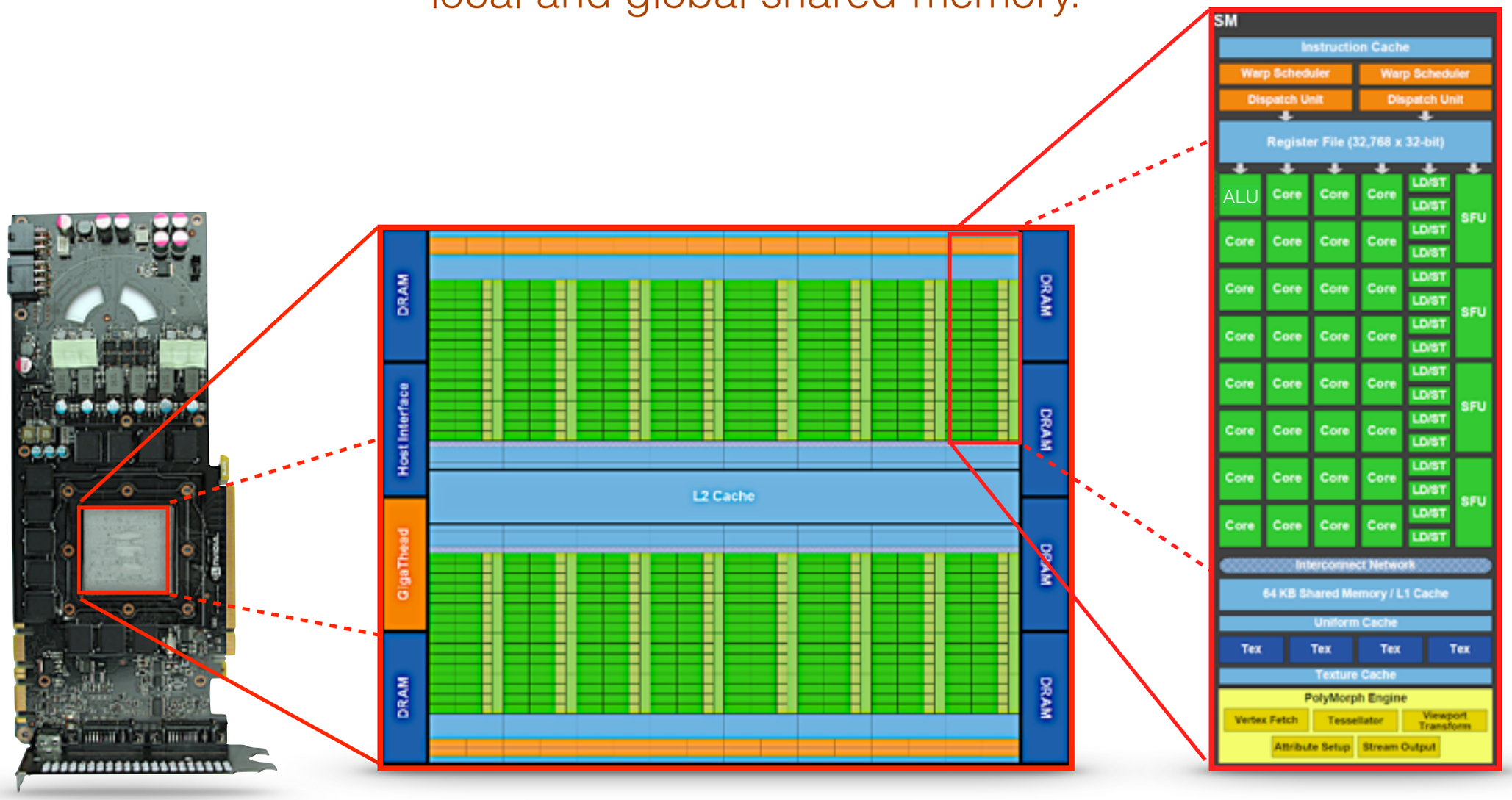
Contributors include: David Medina, Rajesh Gandham, Amik St-Cyr, & Lucas Wilcox.

Part 1: NVIDIA GPUs & CUDA Refresher

I think it is informative to look at the state of the art in NVIDIA hardware,
consider upcoming hardware changes,
and review the CUDA host API

GPU: excess ALUs

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

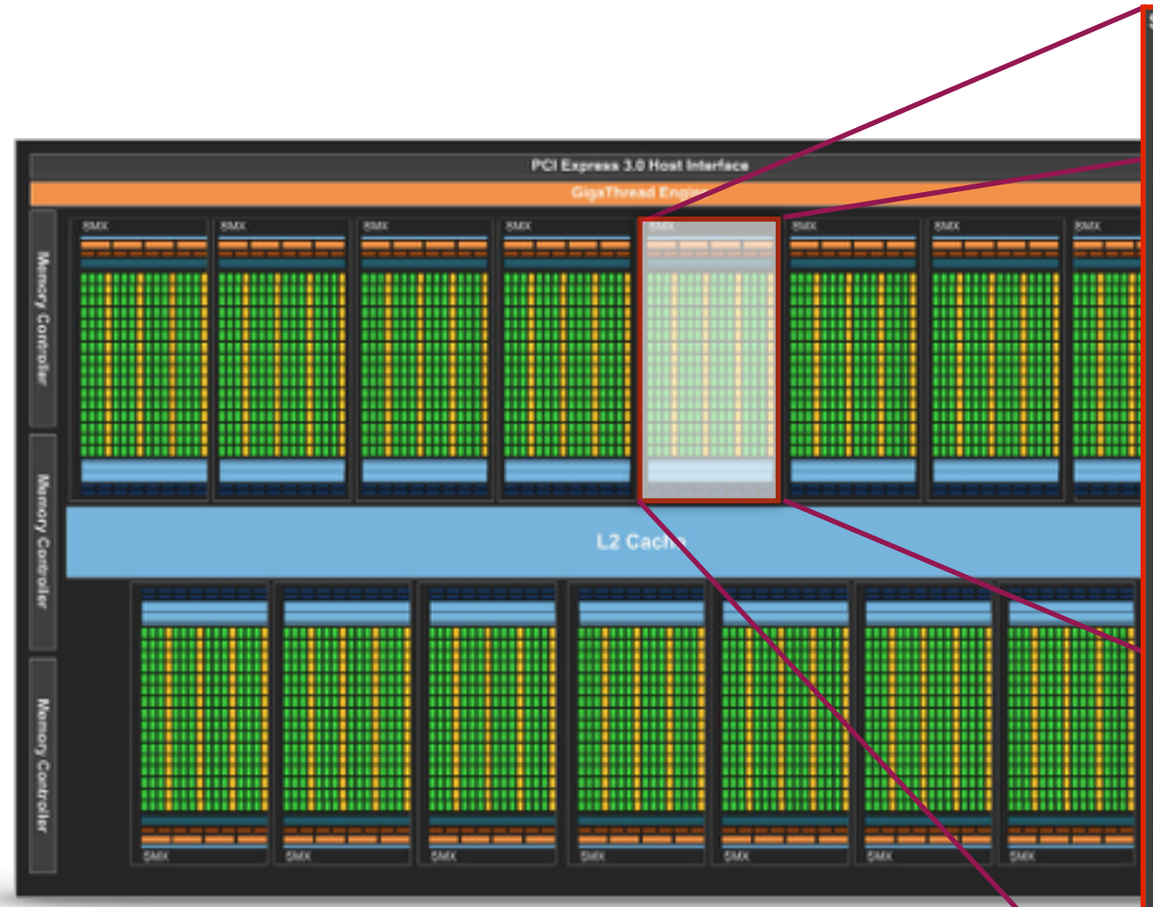


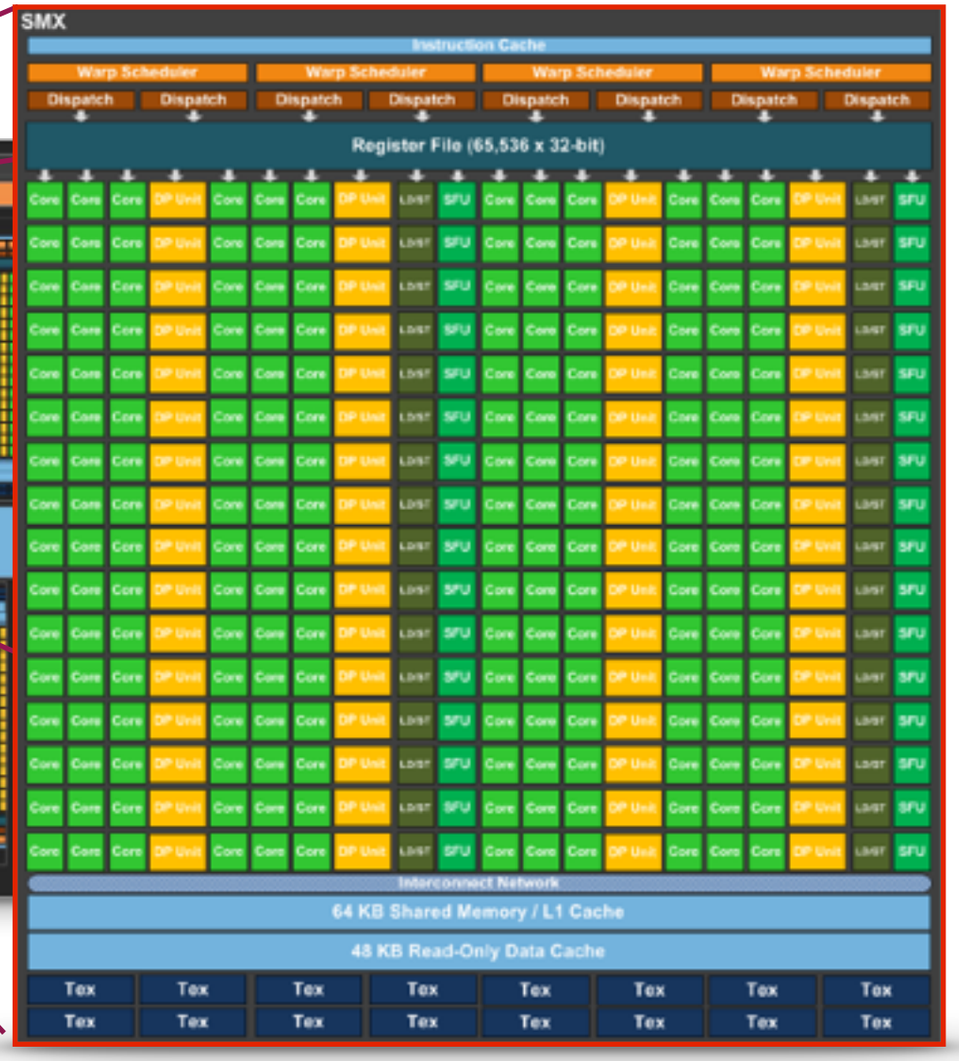
Each Fermi core (SM) has a SIMD clusters of 32 FPUs
Data streams at ~50 GFLOAT/s and computes up to 1.4 TFLOP/s (SP)

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.





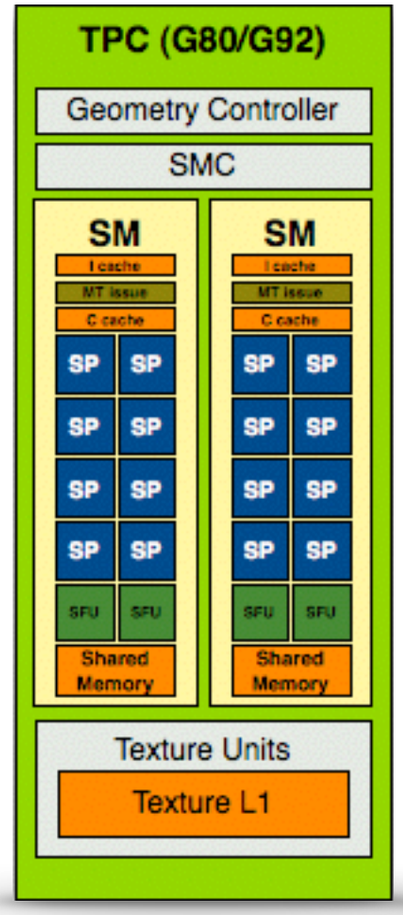
Each Kepler core (SMX) has six SIMD clusters of 32 ALUs
Data streams at ~70 GFLOAT/s and peak 4+ TFLOP/s (SP)

