www.libocca.org

OCCA: An Extensible Portability Layer for Many-core Programming

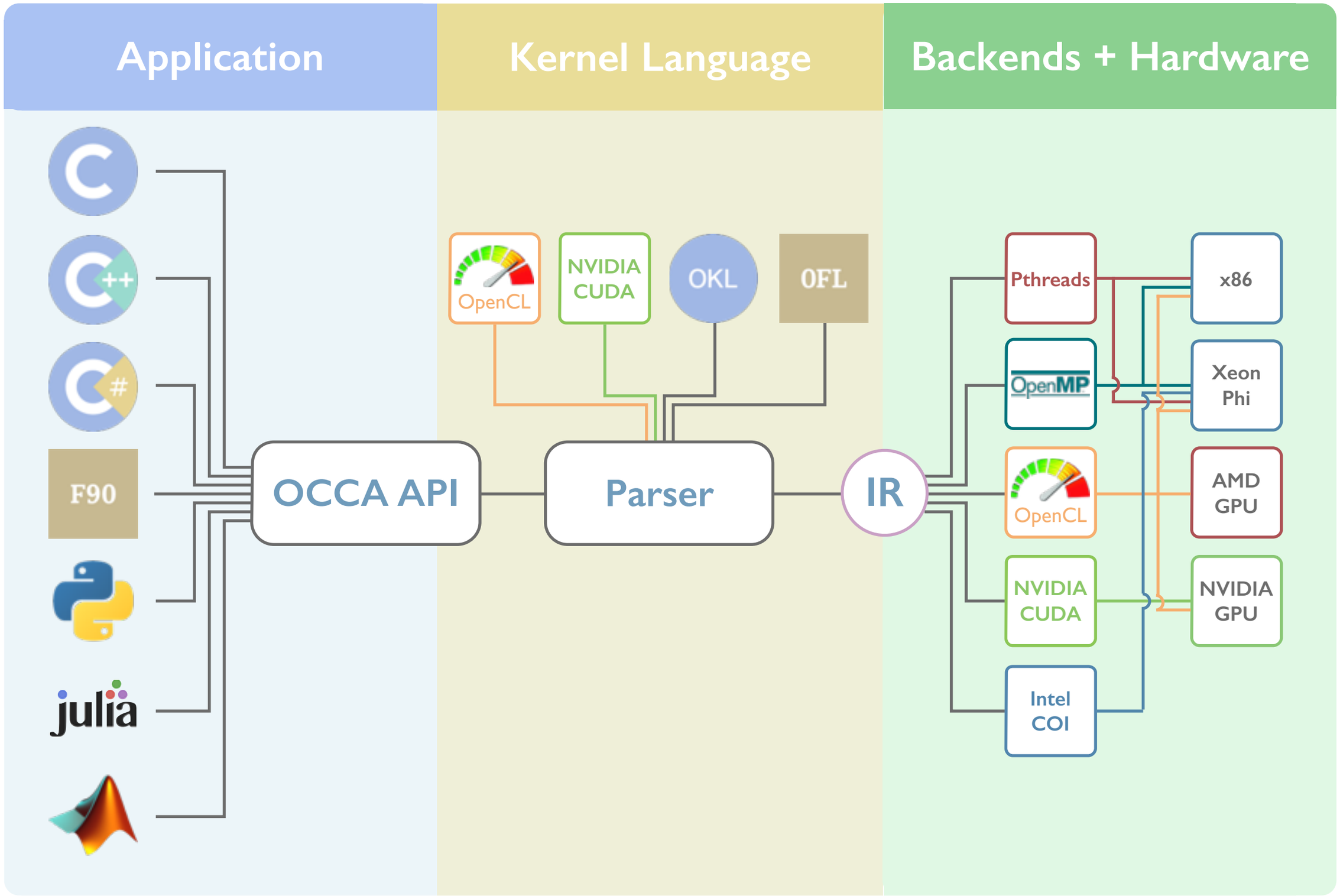
David S. Medina

Advisor: Prof. Tim Warburton

Computational and Applied Mathematics

Rice University

One-Slide Overview

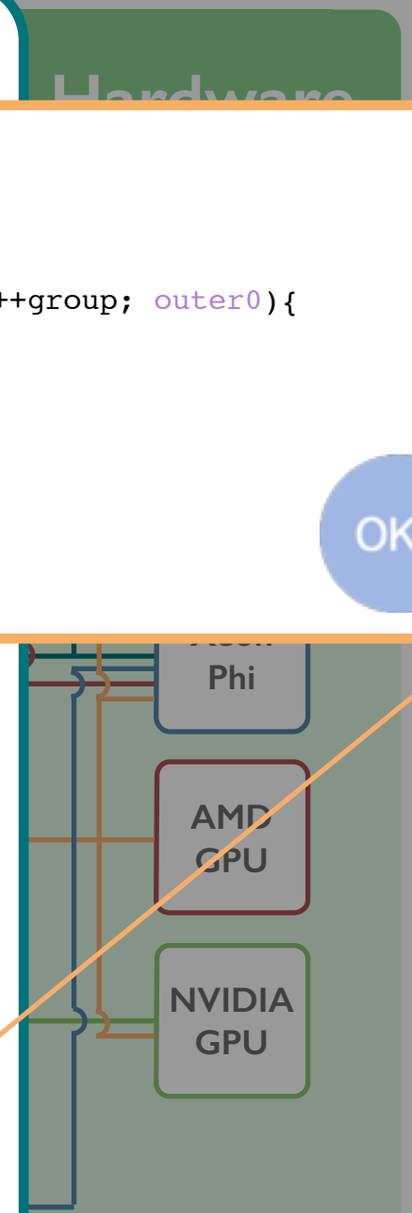


One-Slide Overview


```
#include "occa.hpp"

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

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