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

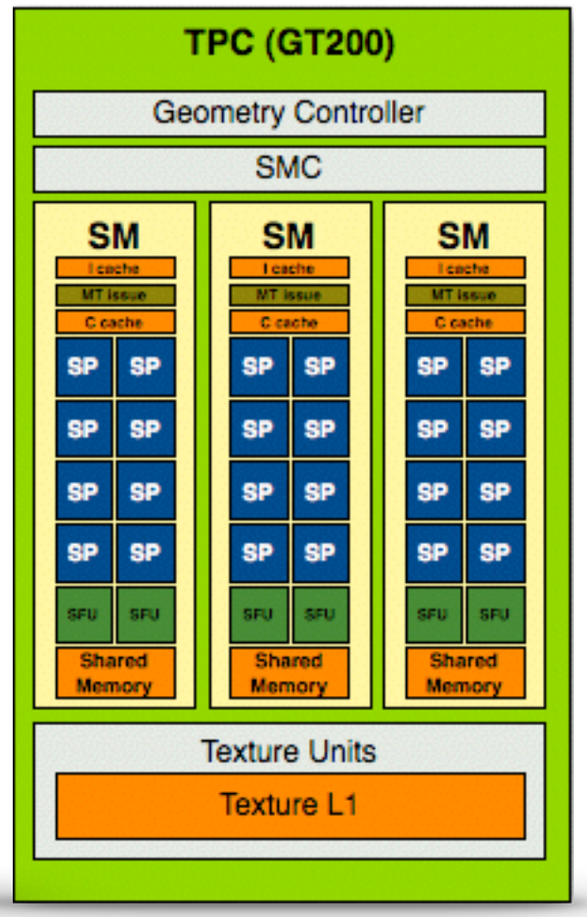
GPU: trends in FPU Clusters

The FPU clusters (“core”) in 4 NVIDIA generations

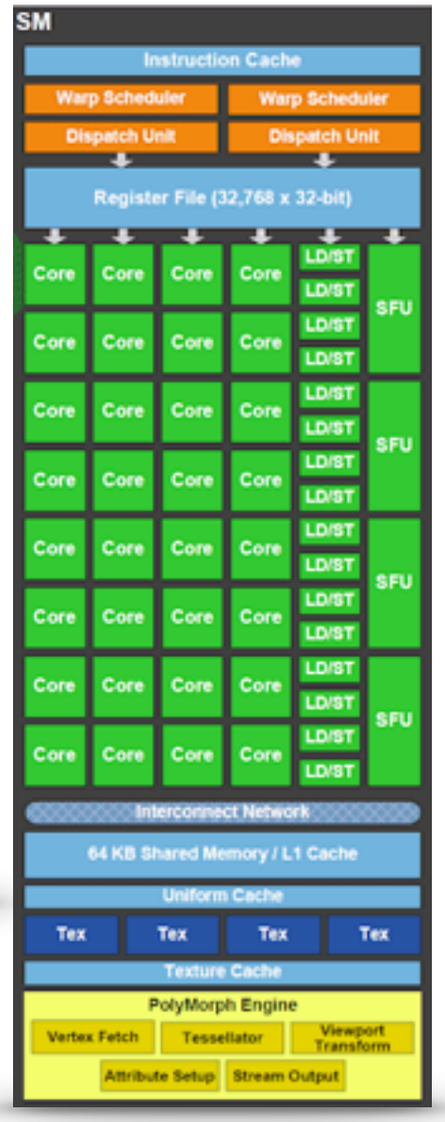
2007: G80



2008: Tesla



2010: Fermi



Q4 2012: Kepler GK110

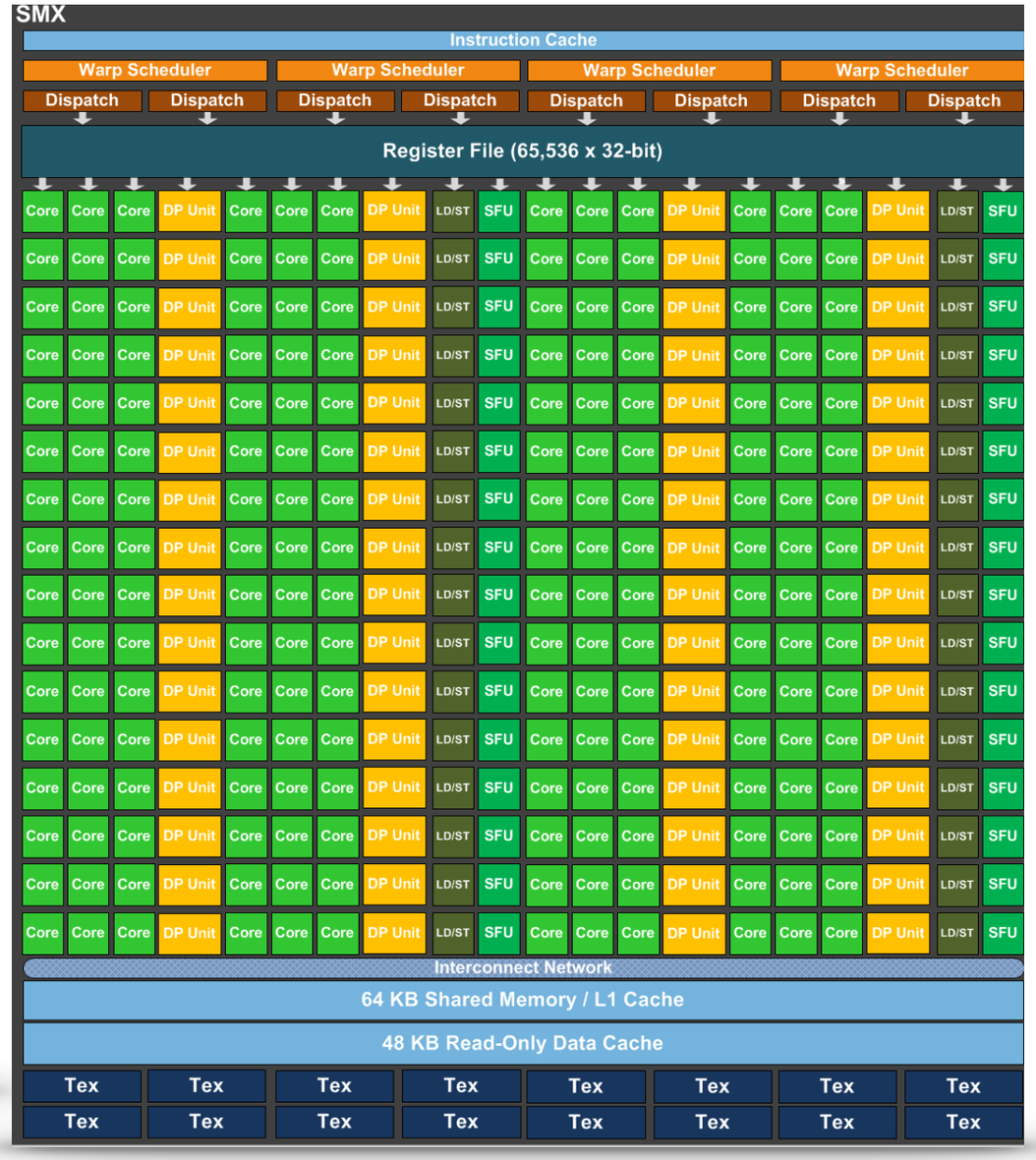


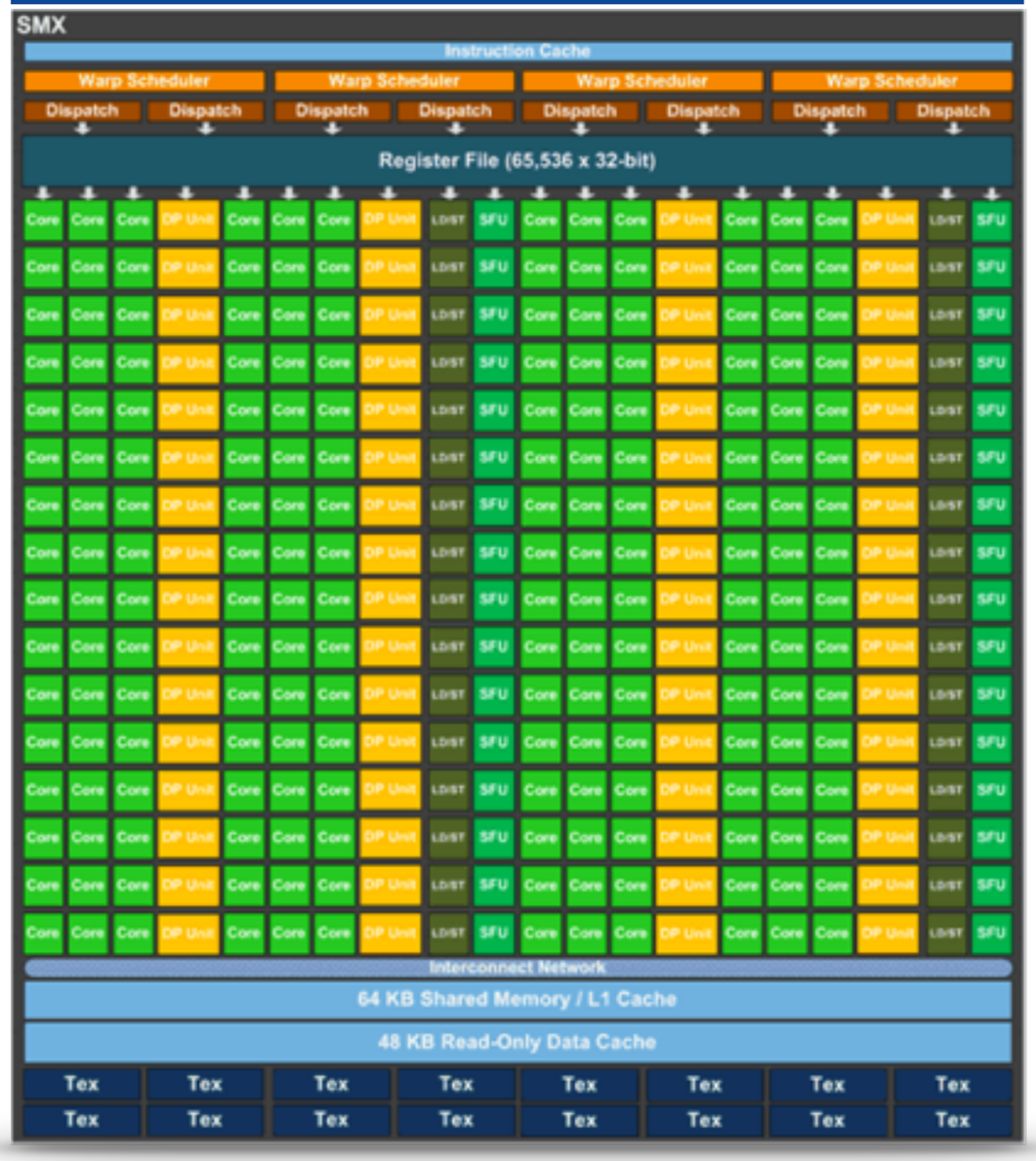
Image sources: http://forum.beyond3d.com/showthread.php?t=58668&page=140, http://www.anandtech.com/show/2549/2 , http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

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



Maxwell SMM

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

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

CUDA: compute capability

Over time the specs for NVIDIA cores have mutated.

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			

RG targeted later compute capability and removed some hard coded parameters.
Table credit: CUDA wikipedia page (http://en.wikipedia.org/wiki/CUDA)

GPU: summary of architecture

Summary of multi-level GPU parallel architecture

- Side-car accelerator (or daughter board) with its own memory spaces.
- Multiple cores.
- Each core has one (or more) wide SIMD vector units.
    - Each wide SIMD unit executes one instruction stream.
    - Each stream may involve vector operations [VLIW vector ops on AMD GCN]
- 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.*

GPU: natural thread model

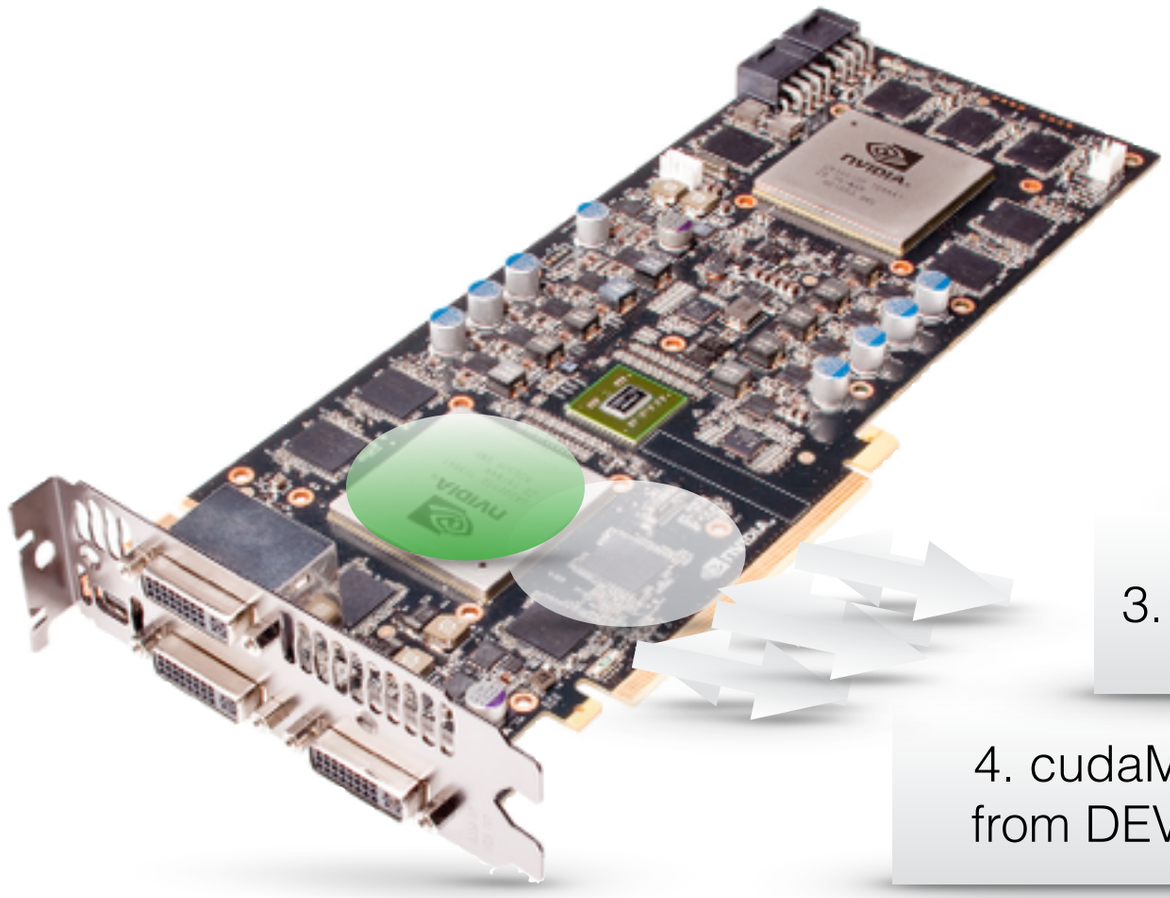
The GPU architecture admits a natural parallel threading model

- Programmer migrates data between HOST and DEVICE.
- Programmer divides 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 API basics

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: 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:

```
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;

  }
}
```

[we are doing this for a reason to be clear later]

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

CUDA: motivating serial function

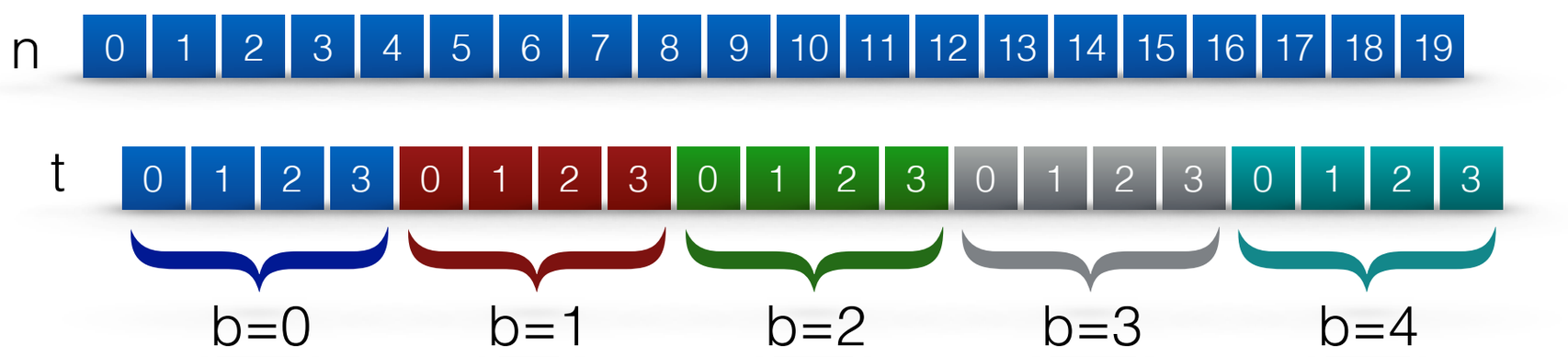
Consider the case with N=20 - then break the for loop into independent tiles:

```
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)

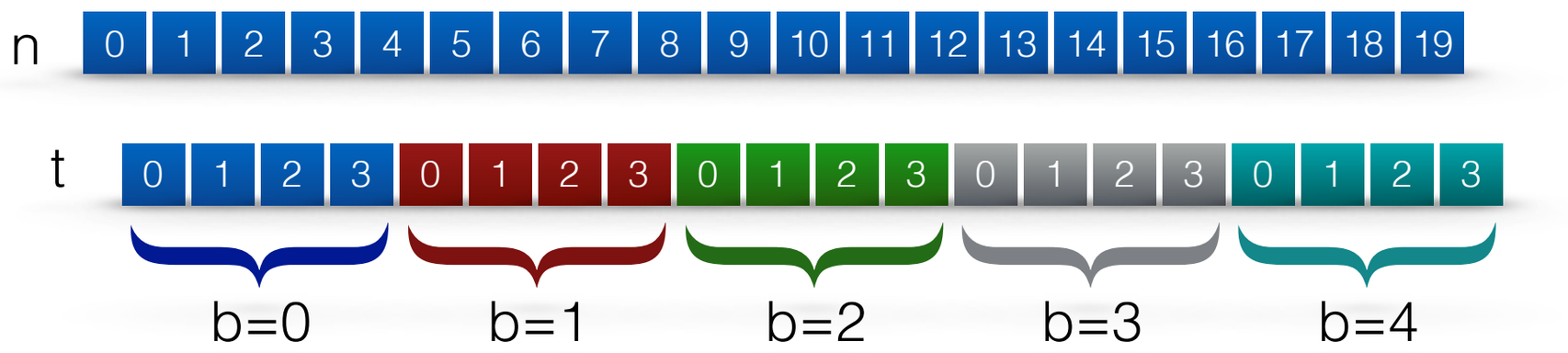
```
void tiledSerialSimpleKernel(int N, float *d_a){

  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: 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;
}
```

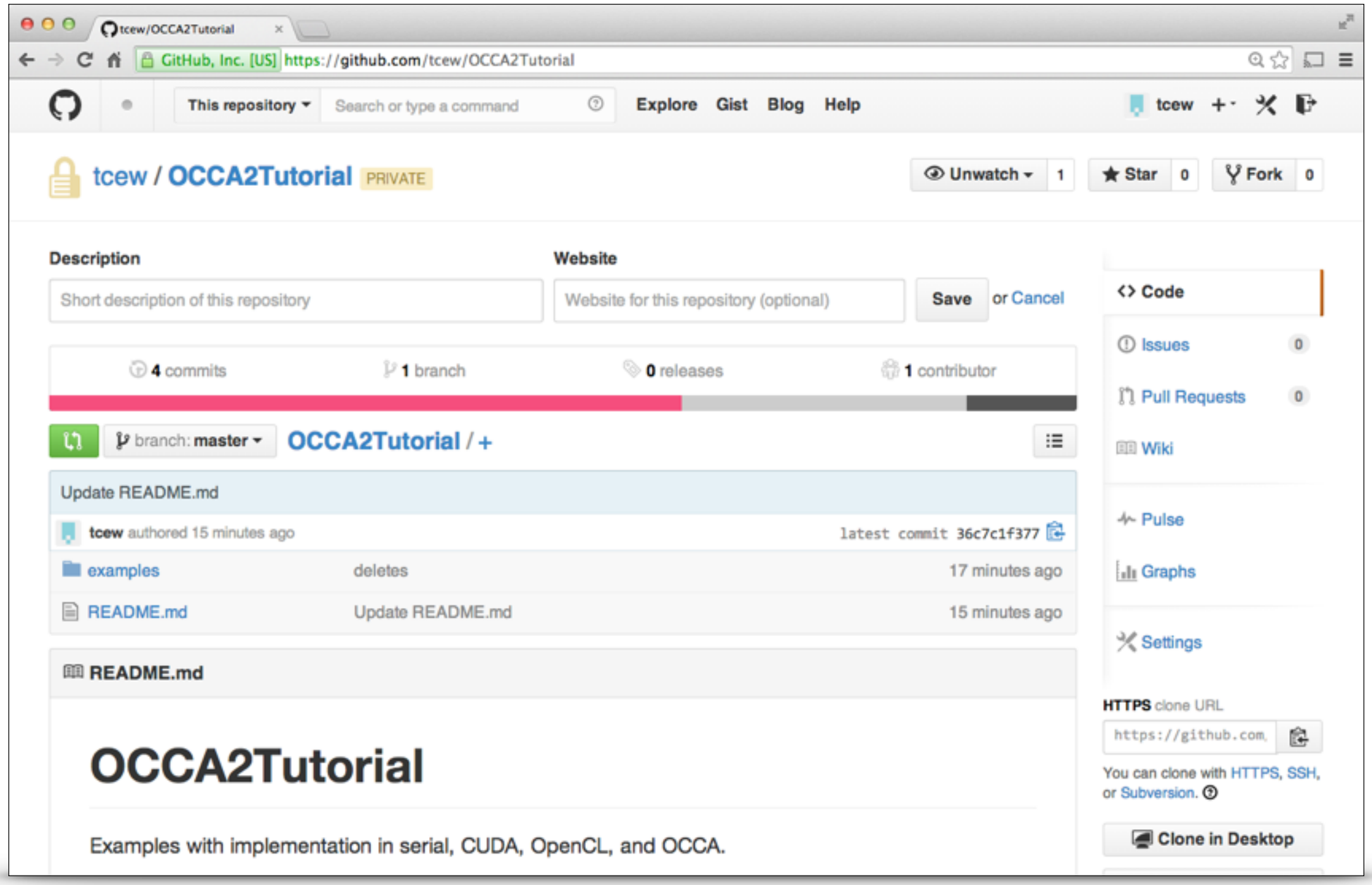
Action performed by each thread

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

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

Example: codes in serial, CUDA, OpenCL, & OCCA.



github repo: https://github.com/tcew/OCCA2Tutorial
See: examples/cuda/simple

CUDA: compiling & running example

Note: CUDA is a C-extension and typically requires *nvcc* to compile:

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

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

Make sure you can complete this exercise now !

Source code: https://github.com/tcew/OCCA2Tutorial/examples/cuda/simple

CUDA: elliptic solver example

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

Elliptic Poisson problem:

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

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

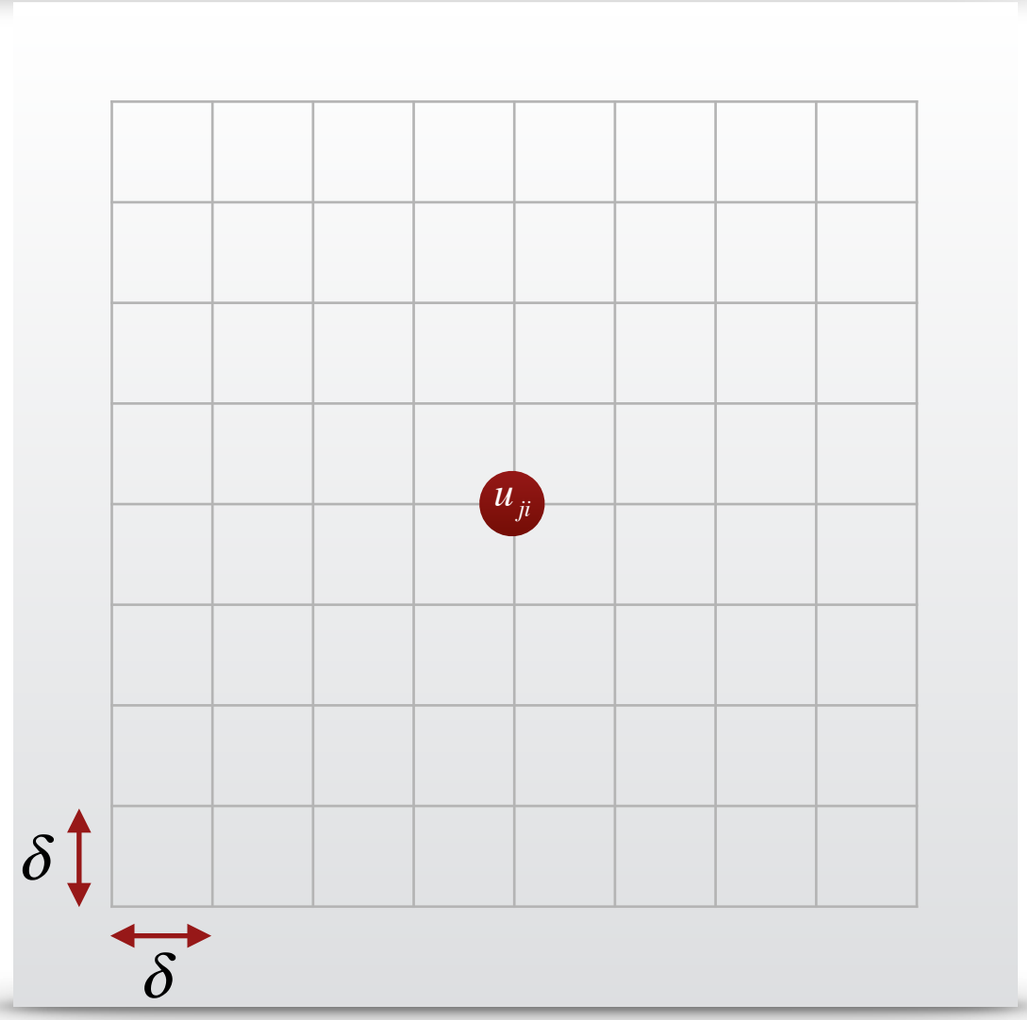
Poisson problem is an archetypal building block for many physics packages.

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):

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

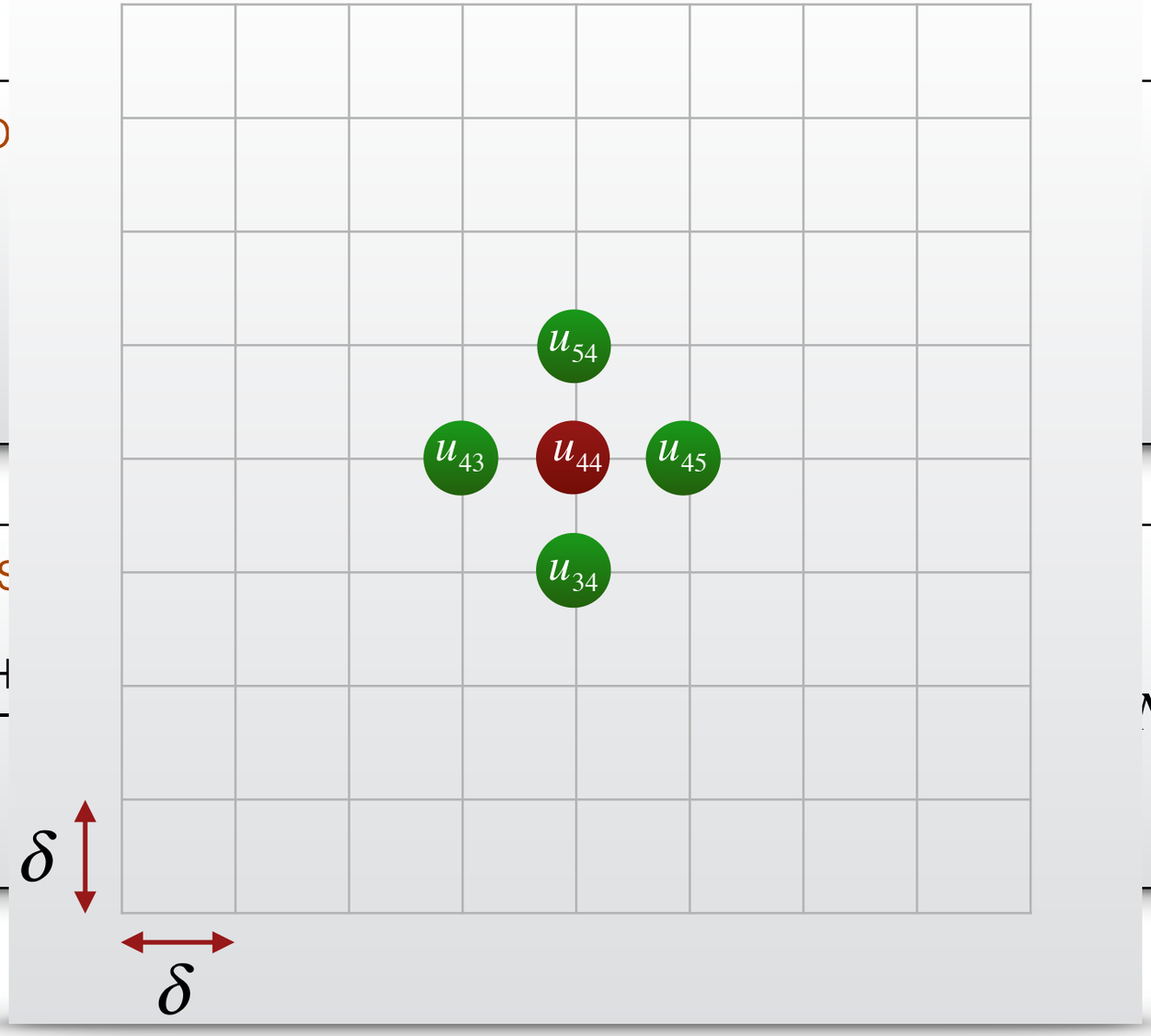
$$u_{ji}=0 \text{ for } i=0,N+1 \text{ or } j=0,N+1$$

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 Poisso



Discrete Poiss

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

N

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: 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:

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

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

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:

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

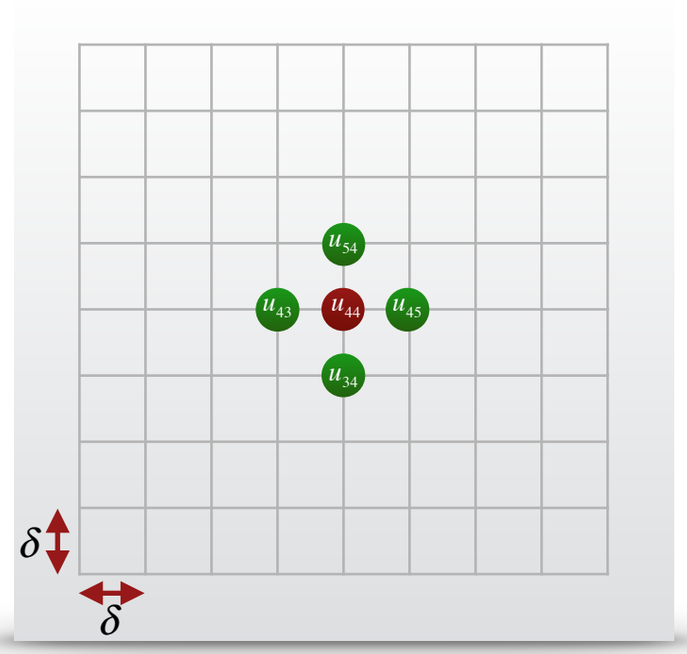
Source code: https://github.com/tcew/OCCA2Tutorial/examples/cuda/simple

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: serial Jacobi iteration

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

Iterate:

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

Serial kernel:

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

  for(int i=0;i<N;++i){
    for(int j=0;j<N;++j){

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

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

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

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

CUDA: parallel Jacobi iteration

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

Iterate:

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

CUDA kernel:

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

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

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

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

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

  }
}
```