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

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

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

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

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

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

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

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

OKL



Portability Approaches

Directive Approach (e.g. OpenMP, OpenACC)

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

Source-to-Source Approach

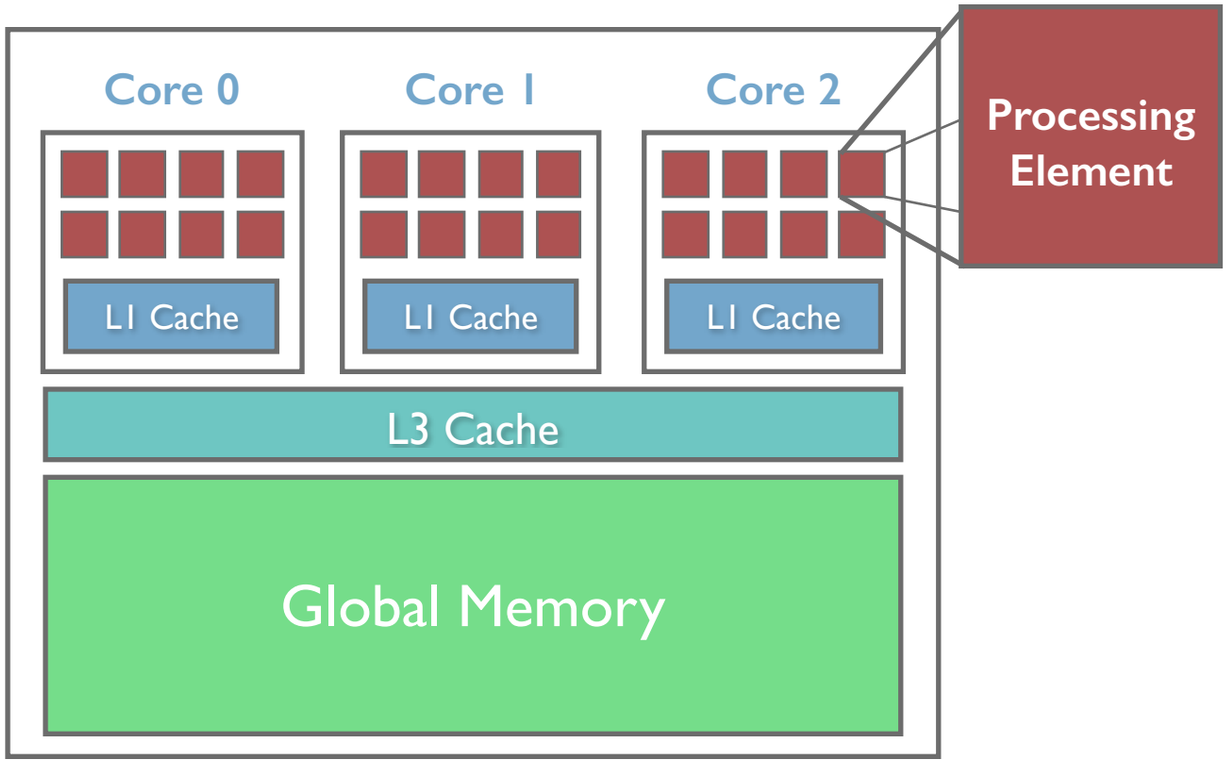
- Compiler tools can be used to translate across specifications/languages
- Performance is not always portable
- Maintenance of original and translated codes

Wrapper Approach (e.g. Kokkos, RAJA)

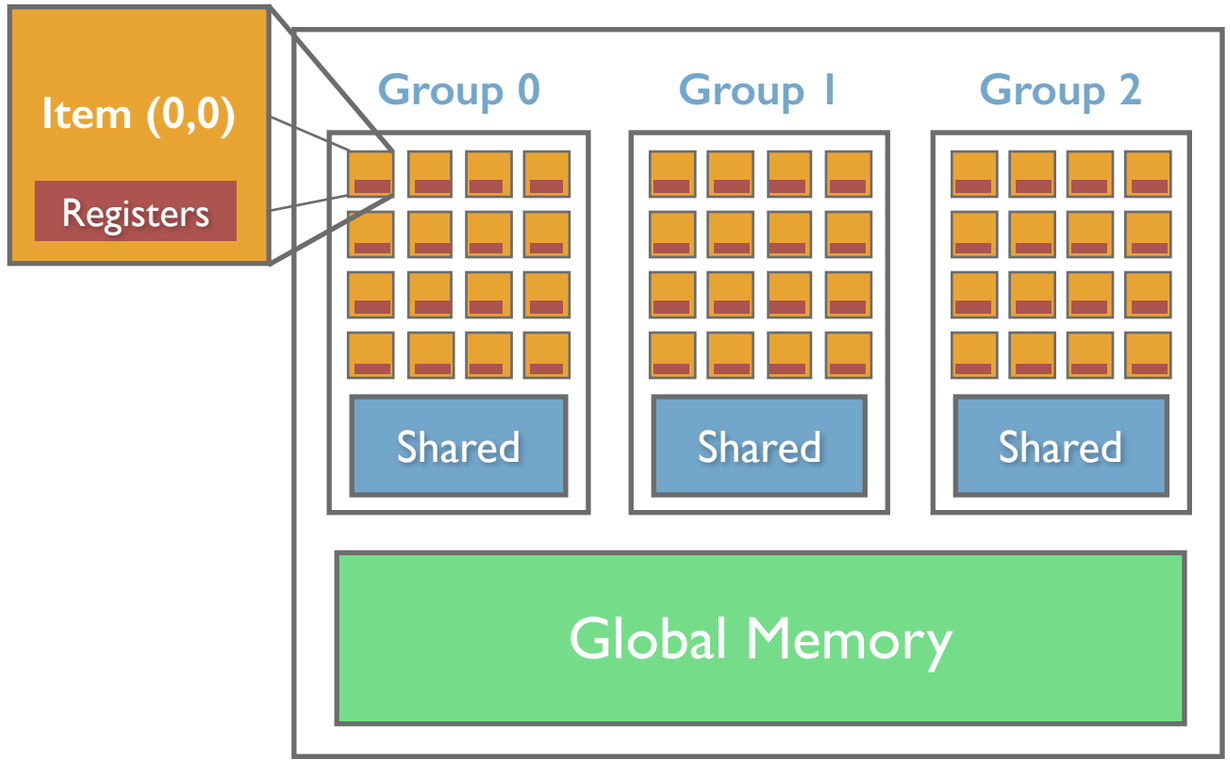