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

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

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!!

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];
}
```

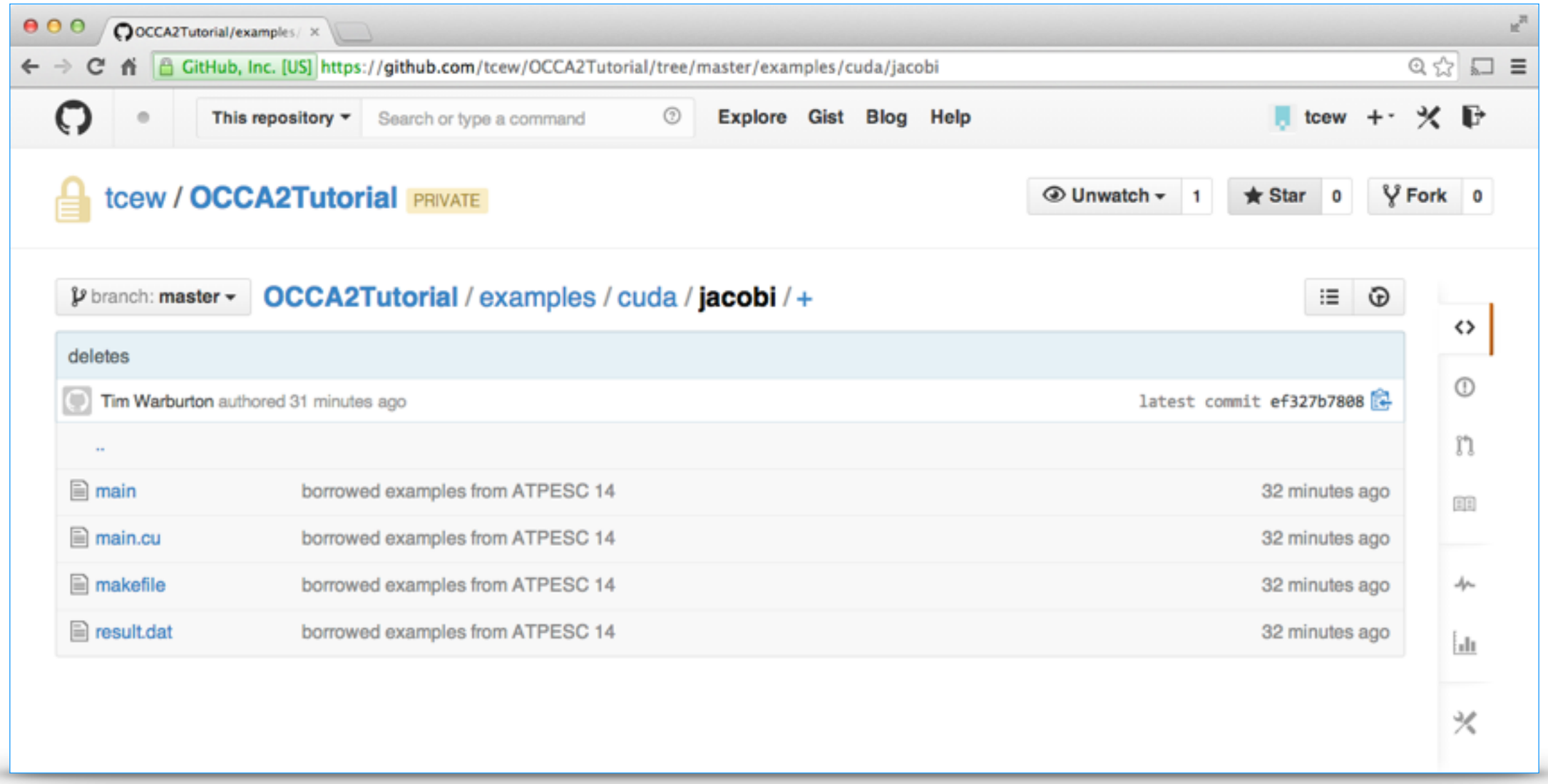
Thread \ Step	0	1	2	3	4	5	6
0	1	1+7	8	8+11	19	19+17	36
1	3	3+8	11	11+6	17		
2	5	5+6	11				
3	2	2+4	6				
4	7						
5	8						
6	6						
7	4						

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 OCCA2Tutorial 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.

Part 2: 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: 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:		(Help)
Active Threads per Multiprocessor	2048	
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

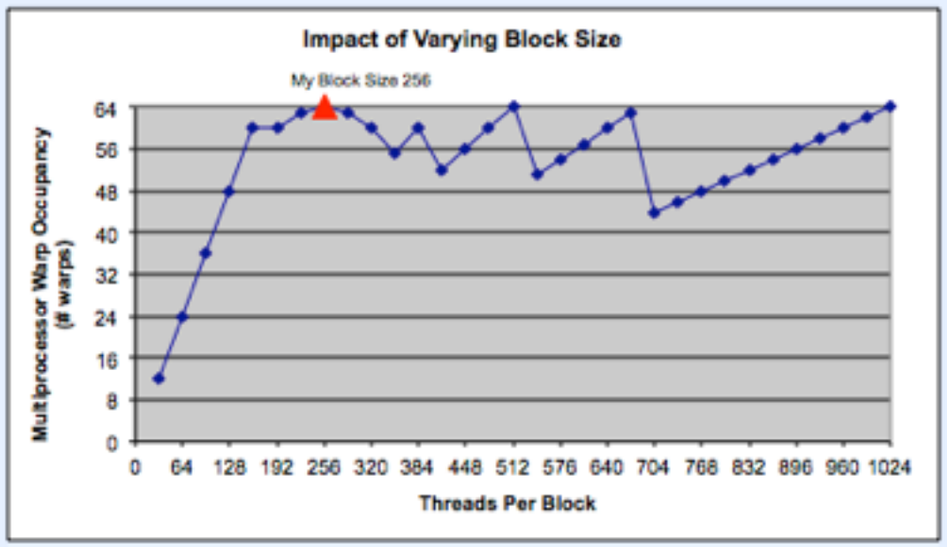
Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	4096	49152	12

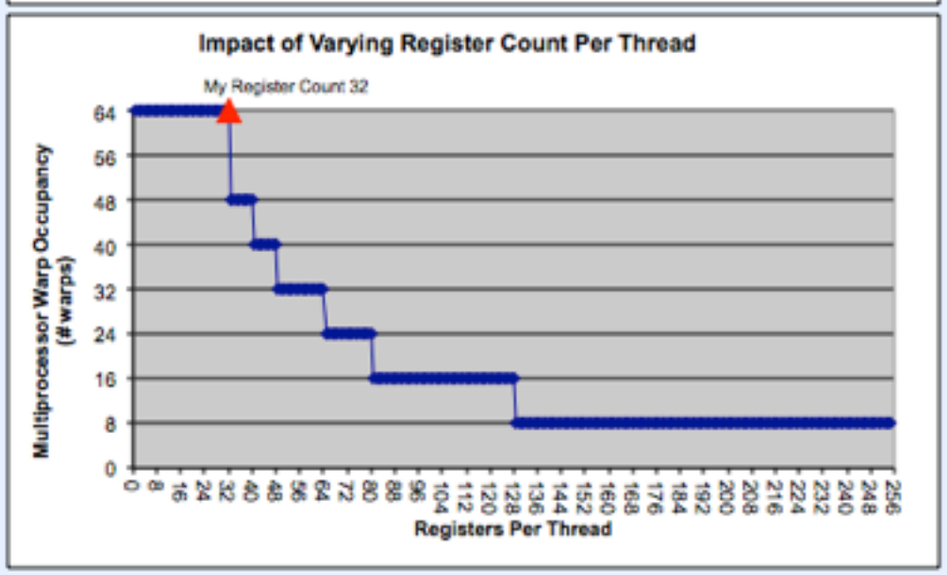
Note: SM is an abbreviation for (Streaming) Multiprocessor

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	64

Note: Occupancy limiter is shown in orange

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





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						

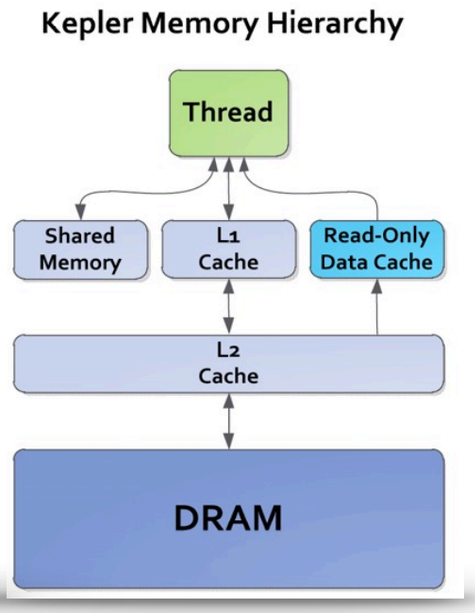
Note the Kepler SMX can operate in 32-bit or 64-bit bank mode.

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.

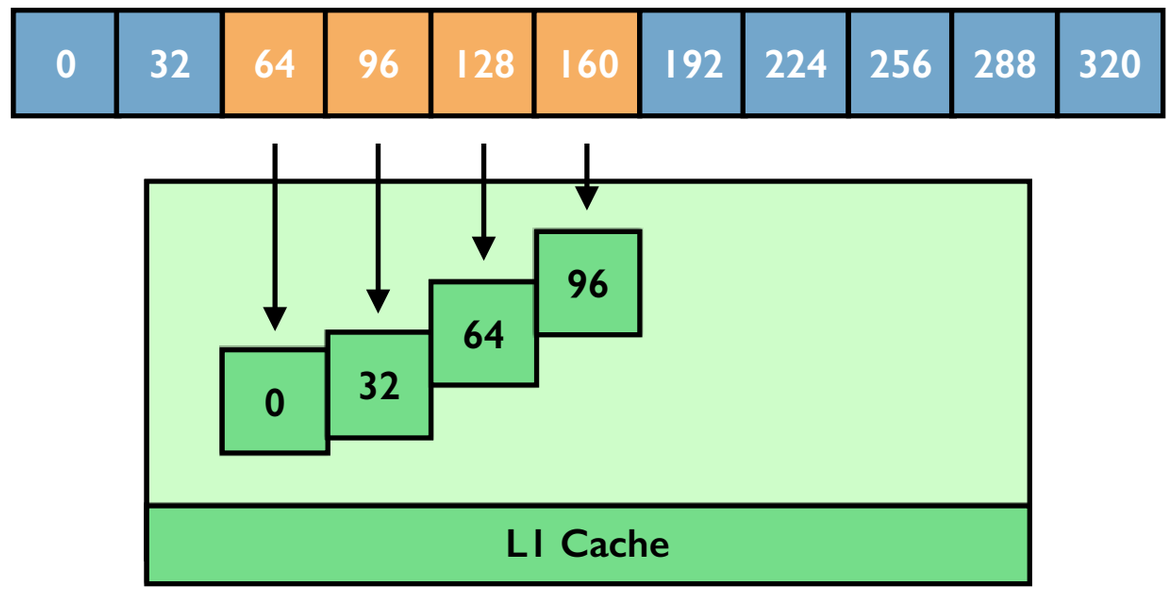


Rule of thumb: avoid non unitary stride DEVICE (DRAM) array access.
Useful slides , these , and image credit: link

CPU Optimization Techniques

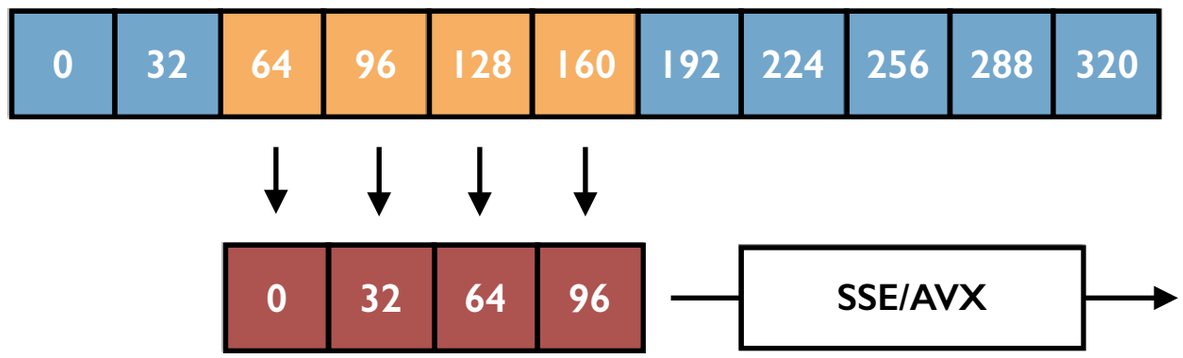
Cache

- 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.

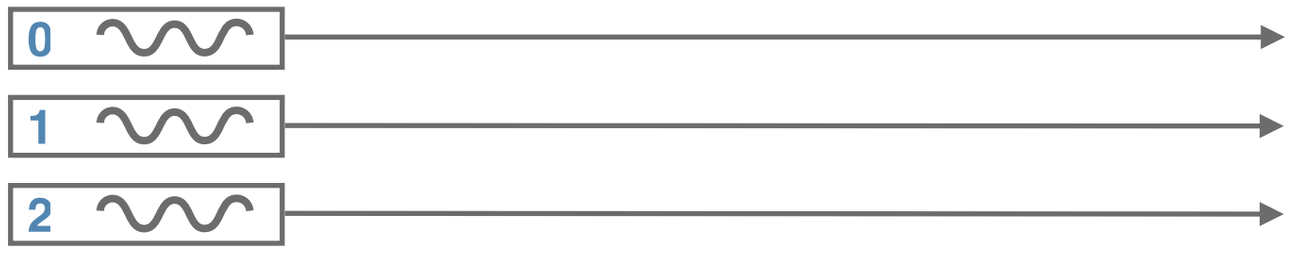


Continuous memory accesses are used for both, cache storage and vectorization

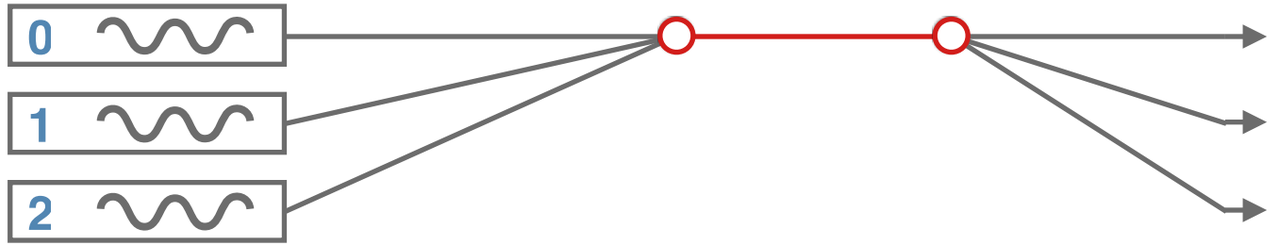
CPU Optimization Techniques

Multithreading

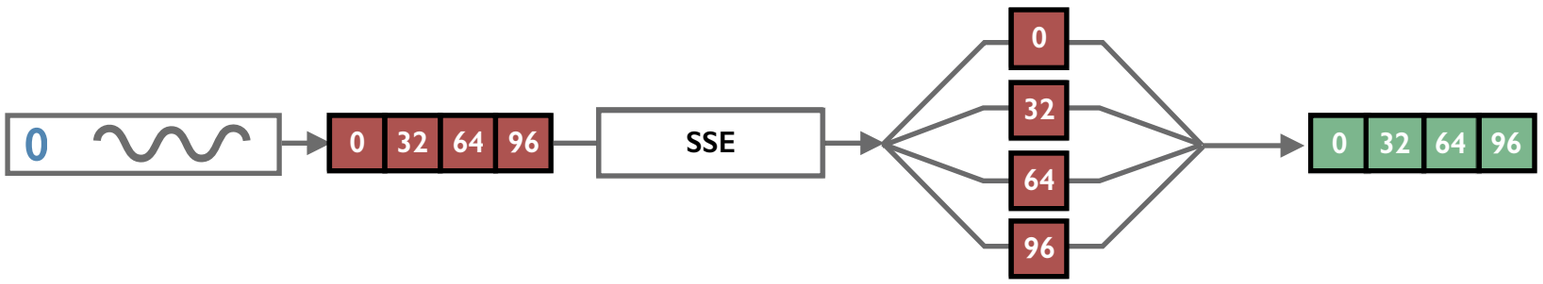
- Threads capable of fully parallelizing generic instructions (ignoring bandwidth).



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



- SIMD Lanes

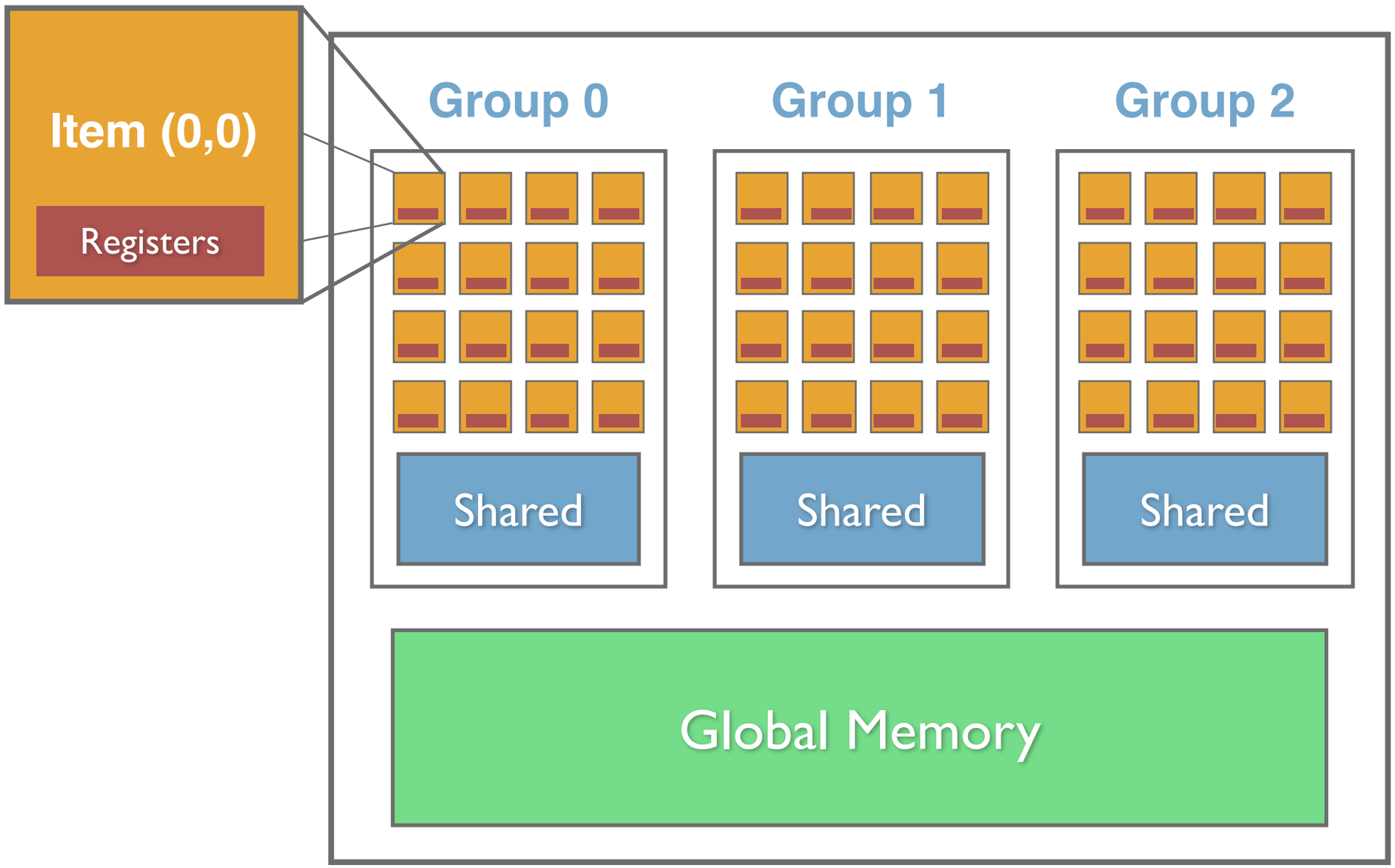


Executing parallel instructions using multithreading

GPU Optimization Techniques

GPU Architecture

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

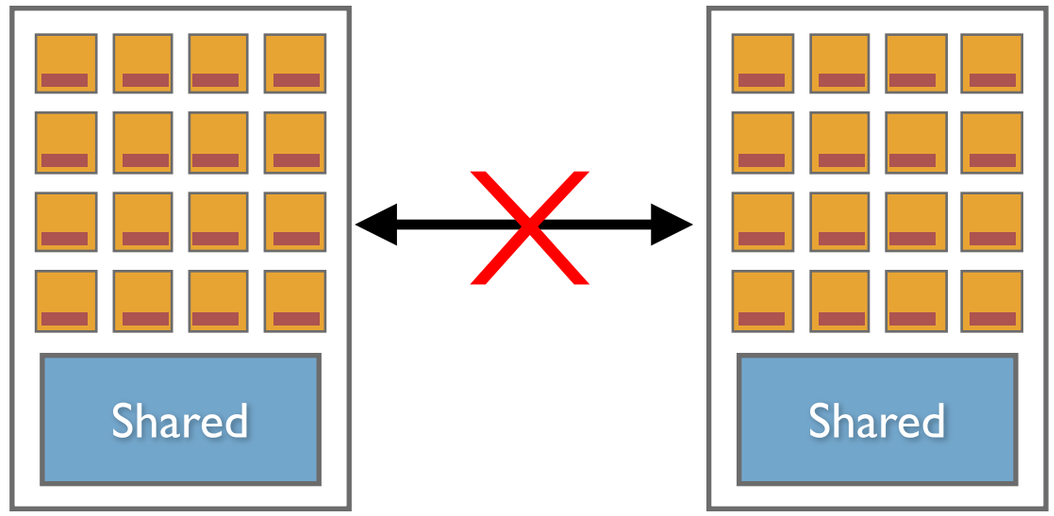


Kernel code describes the work-item operations

GPU Optimization Techniques

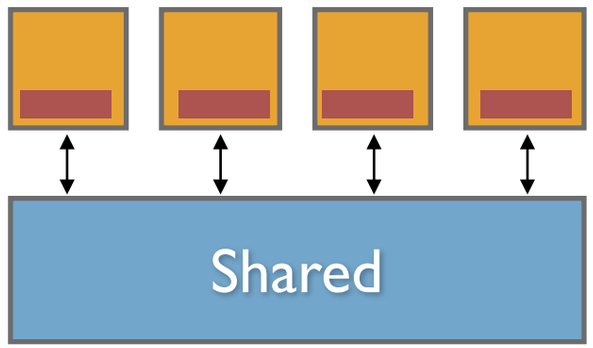
Work-groups

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



Work-items

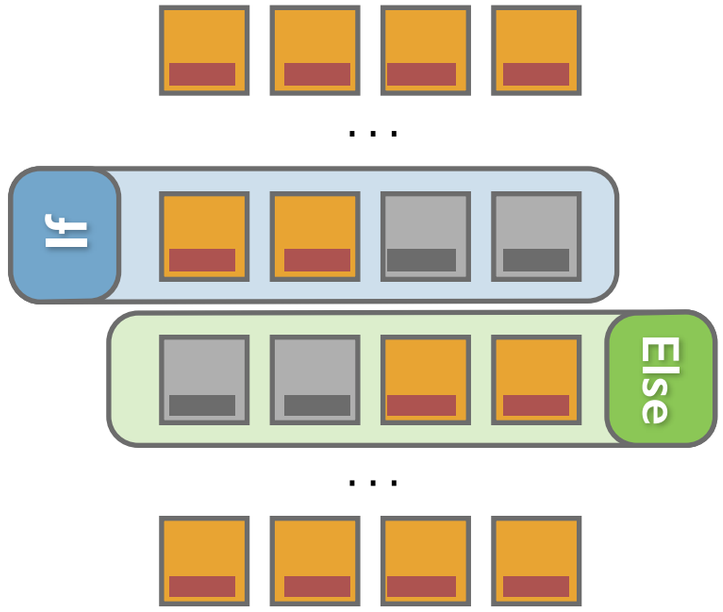
- Work-items are executed in parallel, able to barrier and share data using shared memory (& CUDA's shuffle).
- Avoid data race dependencies between work-items.



GPU Optimization Techniques

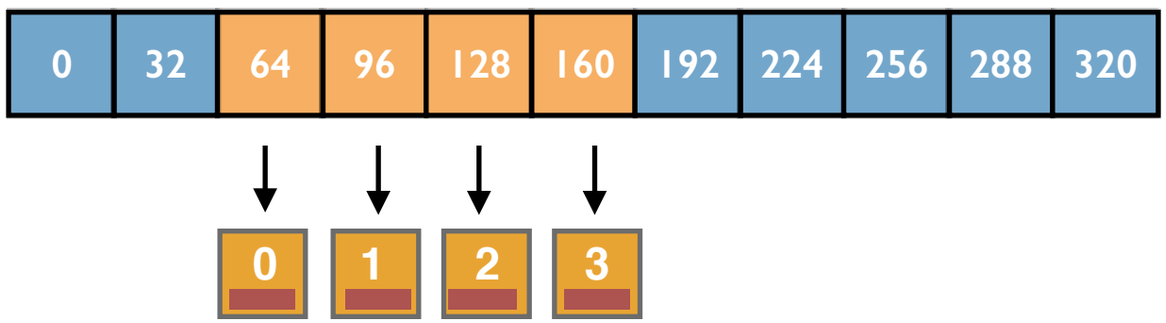
Parallel Work-item Execution

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



Data Transfer

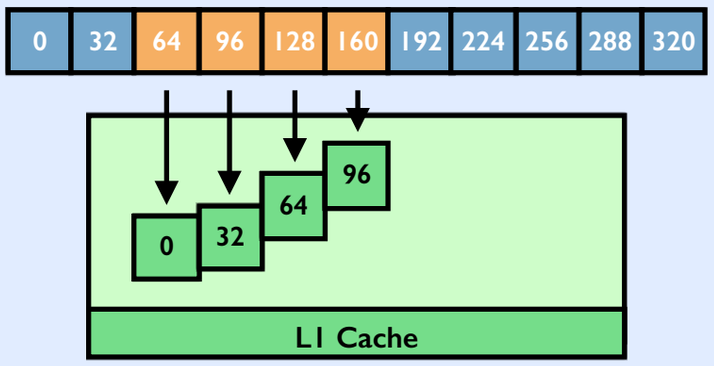
- Low individual bandwidth and high latency.
- Coalesced memory access on contiguous and aligned work-items.



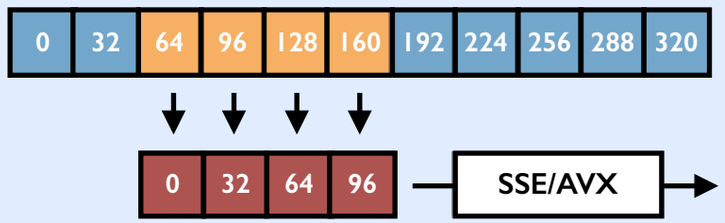
CPU & GPU Similarities

CPU Optimizations

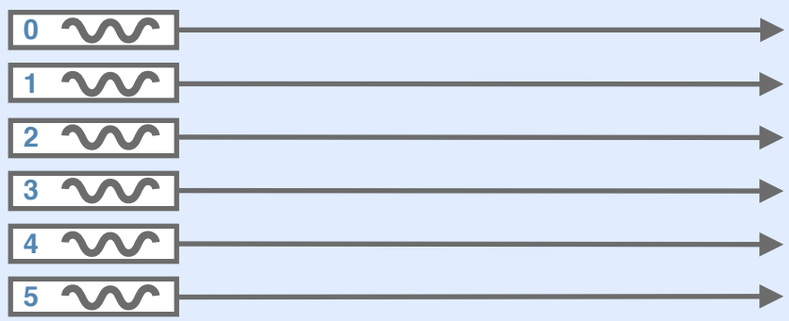
Cache



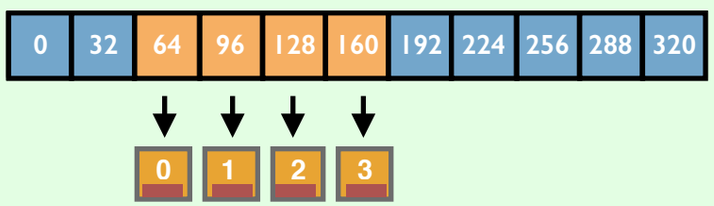
Vectorization



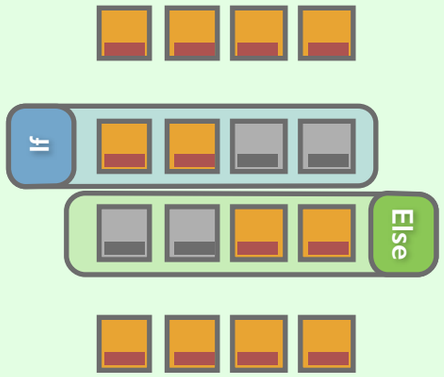
Thread Independence



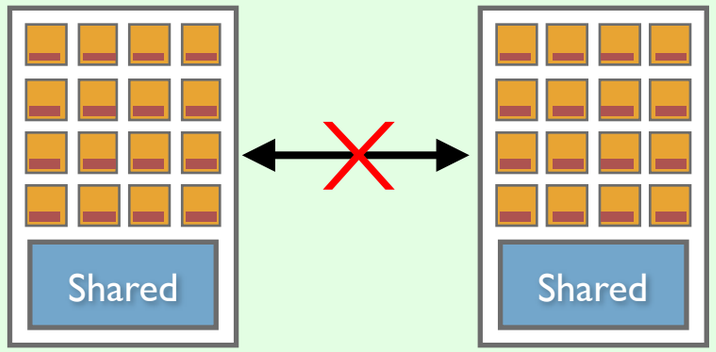
Coalescing



No Branching



Work-group Independence



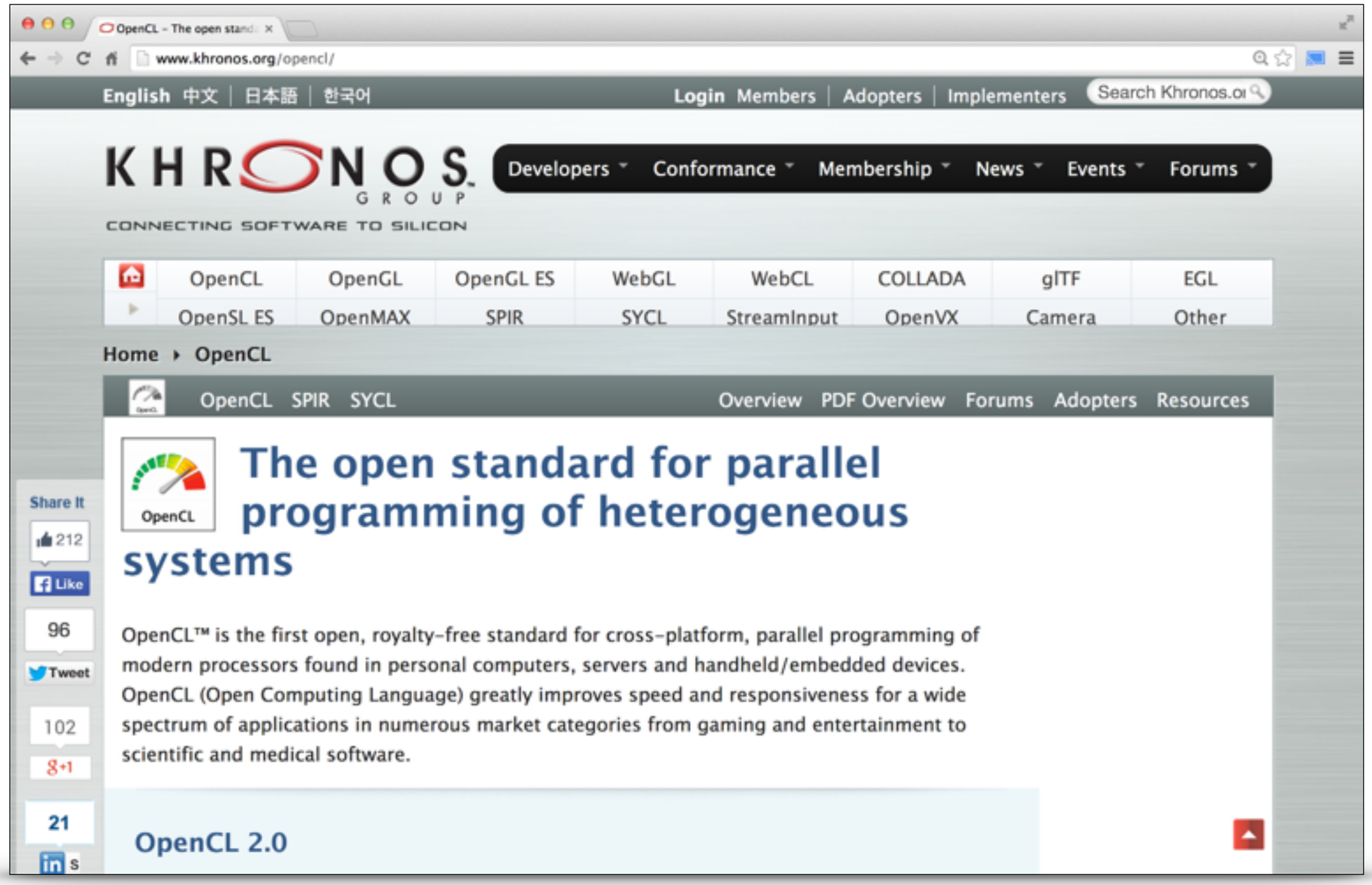
GPU Optimizations

Exposing vectorization / SIMD parallelism are vital in both architectures

Optional Part 3: 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)

KHRONOS GROUP

OpenCL

OpenCL

AMD

The Khronos Group administers the OpenCL standard.

OpenCL: who ?

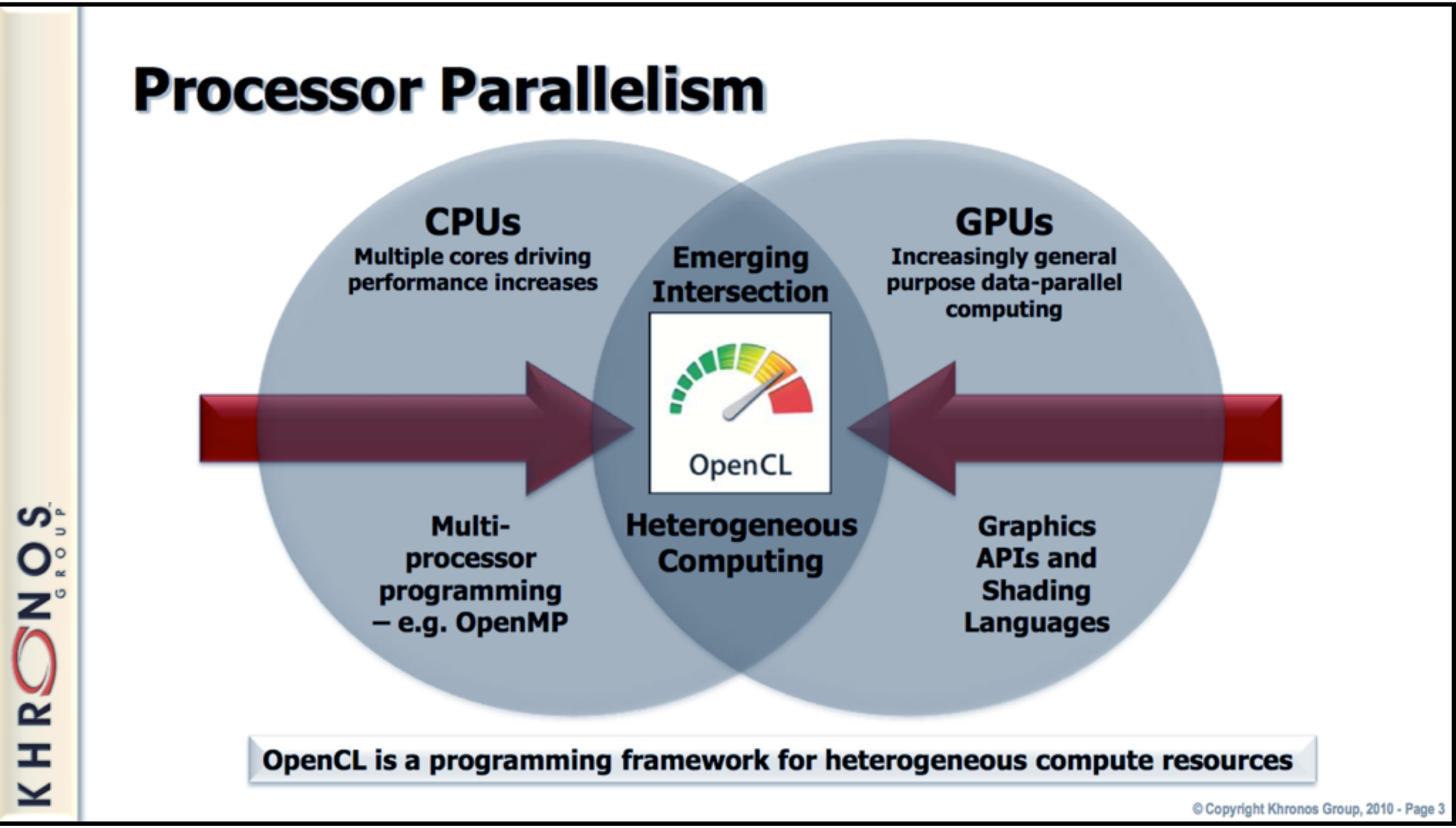
OpenCL Working Group

- **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 ?



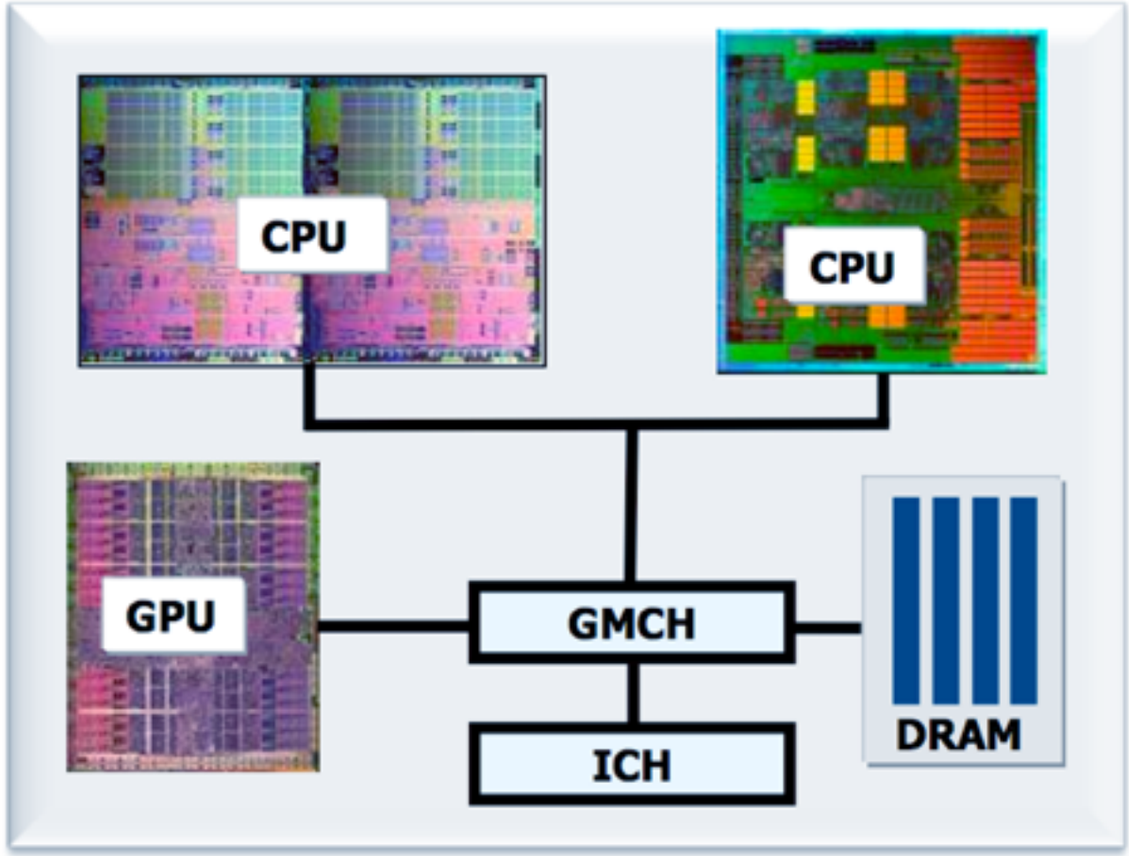
Emphasis on heterogeneous computing.
Credit: Khronos Group

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

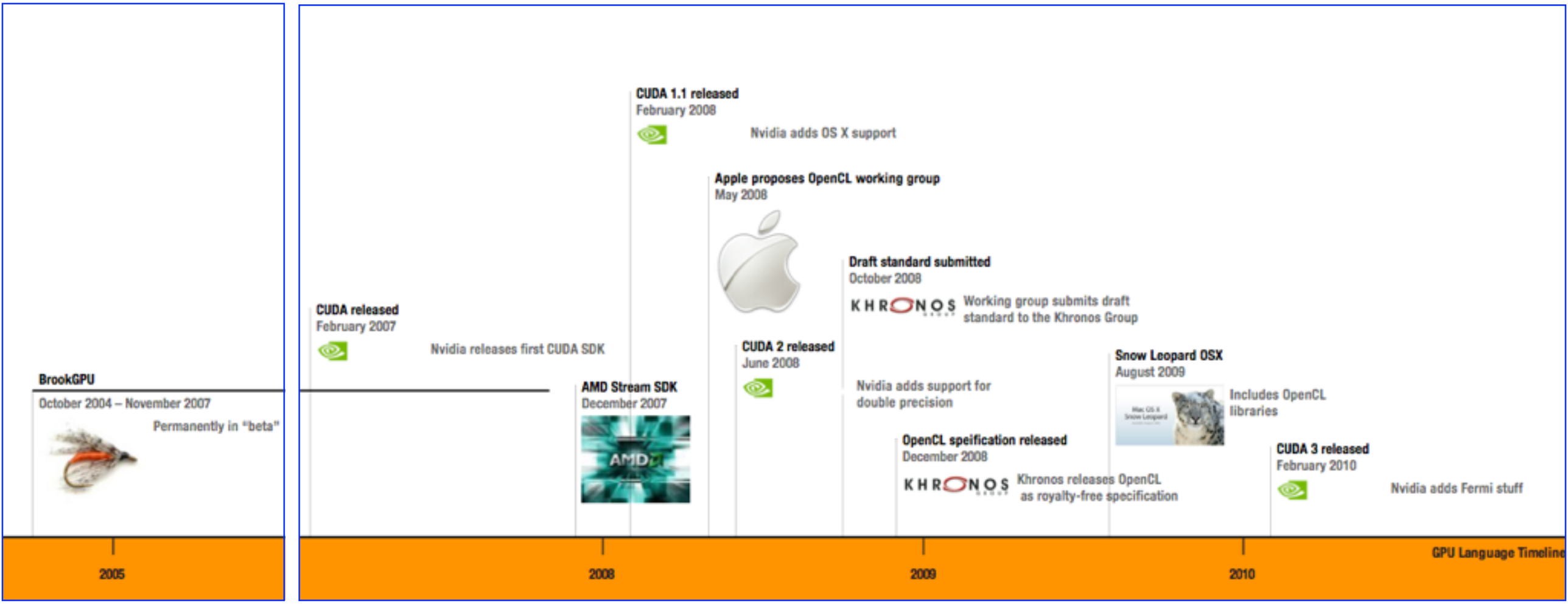


GMCH = graphics memory control hub
ICH = Input/output control hub

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

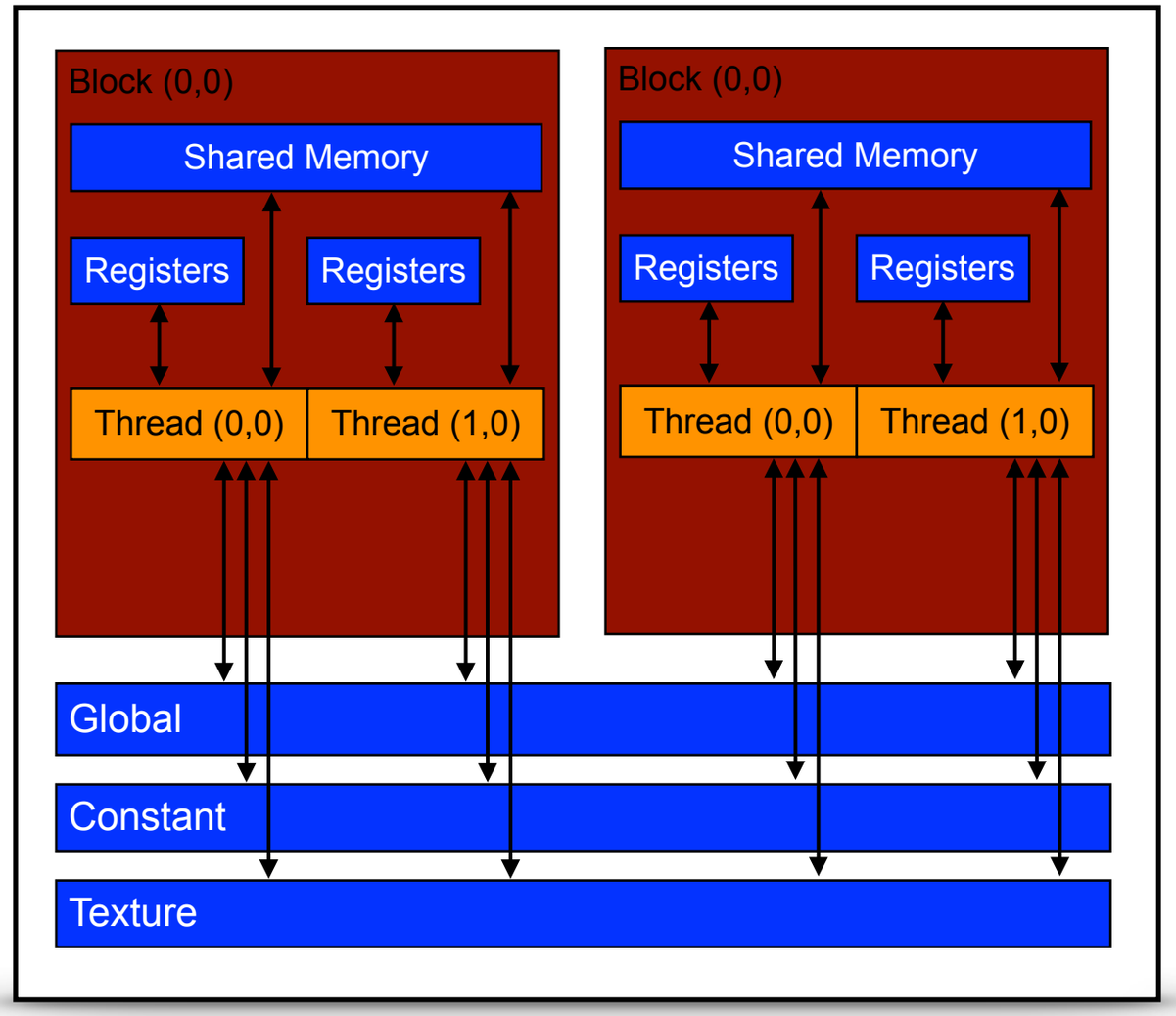
OpenCL is ** very ** closely related to CUDA

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

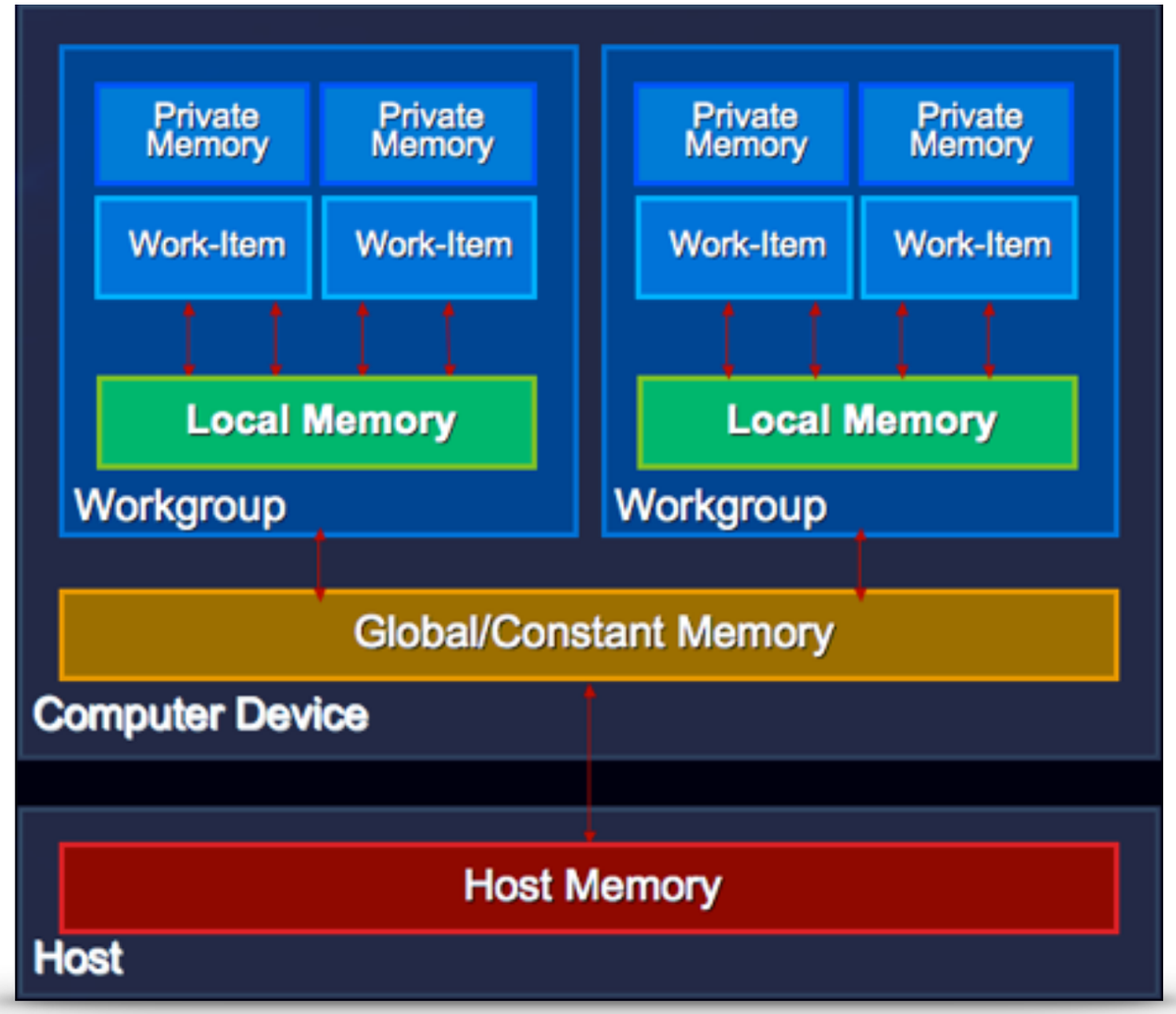
The rapid development of OpenCL helps explain the similarities

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

…
```