- Create a tailored library with optimized functions
- Flexibility comes from functors/lambdas at compile-time

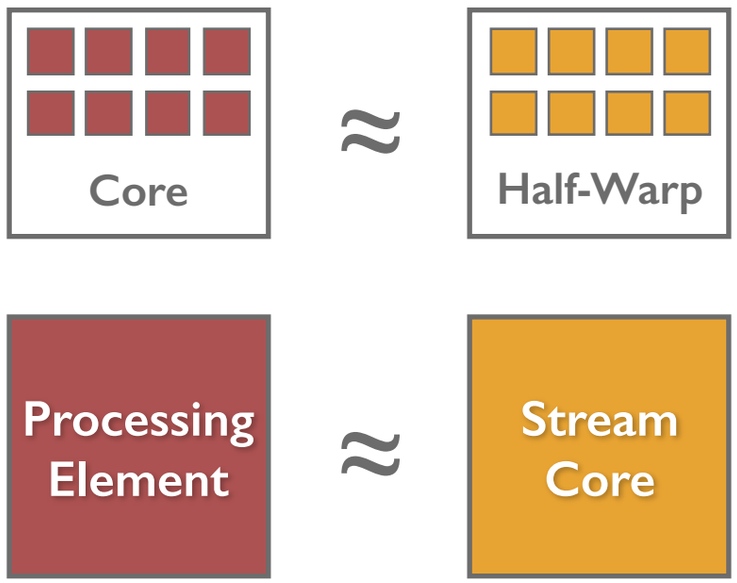
Parallelization Paradigm

CPU Architecture



GPU Architecture

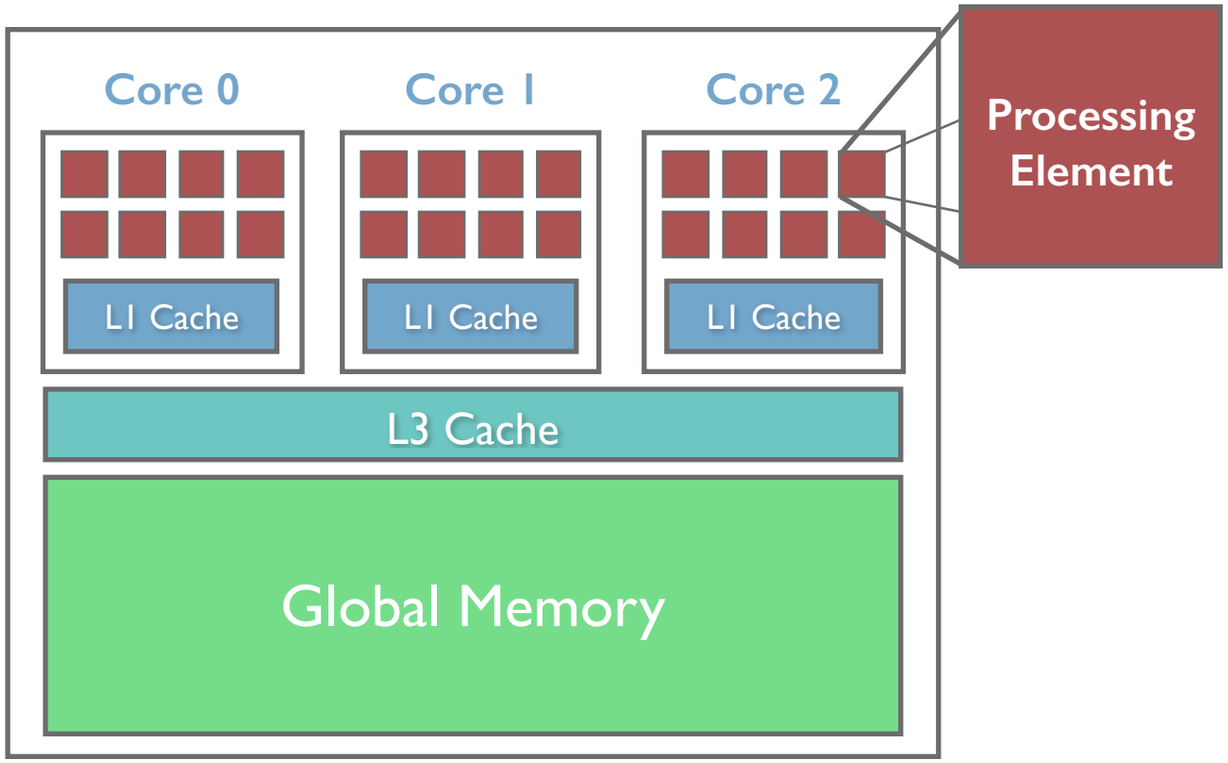




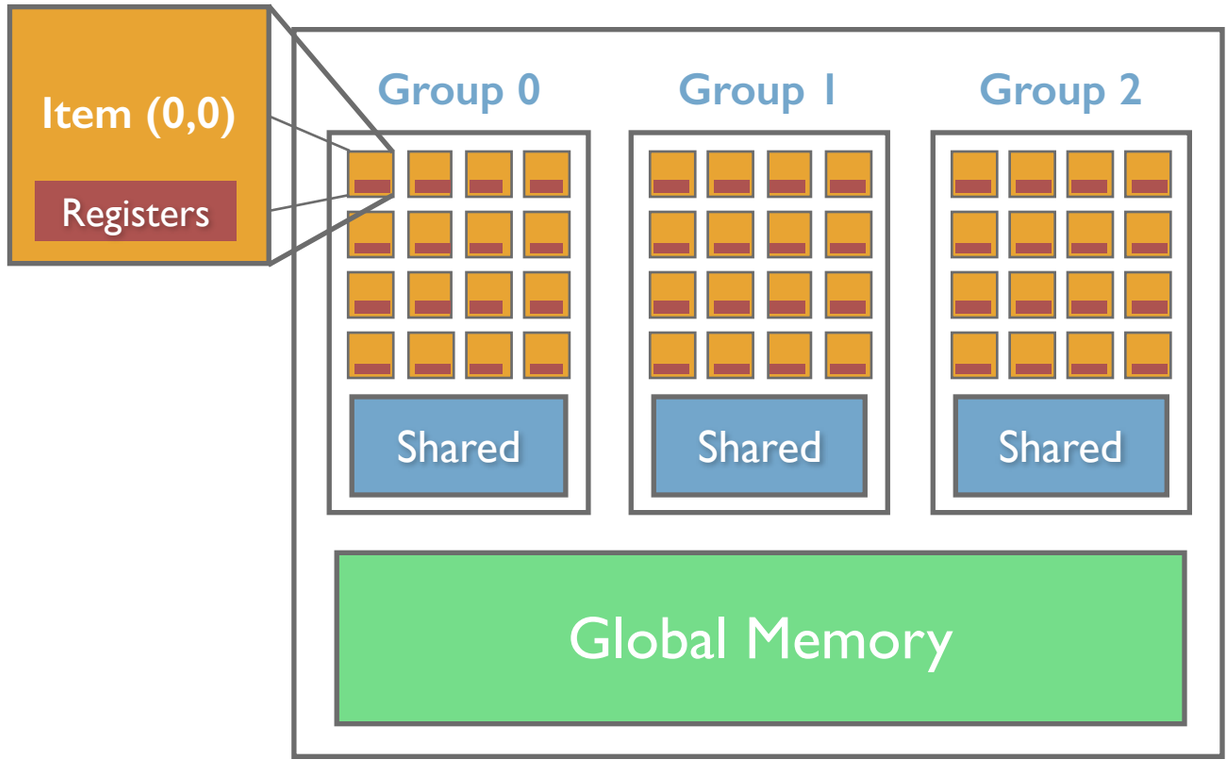
Computational hierarchies are similar

Parallelization Paradigm

CPU Architecture



GPU Architecture



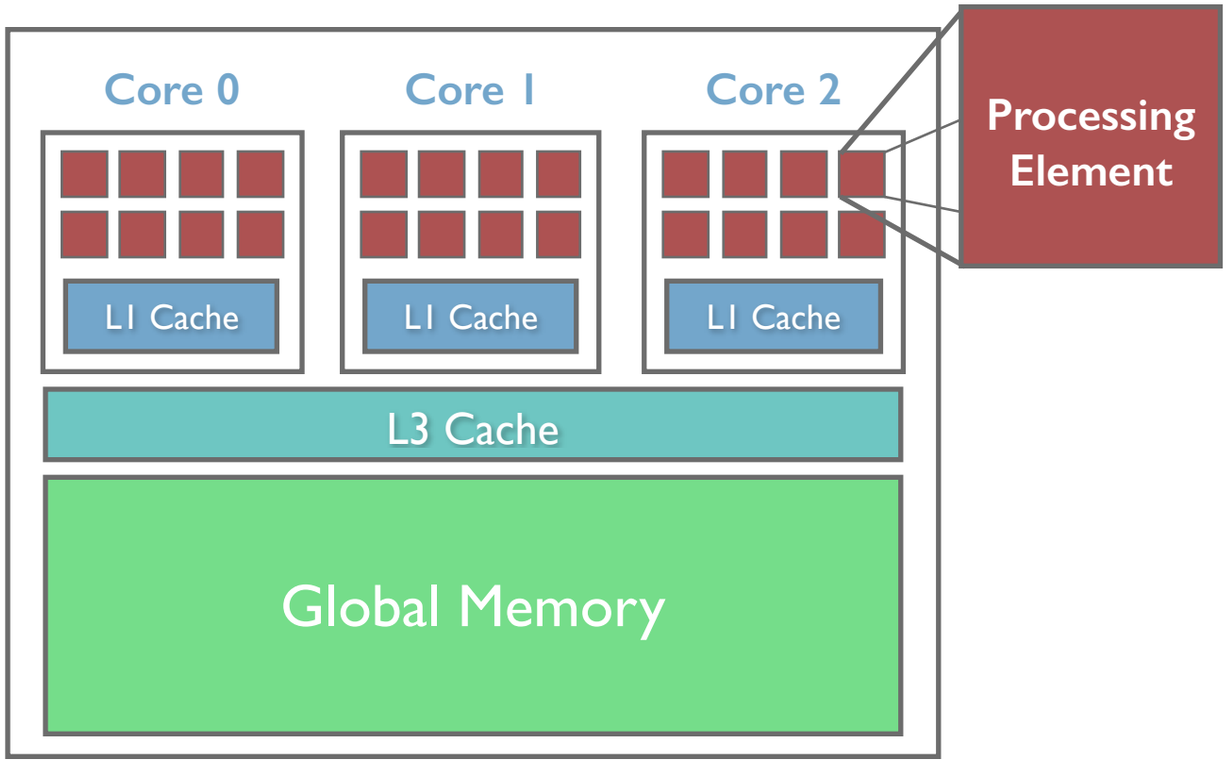

```
void cpuFunction(){
  #pragma omp parallel for
  for(int i = 0; i < work; ++i){

    Do [hopefully thread-independent] work

  }

}
```

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

  Do [group-independent] work

//   }
// }
}
```

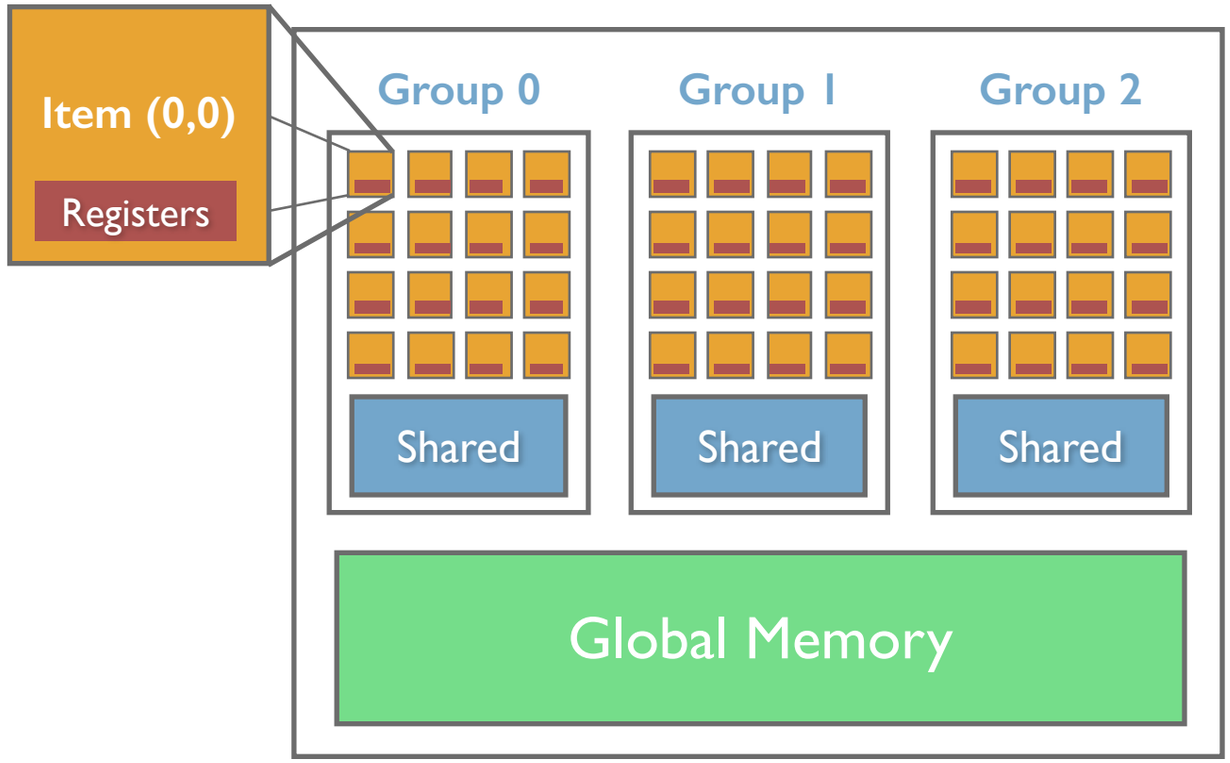
Computational hierarchies are similar

Parallelization Paradigm

CPU Architecture



GPU Architecture




```
void ompFunction(){
// for each thread {
  for(thread's work){

    Do [hopefully thread-independent] work

  }
// }
}
```

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

  Do [group-independent] work

//   }
// }
}
```

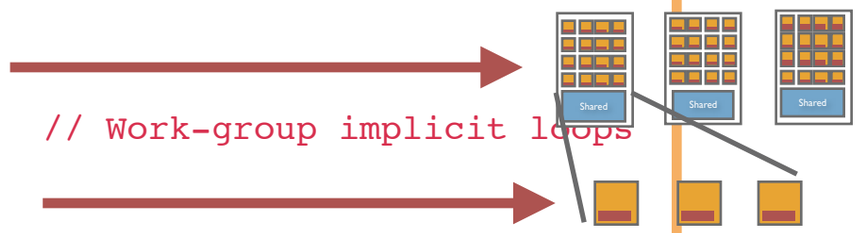
Computational hierarchies are similar

OKL: OCCA Kernel Language

Description

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

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

  for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
    for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
      for(int groupX = 0; groupX < xGroups; ++groupX; outer0){     // Work-group implicit loops

        for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
          for(int itemY = 0; itemY < yItems; ++itemY; inner1){
            for(int itemX = 0; itemX < xItems; ++itemX; inner0){  // Work-item implicit loops
            // GPU Kernel Scope
        }}}
  }}}

  ...
}
```