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 ATPESC 14 github

ATPESC14/simple.cpp at m

GitHub, Inc. [US] https://github.com/tcew/ATPESC14/blob/master/examples/opencl/simple/simple.cpp

This repository | Search or type a command | Explore | Gist | Blog | Help | tcew

tcew / ATPESC14 — Unwatch 2 | Star 0 | Fork 0

branch: master — ATPESC14 / examples / opencl / simple / simple.cpp

Tim Warburton 2 minutes ago clean up

0 contributors

169 lines (127 sloc) | 4.692 kb — Raw | Blame | History

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/stat.h>
4
5  #ifdef __APPLE__
6  #include <OpenCL/OpenCl.h>
7  #else
8  #include <CL/cl.h>
9  #endif
10
11 void pfn_notify(const char *errinfo, const void *private_info, size_t cb, void *user_data)
12 {
```

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/ATPESC14/examples/cuda/simple

OpenCL: comparing Jacobi kernels

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

Iterate:

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

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!!

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 ATPESC14/examples/opencl/jacobi/reduce.cl is more verbose

Part 4: OCCA

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

Popular Multi-threading APIs

Brief overview of subset of multi-threading approaches.

OpenMP:

- Shared memory model.
- Directive based parallelization.
- OpenMP Architecture Review Board.
- Supports: CPUs + Intel Xeon Phi.
- GPU support (OpenMP 4).

CUDA:

- API manages host-device transfers and on-device calculations.
- Kernel based calculations.
- Proprietary language by NVIDIA.
- Supports: NVIDIA GPUs [and x86].

OpenCL:

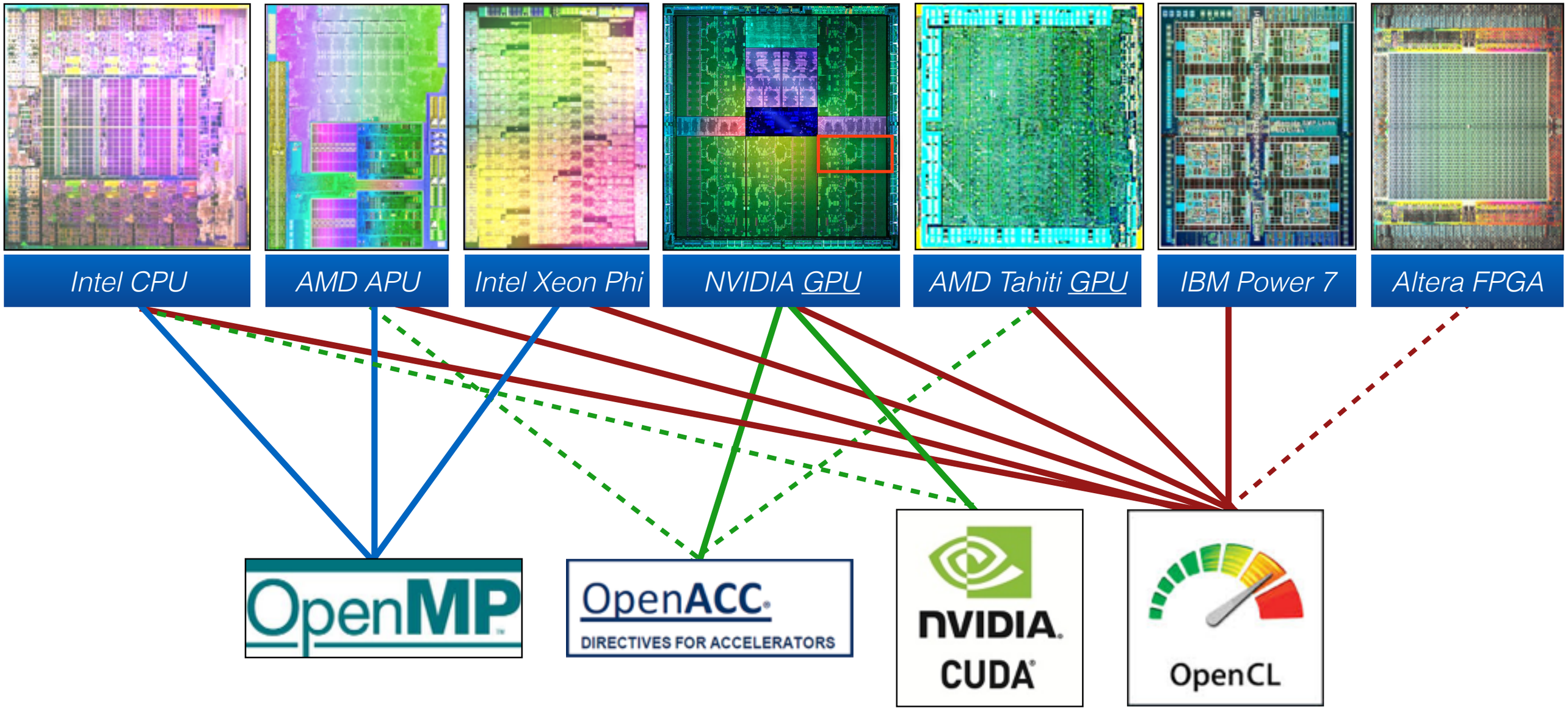
- Similar structure to CUDA.
- Standard by the Khronos Group.
- Compilers: library based.
- Supports: almost all CPUS, GPUs, FPGAs, and Accelerators.

OpenACC:

- Directive based accelerator coding.
- Compilers: Cray, ~~CAPS~~, PGI/NVIDIA.
- In progress: gcc+OpenACC link
- Evolving: may merge with OpenMP ?
- Supports: accelerators/GPUs/…

Many-core Smorgasbord

There is a zoo of competing architectures for many-core devices…



... and a range of thread programming models (with vendor favorite pairs).

Can we insulate against uncertainty and fragmentation ?
Can we avoid maintaining multiple code bases ?

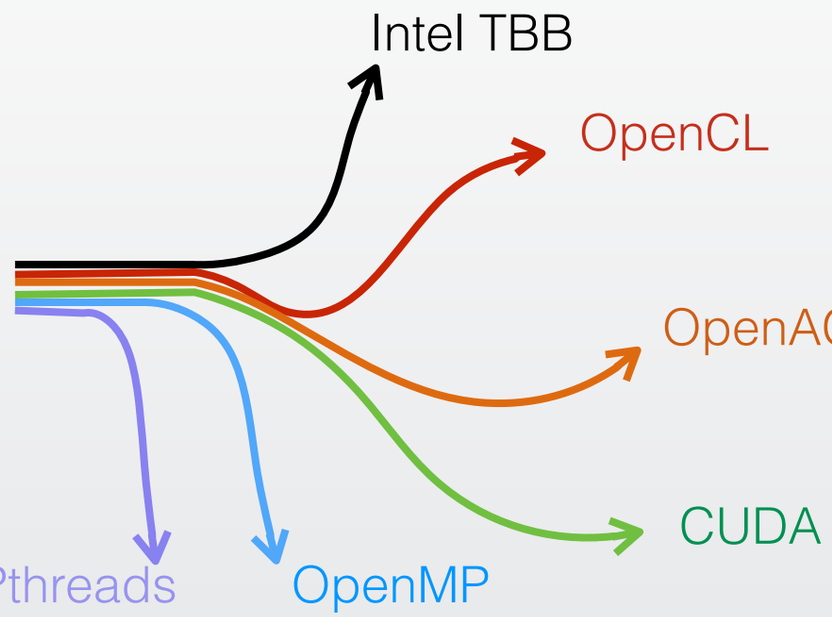
Many-core Challenges

Uncertainty:

- Code life cycle measured in decades.
- Architecture & API life cycles measured in Moore doubling periods.
- Example: if you coded for the IBM Cell processor API…

Fragmentation:

- Pthreads, CUDA, OpenCL, OpenMP, OpenACC, Intel TBB… are not source code compatible.
- Not all APIs are installed on all systems.



Performance*:

- Naively porting between OpenMP, CUDA, OpenCL may yield low performance.
- Manufacturers devote resources to enhancing specific APIs.
- High level APIs may induce excess data movement.

Parallelism*:

- The programmer has to expose parallelism in memory access and operations.

We need a simple and uniform approach so codes can run with CUDA, OpenCL, Pthreads, or OpenMP backends automatically.

Subset of Approaches to Portability

Numerous approaches to portability

API	Type	Front-ends	Kernel Language	Back-ends
Kokkos	ND arrays	C++	Custom	CUDA & OpenMP
VexCL	Vector class	C++	-	CUDA & OpenCL
OCCA 2.0	API	C,C, F90, Python,MATLAB,Julia (C#,Objective C)	Custom Unified Kernel Language	CUDA, OpenCL, pThreads, COI & OpenMP
CU2CL	Source-to-source	App	CUDA	OpenCL
Insieme	Intermediate Representation	C	OpenMP,Cilk,MPI, OpenCL	OpenCL,MPI, Insieme IR runtime
OmpSs	Directives + kernels	C,C++	Hybrid OpenMP, OpenCL, CUDA	OpenMP, OpenCL, CUDA
Ocelot	PTX Translator	CUDA	CUDA	OpenCL

DSL

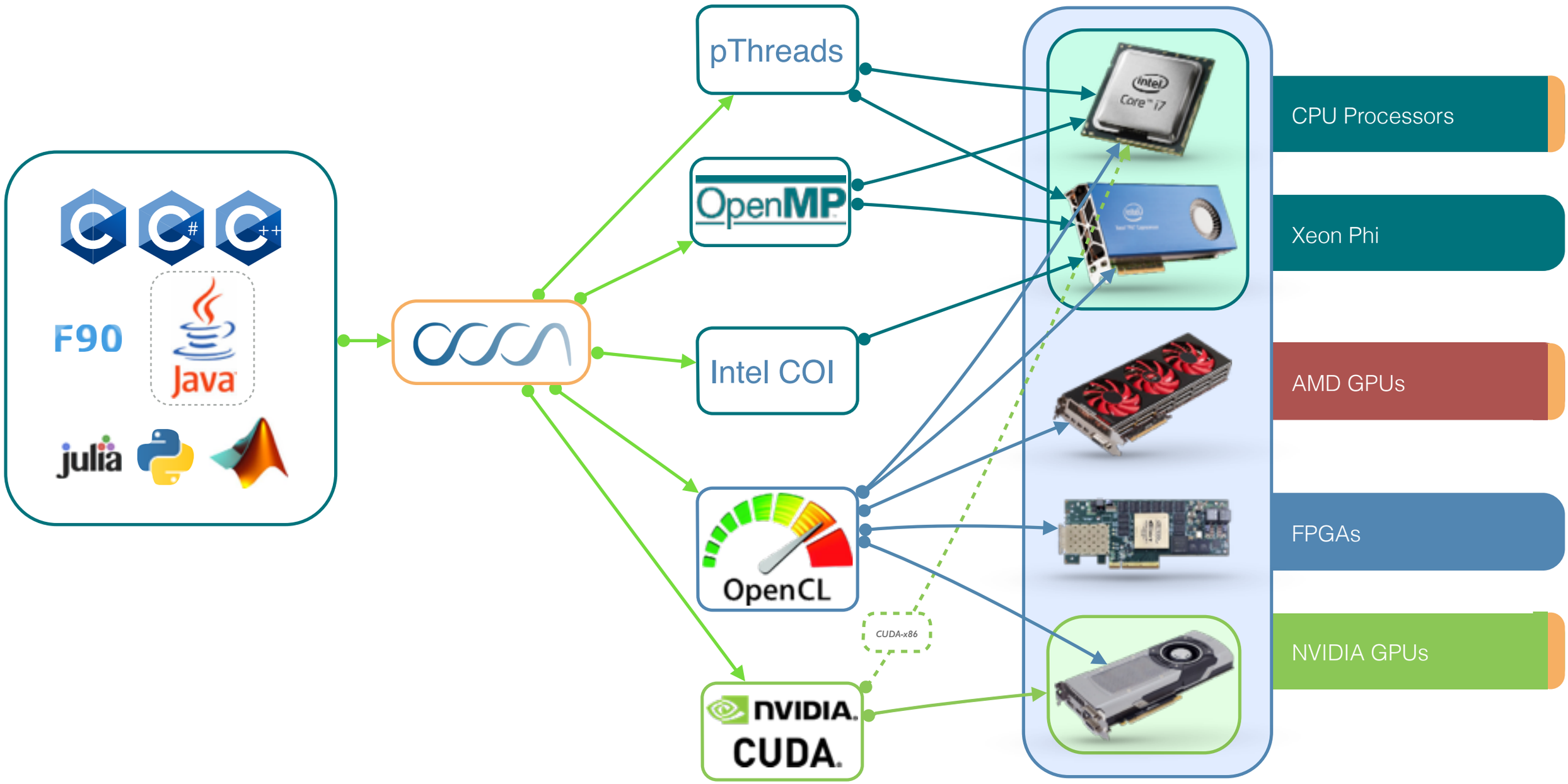
Compiler

Also RAJA from LLNL with OpenACC & OpenMP back-ends.

OCCA emphasis: lightweight and extensible.

OCCA2: unified threading model

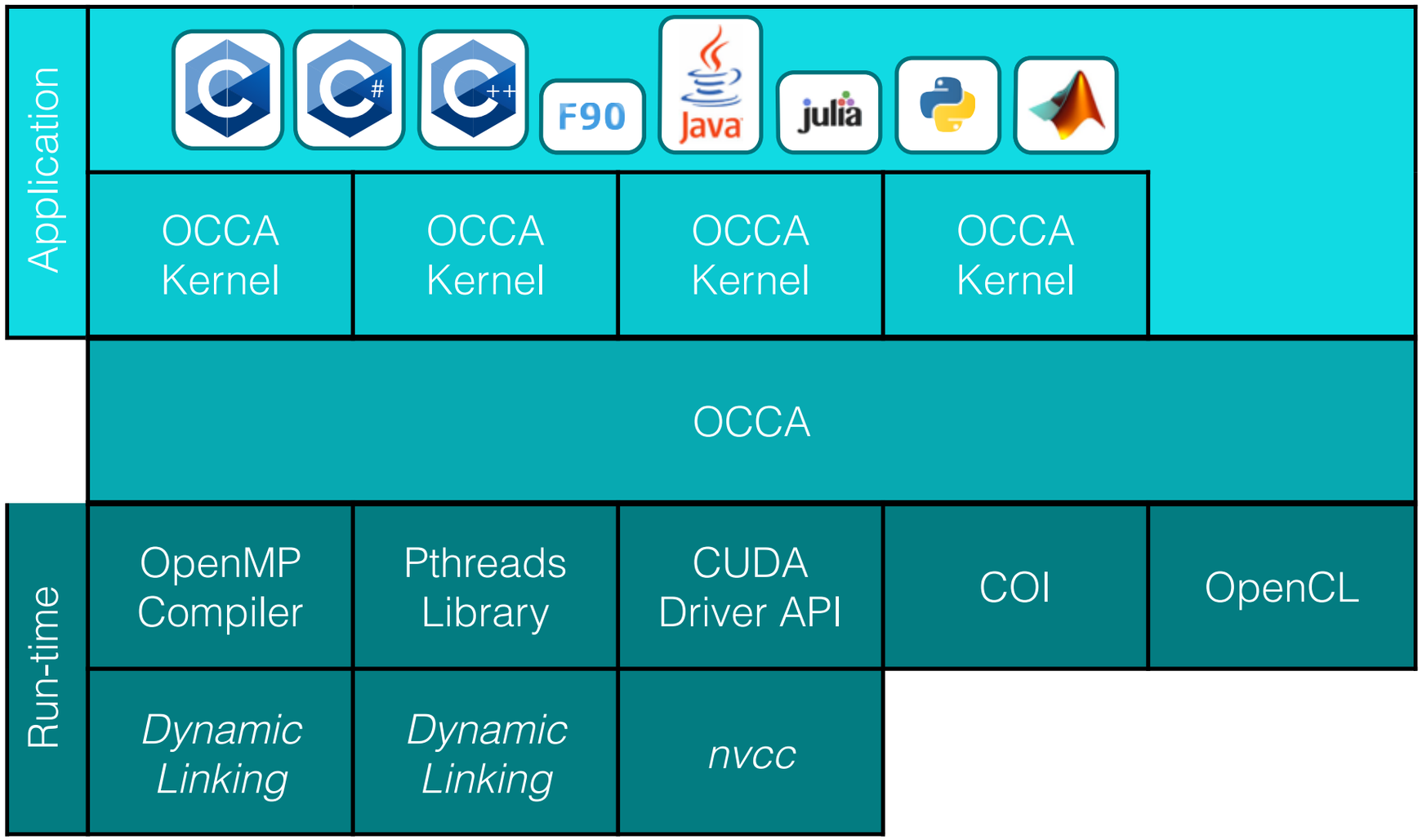
Portability & extensibility: device independent kernel language and native host APIs.



https://github.com/tcew/OCCA2

OCCA2: API stack

OCCA2 communicates with several different run-times.



Continuing to add new front-ends and back-ends.
Access to pre-alpha available on request.

OCCA2: abstractions

There are 4 main abstractions: API, device, memory, and kernel.

occa

- Namespace encapsulating OCCA2 objects and functionality.

occa::device

- Generic compute DEVICE that interacts with multiple concrete backends.
- Thread model, platform*, and device* chosen at run-time.

occa::memory

- Generic memory object (i.e. array) accessible on DEVICE
- Manages data transfers between HOST and DEVICE.
- Garbage collection is done manually.

occa::kernel

- Generic callable kernel function.
- Kernels are compiled from source at run-time.
- Compiled kernels are cached [hashing is used to detect changes in compiler options, source code …].

** where appropriate.*

OCCA2 v. CUDA: 1-1 map

Side by side code demos 1-1 map between OCCA2 and a subset of CUDA

CUDA

```
CUDA runtime API code;
```

OCCA2

```
OCCA2 runtime API code;
```

In reality OCCA2 is more closely related to the CUDA Driver API.
We use a gray color scheme for OCCA2 to denote the averaging of the other APIs 😃.

OCCA2: HOST code API headers

The include files …

CUDA

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

OCCA2

```
#include “occa.h”
```

OCCA2: choosing a device

Next we choose a device supported by the platform.

CUDA

```
…

int dev = 0;
cudaSetDevice(dev);

…
```

OCCA2

```
…

// in future versions there will be an enhanced platform/device selector
int plat = 0, dev = 1;

// notice that we pull the device object type out of the occa namespace
occa::device device;

// choose threading model from “OpenCL”, “CUDA”, “OpenMP”, “Pthreads”, “COI”
device.setup(“OpenCL”, plat, dev); // or …
device.setup(“CUDA”, dev);
…
```

At runtime we can choose the platform, device, and appropriate threading model.

OCCA2: stream management

CUDA

We can create streams [for kernel overlap on Kernel & for async data movement]

```
…

// not necessary although you may wish to use cudaStreamCreate
cudaStream_t stream;
cudaStreamCreate (&stream);

…

cudaStreamDestroy(stream);

…
```

OCCA2

```
…
// generate an instruction stream
occa::stream stream;
stream = device.genStream();

…
device.setStream(stream);
…

device.freeStream(stream);
…
```

OCCA2 example: https://github.com/tcew/OCCA2/tree/master/examples/usingStreams

OCCA2: building a kernel

We build the kernels at run-time to match the platform/device/thread-model.

CUDA

```
…

// not necessary [ single vendor solution ]

…
```

OCCA2

```
…

// create program from source
occa::kernel knl = device.buildKernelWithSource(“foo.occa”, “foo”);

// NOTE: the source may not compile and error messages may result

// Live demo.

…
```

Optional third argument for device.buildProgramWithSource allows us to pass in compiler flags.
[OCCA2 uses nvcc & the CUDA driver API functions: cuModuleLoad, cuModuleGetFunction]

OCCA2: thanks for the memory

We next allocate array space on the DEVICE:

CUDA

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

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

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

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

OCCA2

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

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

// create device buffer and copy from host buffer
occa::memory c_x = device.malloc(sz);
```

We can also provide a pointer as an optional second argument to the device malloc.

OCCA2: array life cycle

```
size_t sz = N*sizeof(float)
float *h_x = (float*) malloc(sz); // allocate HOST array
```

CUDA

```
cudaMalloc((void**) &c_x, sz);    // allocate DEVICE array

// copy from HOST to DEVICE
cudaMemcpy(c_x, h_x, sz, cudaMemcpyHostToDevice);

// copy from DEVICE to HOST
cudaMemcpy(h_x, c_x, sz, cudaMemcpyDeviceToHost);

// free DEVICE array
cudaFree(c_x);
```

OCCA2

```
occa::memory c_x = device.malloc(sz); // allocate DEVICE array

// copy from HOST to DEVICE
c_x.copyFrom(h_x); // or occa::memcpy(c_x, h_x);

// copy from DEVICE to HOST
c_x.copyTo(h_x);   // or occa::memcpy(h_x, c_x);

// free DEVICE array
c_x.free();
```

We can also provide a pointer as an optional second argument to the device malloc.