NVIDIA CUDA

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

OKL

The concept of iterating over groups and items is simple

OKL: OCCA Kernel Language

Multiple outer-loops

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

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

  for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
    for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
      for(int groupX = 0; groupX < xGroups; ++groupX; outer0){      // Work-group implicit loops

        for(int itemZ = 0; itemZ < zItems; ++ itemZ; inner2){
          for(int itemY = 0; itemY < yItems; ++ itemY; inner1){
            for(int itemX = 0; itemX < xItems; ++ itemX; inner0){   // Work-item implicit loops
            // GPU Kernel Scope
        }}}
  }}}

  ...
}
```

OKL

Data dependencies are found through a variable dependency graph

OKL: OCCA Kernel Language

Multiple outer-loops

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

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

  for(outer){
    for(inner){
    }
  }

  ...
}
```

OKL

Data dependencies are found through a variable dependency graph

OKL: OCCA Kernel Language

Multiple outer-loops

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

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

  for(outer){
    for(inner){
    }
  }

  for(outer){
    for(inner){
    }
  }
}
```

OKL

Outer-loops are synonymous with CUDA and OpenCL kernels

OKL: OCCA Kernel Language

Multiple outer-loops

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

```
kernel void kernelName(…){

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

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

}
```

OKL

Outer-loops are synonymous with CUDA and OpenCL kernels

OKL: OCCA Kernel Language

Shared Memory

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

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

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

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

OKL

Exclusive Memory

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

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

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

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

OKL

Local barriers are auto-inserted (gives a warning)

OKL: OCCA Kernel Language

Shared Memory

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

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

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

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

OKL

Exclusive Memory

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

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

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

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

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

?

OKL

Local barriers are auto-inserted (gives a warning)

OFL: OCCA Fortran Language

Description

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

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

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

  ...
end subroutine kernelName
```

OFL

Shared and Exclusive Memory

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

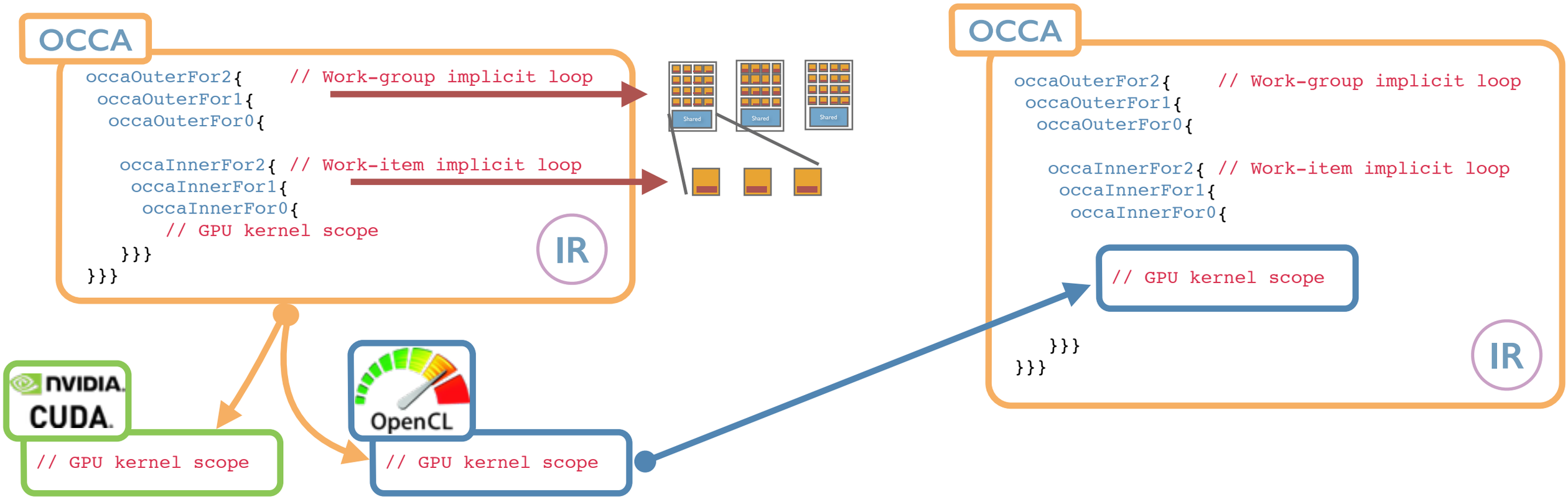
OFL

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

OpenCL/CUDA to OCCA IR

Description

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



Unsupported features include: textures, images, samplers

OCCA API: Adding Two Vectors

```
#include "occa.hpp"

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

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

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

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

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

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

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

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

OCCA API: Adding Two Vectors

```
#include "occa.hpp"

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

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

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

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

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

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

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

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

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

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

OKL

OFL: Adding Two Vectors

```
program main
  use occa
  implicit none

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

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

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

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

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

  device = occaGetDevice(deviceInfo)

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

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

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

  call occaCopyMemToPtr(ab(1), o_ab);

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

end program main
```

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

OFL: Adding Two Vectors

```
program main
  use occa
  implicit none

  character(len=1024)

  real(4), allocatabl

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

  allocate(a(1:5), b(

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

  device = occaGetDevi

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

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

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

  call occaCopyMemToPtr(ab(1), o_ab);

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

end program main
```

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

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

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

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

end subroutine addVectors
```

OFL

OCCA API: Original

```
#include "occa.hpp"

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

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

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

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

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

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

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

  addVectors(5, o_a, o_b, o_ab);

  o_ab.copyTo(ab);