OCCA2: specifying thread array

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
knl <<< dimGrid, dimBlock >>> (N, c_x);
…
```

OCCA2

```
…
// set one-dimensional thread array
int dims = 1;
occa::dim inner(256,1,1);
occa::dim outer((N+255)/256,1,1);

// set up work dimension
knl.setWorkingDims(dims, inner, outer);

// queue up kernel
knl(N, c_x);
…
```

Currently in OCCA2: working dimensions == thread array size.
Eventually these will be specified separately.

OCCA2: simple kernel example

The kernel programming languages are similar:

CUDA

```
__global__ void knl(int N,
                    float *a)
{

  /* get thread coordinates */
  int i = threadIdx.x +
          blockIdx.x*blockDim.x;

  /* do simple task */
  if(i<N)
     a[i] = i;

}
```

OCCA2

```
occaKernel void knl(occaKernelInfoArg,
                    int occaVariable N,
                    occaPointer float *a)
{

 occaGlobalFor0{

    /* get thread coordinates */
    int i = occaGlobalId0;


    /* do simple task */
    if(i<N)
      a[i] = i;
  }

}
```

Some differences in syntax & identifiers.
Main difference: notice the explicit parallel for macros (occaOuterFor0, occaInnerFor0).

OCCA2: thread indexing

OCCA2 is ** very ** closely related to CUDA

CUDA		OCCA2	
Local indices:		Local indices:	
threadIdx.x	threadIdx.y	occaInnerId0	occaInnerId1
Block indices:		Block indices:	
blockIdx.x	blockIdx.y	occaOuterId0	occaOuterId1
Global indices:		Global indices:	
blockIdx.x*blockDim.x + threadIdx.x	blockIdx.y*blockDim.y + threadIdx.y	occaGlobalId0	occaGlobalId1

Third dimension also available.

OCCA2: thread array dimensions

OCCA2 is ** very ** closely related to CUDA

CUDA	OCCA2
gridDim.x	occaOuterDim0
blockIdx.x	occaOuterId0
blockDim.x	occaInnerDim0
gridDim.x*blockDim.x	occaGlobalDim0

Analogs for x/y/z and 0/1/2 directions.

OCCA2: kernel language qualifiers

OCCA, CUDA, and OpenCL are very closely related.

CUDA	OCCA2
__global__ void function	occaKernel void function
__device__ function	occaDeviceFunction
__constant__	occaConstant
__device__	occaPointer
__shared__	occaShared
const	occaConst

No surprises here.

OCCA2: misc.

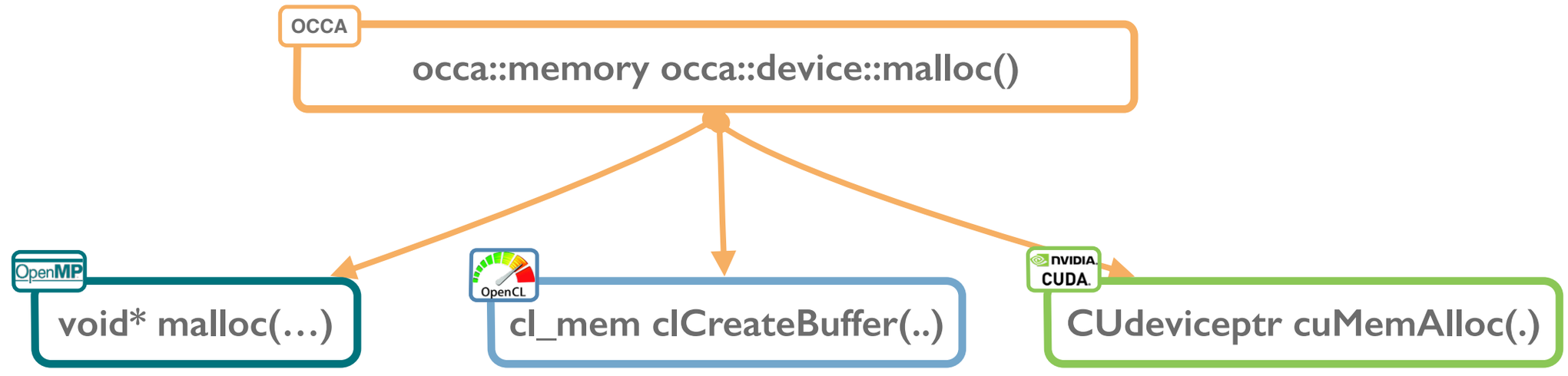
OCCA assumes barriers are really memory fences.

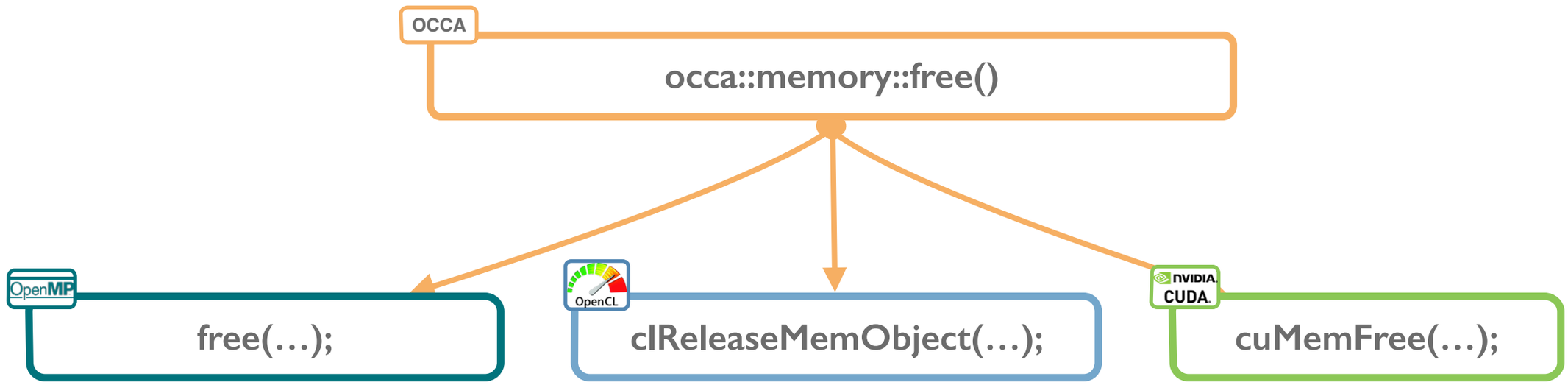
CUDA	OCCA2
__syncthreads();	occaBarrier(occaLocalMemFence);
__syncthreads();	occaBarrier(occaGlobalMemFence);
cudaThreadSynchronize();	occa::device::finish

No surprises here.

OCCA2: C++ API behind the scenes

occa::memory class: abstracts the memory handles found in each language

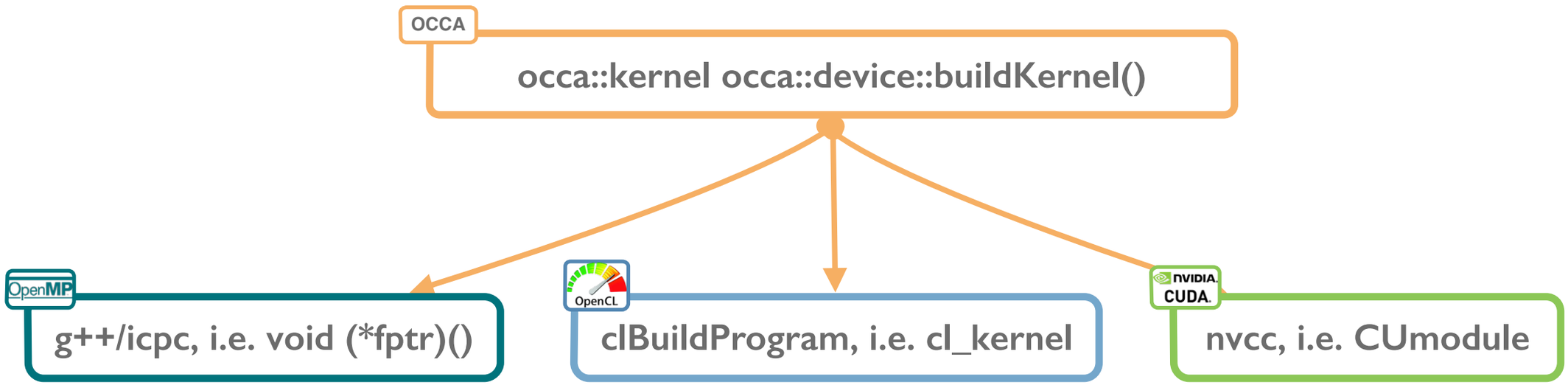




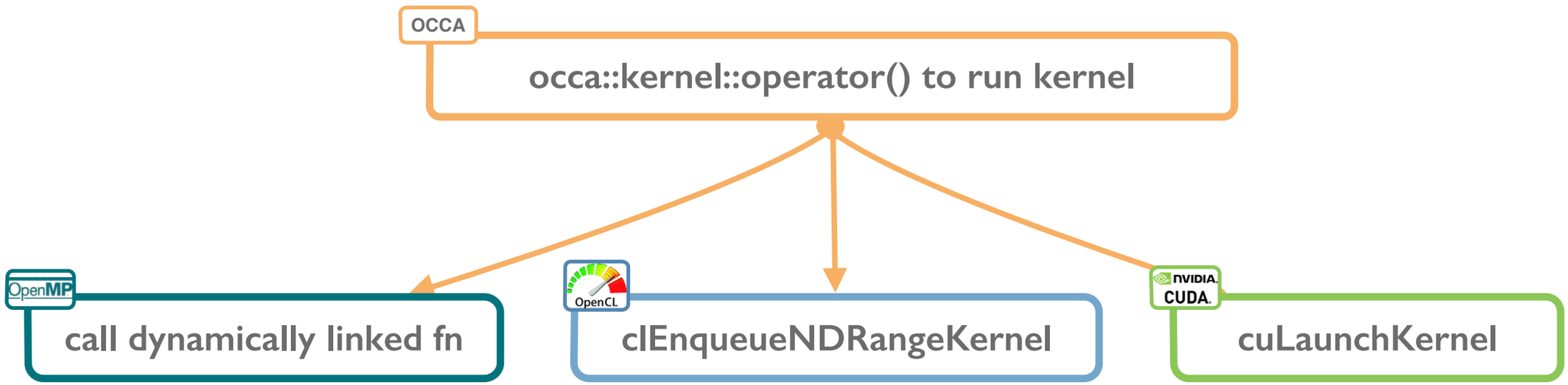
OCCA uses the CUDA Driver API for run-time compilation of OCCA kernels as CUDA kernels.

OCCA2: C++ API behind the scenes

occa::kernel class: encapsulates function handles and uses run-time compilation



occa::kernel class: encapsulates function handles and uses run-time compilation



OCCA uses the CUDA Driver API for run-time compilation of OCCA kernels as CUDA kernels.

OCCA2: exposed kernel loop structure

OCCA Kernel API

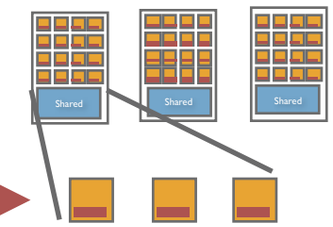
- Relies on macros for masking the different supported languages.
- Uses the GPU programming model of work-groups and work-items.

Explicit Work-groups and Work-Items

- Work-group and work-item are explicitly expressed as OCCA for-loops.

```
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
```

To do: improve OpenMP loop partitioning.

OCCA2: shared memory and registers

Shared memory

- Shared memory crucial for GPU-modes (requires barrier for syncing work-items).
- Manually caching in CPU-mode.

```
occaInnerFor2{
  occaInnerFor1{
    occaInnerFor0{

      set entries in shared memory array;
    }
  }
}

occaBarrier(occaLocalMemFence);

occaInnerFor2{
  occaInnerFor1{
    occaInnerFor0{

      use entries in shared memory array;
    }
  }
}
```

Register memory

- Fastest memory on GPUs.
- Another issue with for-loop scopes … require thread-local-storage-type of data.

OCCA2: private variables

occaPrivate Classes

- Private variables are locally scoped variables in OpenCL & CUDA.
- In OpenMP and pThreads private variables have hidden arrays (one entry per thread).
- The hidden array entries are indexed by local thread index.

CUDA

```
__global__ void knl(int N, float *a)
{

  /* get thread coordinates */
  int i = threadIdx.x + blockIdx.x*blockDim.x;

  /* write to private variable */
  float f = i;

  /* barrier for some reason */
  __syncthreads();

  /* write result */
  if(i < N) a[i] = f;
}
```

The __syncthreads barrier is effectively making sure all threads have reached that line in the kernel.

OCCA2: private variables & barrier

Example: barrier and private variables in OCCA2

```
occaKernel void knl(occaKernelInfoArg, int occaVariable N, occaPointer float *a)
{
  occaOuterFor0 {

    // instantiate two private variables
    occaPrivate(int, i);
    occaPrivate(float, f);

    occaInnerFor0 {
      i = occaGlobalId0; // get thread coordinates

      f = i; // write to private variable for this thread
    }

    // barrier for some reason
    occaBarrier(occaLocalMemFence);

    occaInnerFor0 {

       if(i < N) a[i] = f; // both i & f survived the barrier
    }
}
```

OCCA private variables maintain shadow arrays with entries for each thread

1. i and f behave like thread local variables inside the occaInner loops.
2. in order to barrier we need to close the inner loop(s), barrier, then open new inner loops.

OCCA2: overview of the kernel model

Goal: one implementation for each kernel.

```
__global__ void kernelFunction(arguments){

  // embarrassingly parallel outer loops (implicit)

  // shared memory for collaboration between threads
  __shared__ float s_a[100];

  // embarrassingly parallel inner loops (implicit)
  kernel code here;

  // synchronization writes to shared memory
  __syncthreads();

  // embarrassingly parallel inner loops
  kernel code here;
}
```

```
__kernel void kernelFunction(arguments){

  // embarrassingly parallel outer loops implicit

  // shared memory for collaboration between threads
  __local float s_a[100];

  // embarrassingly parallel inner loops (implicit)
  kernel code here;

  // synchronization writes to shared memory
  barrier(CLK_LOCAL_MEM_FENCE);

  // embarrassingly parallel inner loops (implicit)
  kernel code here;
}
```

```
occaKernel void kernelFunction(arguments){

  // embarrassingly parallel outer loops (explicit)
  occaOuterFor1 {
    occaOuterFor0 {

      // shared memory for collaboration between threads
      occaShared float s_a[100];

      // embarrassingly parallel inner loops (explicit)
      occaInnerFor2{
        occaInnerFor1{
          occaInnerFor0{

            kernel code here;
          }
        }
      }

      // synchronization writes to shared memory
      occaBarrier(occaLocalMemFence);

      // embarrassingly parallel inner loops (explicit)
      occaInnerFor2{
        occaInnerFor1{
          occaInnerFor0{

            kernel code here;
          }
        }
      }
    }
  }
}
```

The map between thread array <> parallel loop is made explicit.

OCCA2: example adding two vectors

```
#include <iostream>

#include "occa.hpp"

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

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

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

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

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

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

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

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

  addVectors.setWorkingDims(dims, itemsPerGroup, groups);

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

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

OCCA2: example adding two vectors

```
#include <iostream>

#include "occa.hpp"

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

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

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

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

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

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

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

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

  addVectors.setWorkingDims(dims, itemsPerGroup, groups);

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

```
occaKernel void addVectors(occaKernelInfoArg,
                           occaConst int occaVariable entries,
                           occaConst occaPointer float * a,
                           occaConst occaPointer float * b,
                           occaPointer           float * ab){
  occaOuterFor0{
    occaInnerFor0{
      const int N = occaGlobalId0;

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

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

Native Host Codes

```
#include <iostream>

#include "occa.hpp"

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

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

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

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

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

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

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

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

  addVectors.setWorkingDims(dims, itemsPerGroup, groups);

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

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

Native Host Codes

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

#include "occa_c.h"

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

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

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

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

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

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

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

  int dims = 1;
  occaDim itemsPerGroup, groups;

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

  occaKernelSetWorkingDims(addVectors,
                           dims, itemsPerGroup, groups);

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

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

  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

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

Native Host Codes

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

#include "

int main(i
  int i;

  float *a
  float *b
  float *a

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

  occaDevi
  occaKern
  occaMemc

  device =

  o_a  = c
  o_b  = c
  o_ab = c

  addVecto



  int dims
  occaDim

  itemsPer
  groups.x

  occaKern


  occaCopy
  occaCopy

  occaKern


  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

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

entries = 5

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

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

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

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


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

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

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

occa.memcpy(ab, o_ab)

println(ab)
```

All the HOST codes use the same kernel.

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

Native Host Codes

```python
from ctypes import *
import occa

entries = 5

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

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

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

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

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

addVectors.setWorkingDims(dims, itemsPerGroup, groups)

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

o_ab.copyTo(ab, c_float)

print ab
```

```
require( bytestri

entries = 5

device = occa.dev

# Dynamic range?
a  = Float32[1 -
b  = Float32[i
ab = Float32[0

o_a  = occa.mallo
o_b  = occa.mallo
o_ab = occa.mallo

addVectors = occa



dims = 1;
itemsPerGroup = 2
groups = (entries

occa.setWorkingDi


occa.runKernel(ad
               (ent
               o_a, o_b, o_ab)

occa.memcpy(ab, o_ab)

println(ab)
```

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

#include "

int main(i
  int i;

  float *a
  float *b
  float *a

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

  occaDevi
  occaKern
  occaMemo

  device =

  o_a  = o
  o_b  = o
  o_ab = o

  addVecto



  int dims
  occaDim

  itemsPer
  groups.x

  occaKern


  occaCopy
  occaCopy

  occaKern


  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

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

Native

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

#include "

int main(i
  int i;

  float *a
  float *b
  float *a

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

  occaDevi
  occaKern
  occaMemo

  device =

  o_a  = o
  o_b  = o
  o_ab = o

  addVecto



  int dims
  occaDim

  itemsPer
  groups.x

  occaKern


  occaCopy
  occaCopy

  occaKern


  occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

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

```
require( bytestri

entries = 5

device = occa.dev

# Dynamic range?
a  = Float32[1 -
b  = Float32[i
ab = Float32[0

o_a  = occa.mallo
o_b  = occa.mallo
o_ab = occa.mallo

addVectors = occa



dims = 1;
itemsPerGroup = 2
groups = (entries

occa.setWorkingDi


occa.runKernel(ad
                (ent
               o_a, o_b, o_ab)

occa.memcpy(ab, o_ab)

println(ab)
```

```
from ctypes
import occa

entries = 5

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

device = occ

o_a  = devic
o_b  = devic
o_ab = devic

addVectors =


dims = 1
itemsPerGrou
groups = (en

addVectors.s


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

o_ab.copyTo(ab, c_float)

print ab
```

```
entries = 5;

a  = ones(entries, 1);
b  = ones(entries, 1);
ab = zeros(entries, 1);

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

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

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

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

addVectors.setWorkingDims(dims, itemsPerGroup, groups);

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

ab = o_ab(:);

ab
```

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

OCCA2: running in Python 2.6+

export PYTHONPATH=$PYTHONPATH:$OCCA_DIR/lib/

```
from ctypes import *
import occa

entries = 5

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

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

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

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

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

addVectors.setWorkingDims(dims, itemsPerGroup, groups)

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

o_ab.copyTo(ab, c_float)
```

OCCA2: running in Python 2.6+

export PYTHONPATH=$PYTHONPATH:$OCCA_DIR/lib/

```
> python2.7 main.py
Found cached binary of [addVectors.occa] in [/Users/timwar/.occa/76707561f2b670fd]
[1.0, 1.0, 1.0, 1.0, 1.0]

> python2.7
>>> execfile("main.py")
Found cached binary of [addVectors.occa] in [/Users/timwar/.occa/76707561f2b670fd]
[1.0, 1.0, 1.0, 1.0, 1.0]
>>>
>>> % or interactively

> LIVE DEMO ?
```

OCCA2: environment flags

You can choose which APIs OCCA2 builds with

```
export OCCA_CACHE_DIR=~/.occa

export OCCA_DIR=DIRECTORY_CONTAINING_OCCA_CLONE
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OCCA_DIR/lib

export OCCA_OPENCL_ENABLED=1
export OCCA_CUDA_ENABLED=1
export OCCA_OPENMP_ENABLED=1
export OCCA_PTHREADS_ENABLED=1

export OCCA_PTHREADS_COMPILER=g++
export OCCA_PTHREADS_COMPILER_FLAGS=-g

export OCCA_OPENCL_COMPILER_FLAGS="-cl-mad-enable -cl-finite-math-only"

export OCCA_OPENMP_COMPILER="g++"
export OCCA_OPENMP_COMPILER_FLAGS="-O3 -fopenmp"

export OCCA_CUDA_COMPILER="nvcc"
export OCCA_CUDA_COMPILER_FLAGS="-O3 --use_fast_math"
```

Defaults are in OCCA2/scripts/makefile
You can over ride default behavior by setting command line variables.

OCCA2: running

The OCCA2 run-time creates an intermediate file with name that hashes the source code, compiler, & compiler flags.

```
> make > foo
>
> ./simple
OpenCL compiled simple.occa from [/Users/timwar/.occa/i_5d8734bd4942bb43]
h_x[0] = 0
h_x[1] = 1
h_x[2] = 2
h_x[3] = 3
h_x[4] = 4
h_x[5] = 5
…
h_x[14] = 14
h_x[15] = 15
h_x[16] = 16
h_x[17] = 17
h_x[18] = 18
h_x[19] = 19
>
```

The OCCA runtime caches compiled kernels in OCCA_CACHE_DIR

Example code: https://github.com/tcew/OCCA2Tutorial/tree/master/examples/occa/simple

OCCA2: caching and hashing kernels

The OCCA2 run-time checks the cache directory for a suitable precompiled binary to avoid recompiling.

```
>
>
> ./simple
Found cached binary of [simple.occa] in [/Users/timwar/.occa/
5d8734bd4942bb43]
h_x[0] = 0
h_x[1] = 1
h_x[2] = 2
h_x[3] = 3
h_x[4] = 4
h_x[5] = 5
…
h_x[15] = 15
h_x[16] = 16
h_x[17] = 17
h_x[18] = 18
h_x[19] = 19
>
```

The next time we run this code the OCCA run-time finds the precompiled kernel

This can be important when the number of available compiler licenses is limited or when running at scale on a large system.

OCCA2: using raw CUDA kernels

Undocumented use case for practical development.

- OCCA2 applies macros to translate kernel code into the specified threading language.
- CUDA kernels can be tested by adding occaKernelInfoArg to the kernel arguments….
- … and by making the kernel extern “C”.

```
// build simple kernel using slightly modified CUDA kernel code
occa::kernel simple = device.buildKernelFromSource("simple.cu", "simple");
```

```
extern “C” __global__ void simple(occaKernelInfoArg, 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;
}
```

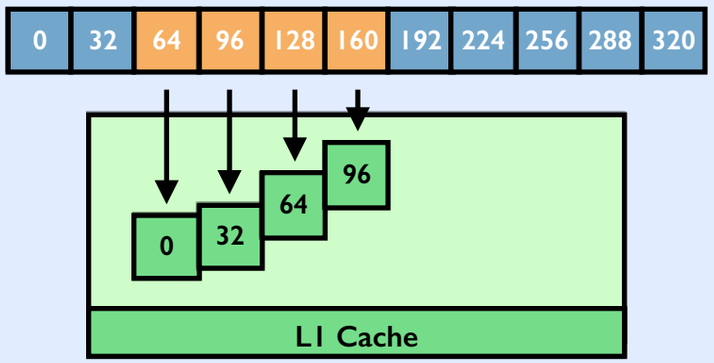
This can be important when the number of available compiler licenses is limited or when running at scale on a large system.

OCCA2: portability ?

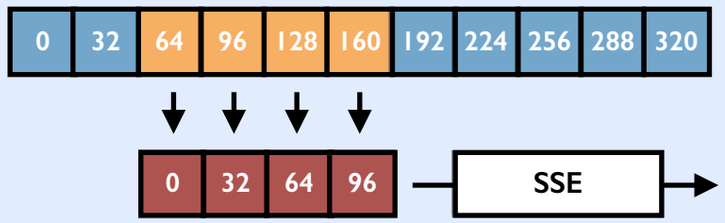
Optimization strategies for different platforms are converging to some extent.

CPU Optimizations

Cache

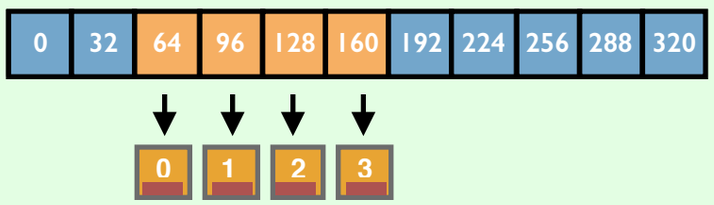


Vectorization

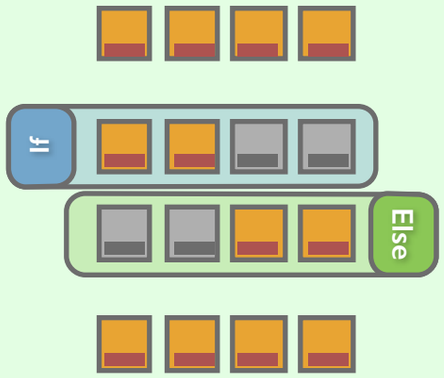


Thread Independence

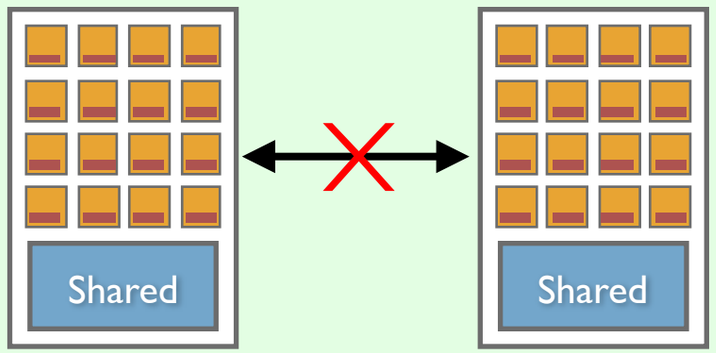
Coalescing



No Branching



Work-group Independence

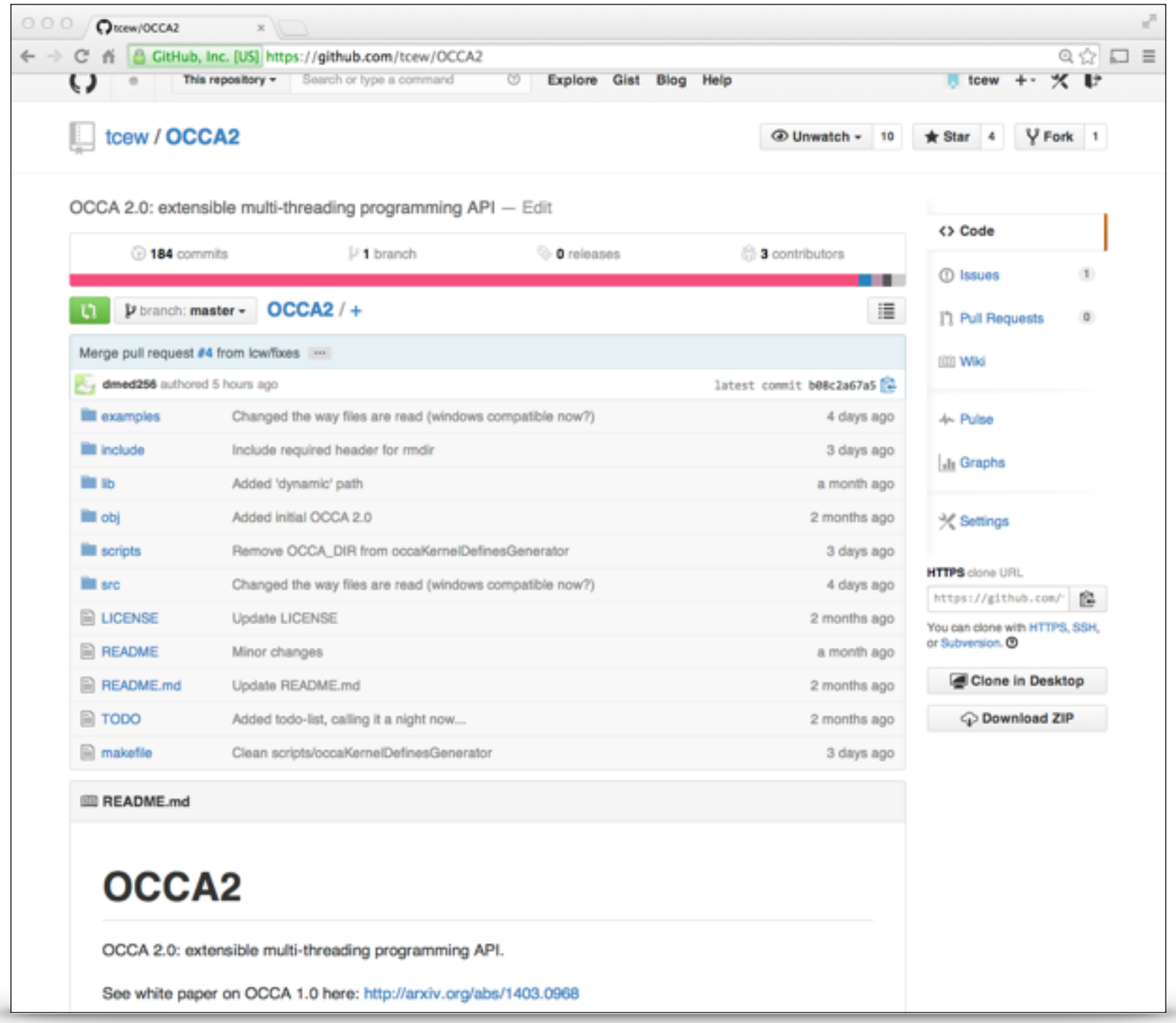


GPU Optimizations

Caching & coalescing are similar in spirit.

OCCA2 github

An experimental version of OCCA2 is available from github



https://github.com/tcew/OCCA2

OCCA2 whitepaper

A preprint is available for limited release:

OCCA: A UNIFIED APPROACH TO MULTI-THREADING LANGUAGES

DAVID S. MEDINA *, AMIK ST-CYR †, AND TIM WARBURTON ‡

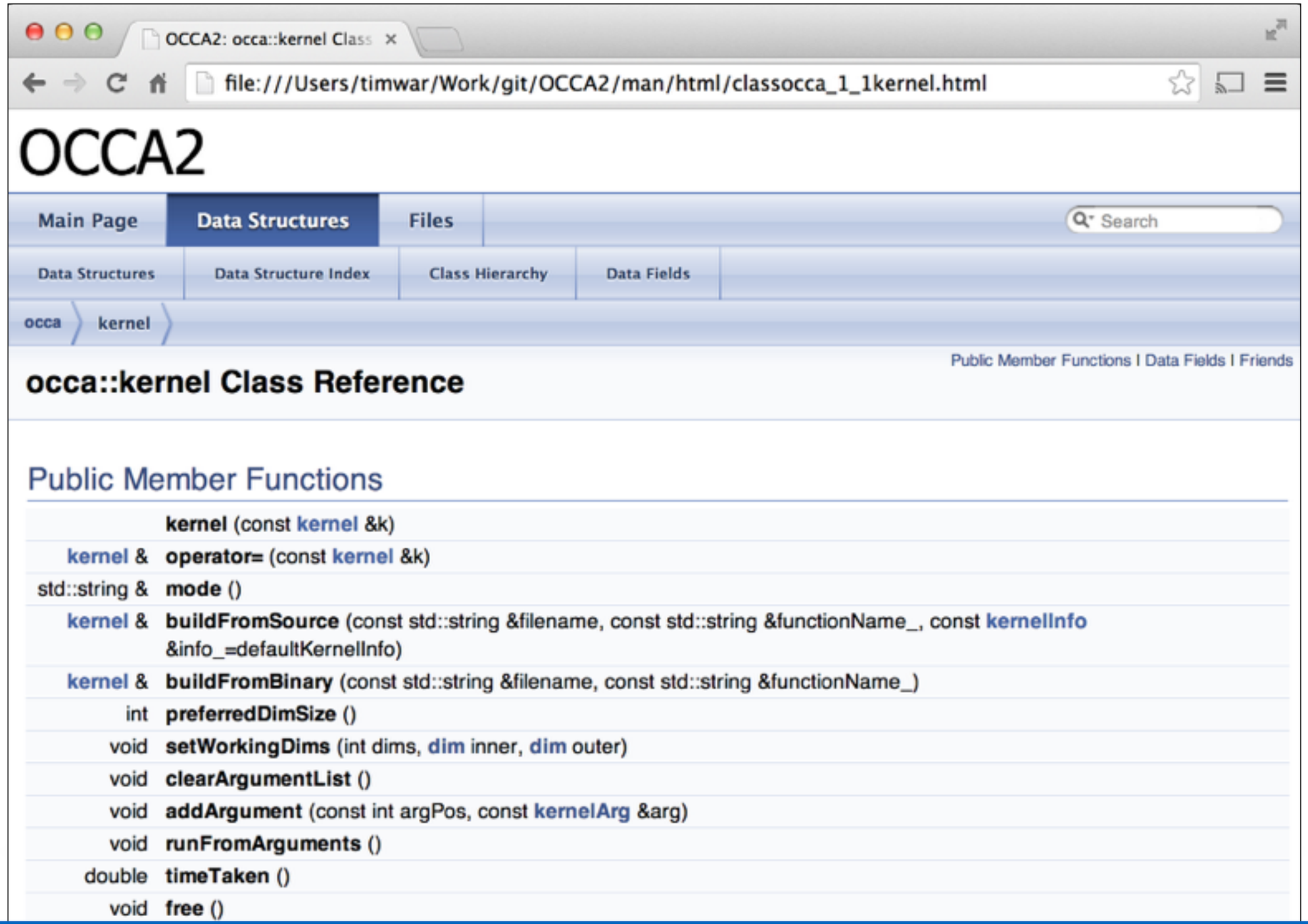
Abstract. The inability to predict lasting languages and architectures led us to develop OCCA, a C++ library focused on host-device interaction. Using run-time compilation and macro expansions, the result is a novel single kernel language that expands to multiple threading languages. Currently, OCCA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

1. Introduction. Until 2001, the top supercomputers were built on a parallel architecture involving multiple single-core vector machines. Multi-core processors then became the standard with co-processor acceleration as early as 2005 [1]. Not knowing which architectures will stand the test of time places programmers in an awkward position. For instance, IBM's cell architecture, known for as the graphics accelerator choice in the Playstation 3, was used in IBM's Roadrunner, the first petaflop-capable machine. Although the cell architecture had high promise, it was put on hiatus less than a decade after development [5]. Graphical processing units (GPUs) quickly dominated the co-processor competition with the only current com-

http://www.caam.rice.edu/~timwar/Hess14/OCCA2whitepaper.pdf

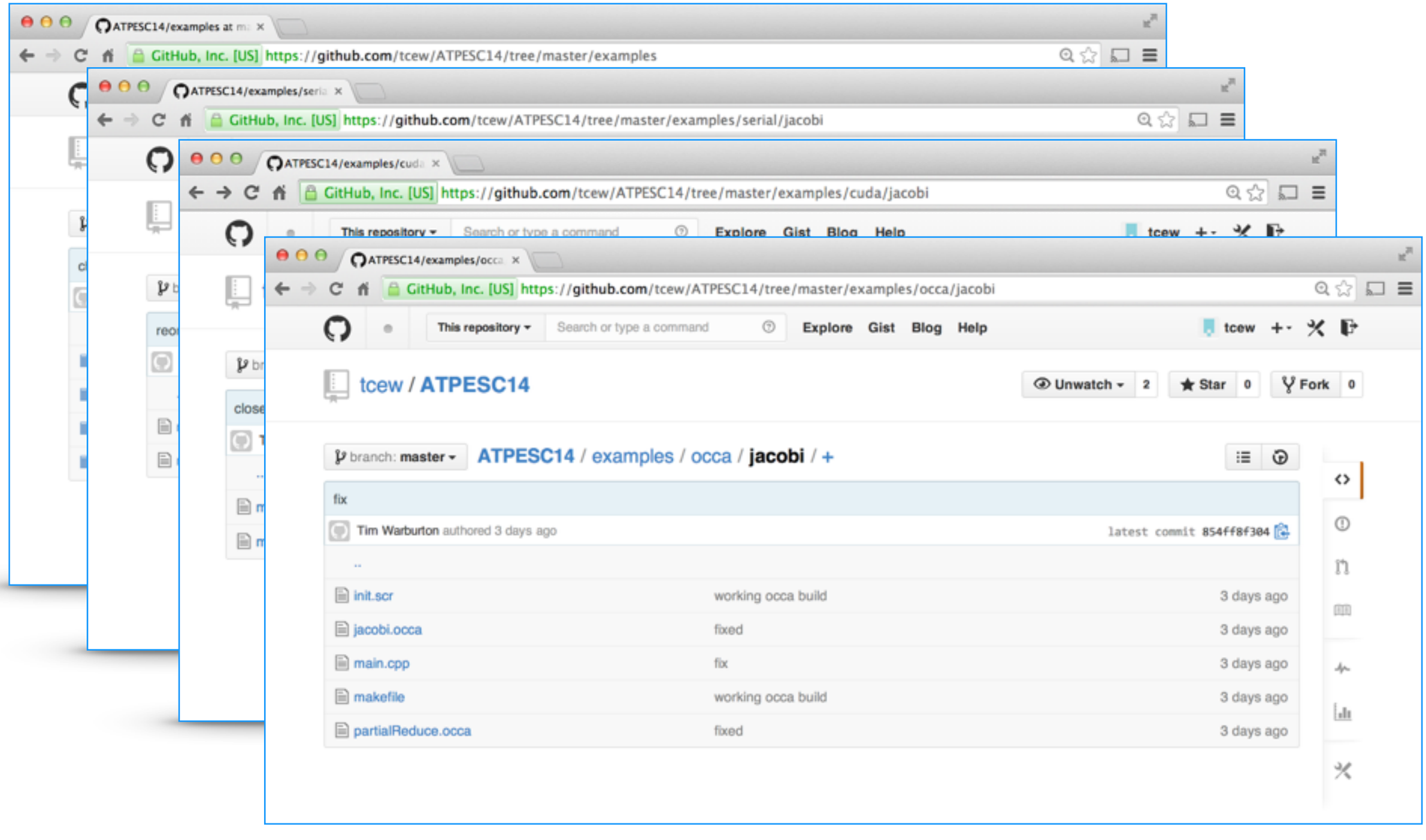
OCCA2:docs

Docs are a work in progress (ahem).



OCCA2: elliptic solver sample code

See the OCCA2Tutorial 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.*

OCCA2: 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 i
      // inner nodes
      const int id =

      newu[id] = 0.25

    }
  }
}
```

CUDA kernel:

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

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

OpenCL kernel:

```
__kernel void jacobi(const int N,
                     __global const datafloat
                     __global const datafloat
                     __global datafloat *newu)

  // Get thread indices
  const int i = get_global_id(0);
  const int j = get_global_id(1);

                                          ){

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

                                          rhs[id]
                                          + u[id - (N+2)]
                                          + u[id + (N+2)]
                                          + u[id - 1]
                                          + u[id + 1]);
```

OCCA 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…)

OCCA2: summary

OCCA2:

✅ Lightweight API that hedges against shifts in programming models.

✅ Front ends: C++, C, Julia, Python, MATLAB, F90,…

✅ Currently works with OpenCL, CUDA, OpenMP, pThreads & COI 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.

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

OCCA2: 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?)

OCCA2: github

Check out  out on  at https://github.com/tcew/OCCA2