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

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

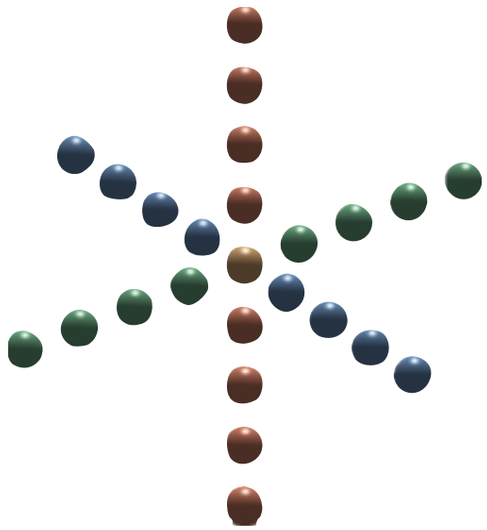
OCCA API: Original + UVA

```
#include "occa.hpp"

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

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

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

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

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

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

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

  addVectors(5, o_a, o_b, o_ab);

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

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

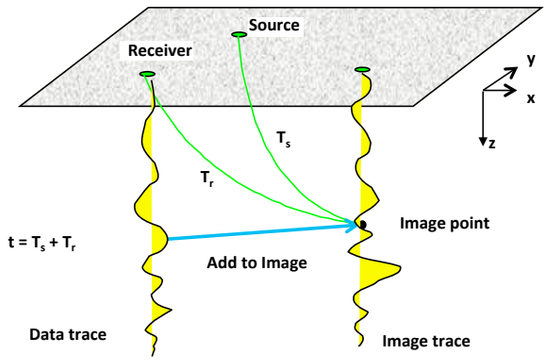
OCCA API: Original + UVA + Managed Memory

```
#include "occa.hpp"

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

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

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

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

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

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

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

  addVectors(5, a, b, ab);

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

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

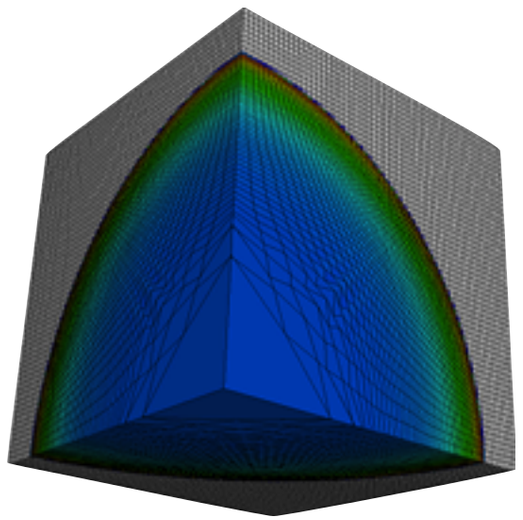
Applications using OCCA



Finite Difference

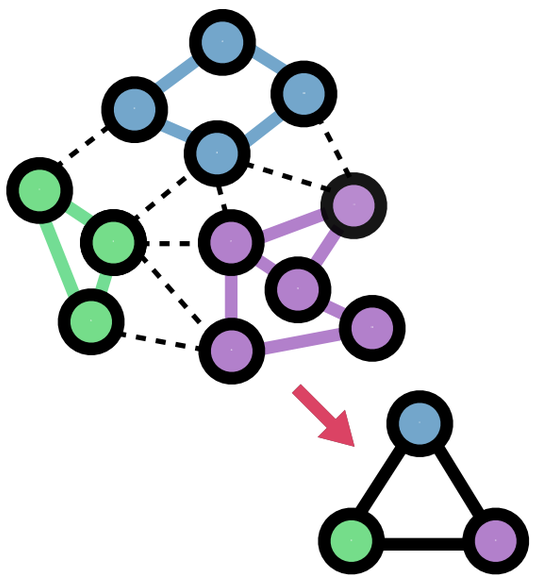


Kirchhoff migration

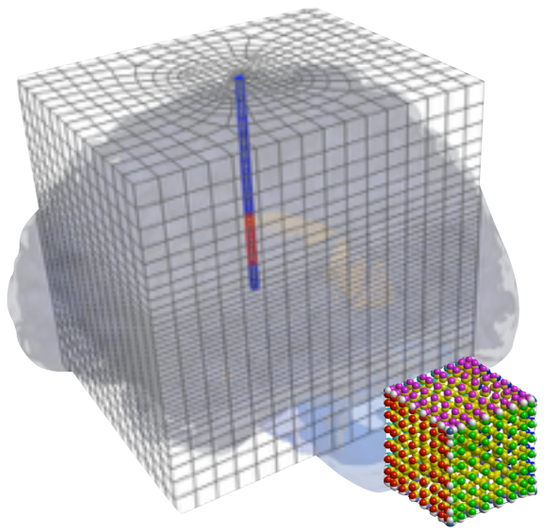


Unstructured Lagrangian

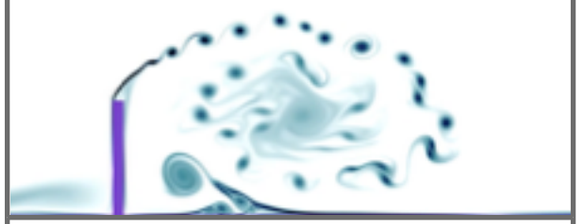
Monte Carlo (Neutronics)



Algebraic Multigrid



Spectral Element Method

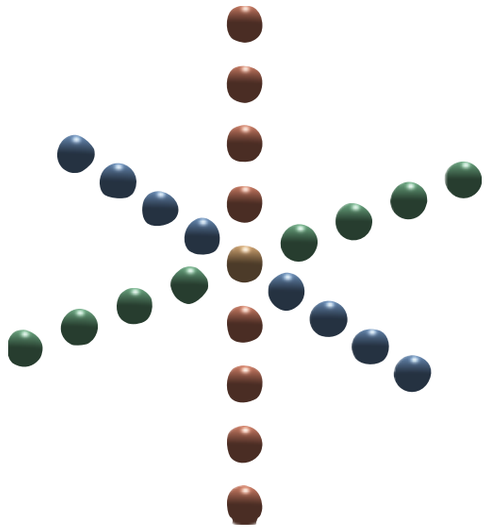


DG Compressible Flow

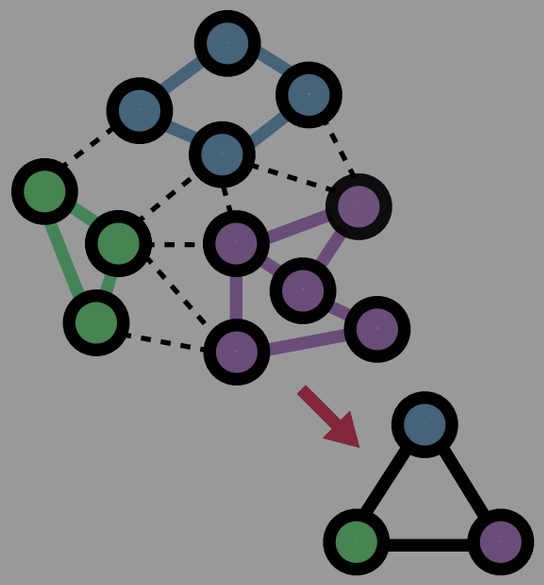
DG Shallow Water

+ GPU Benchmarks and others

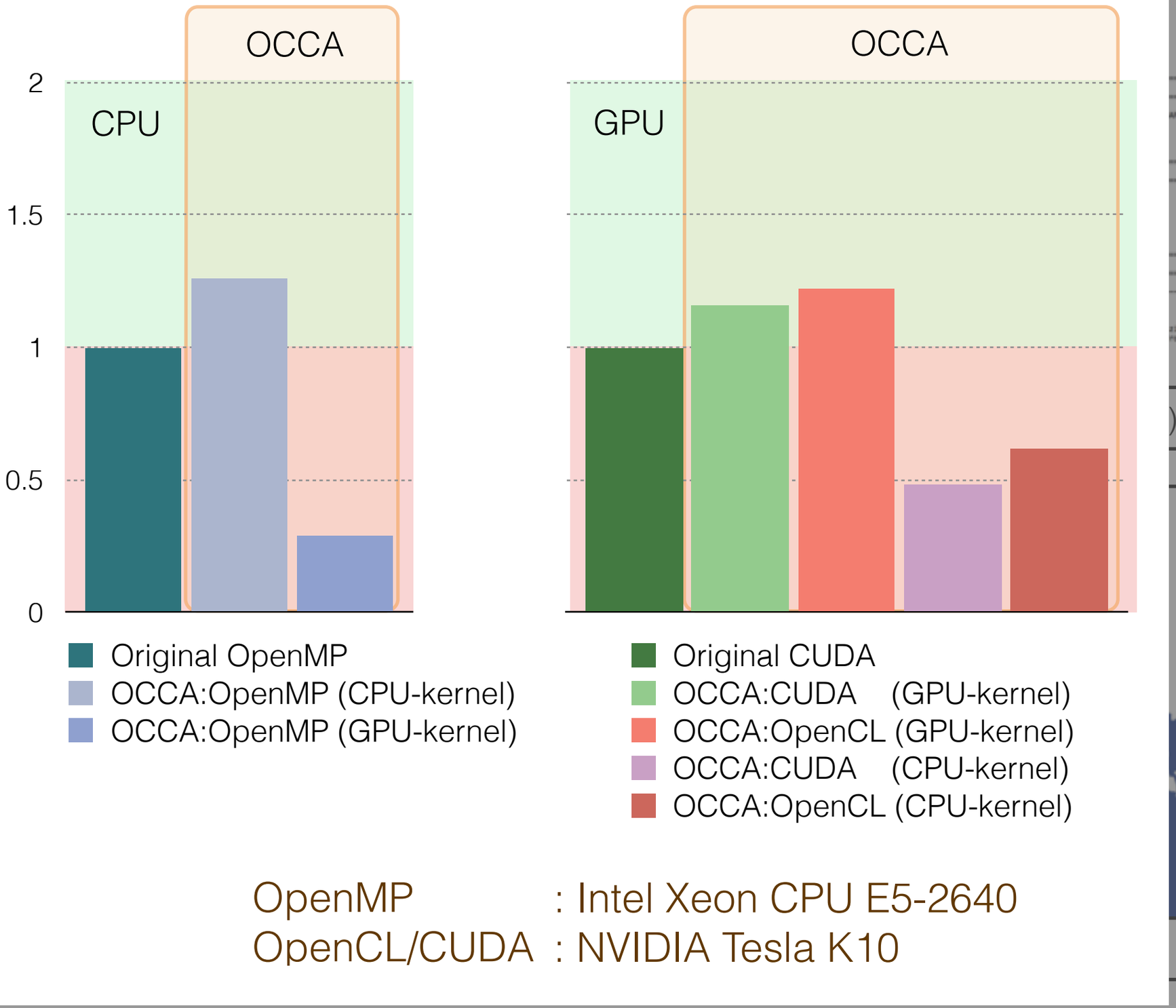
Applications using OCCA



Finite Difference



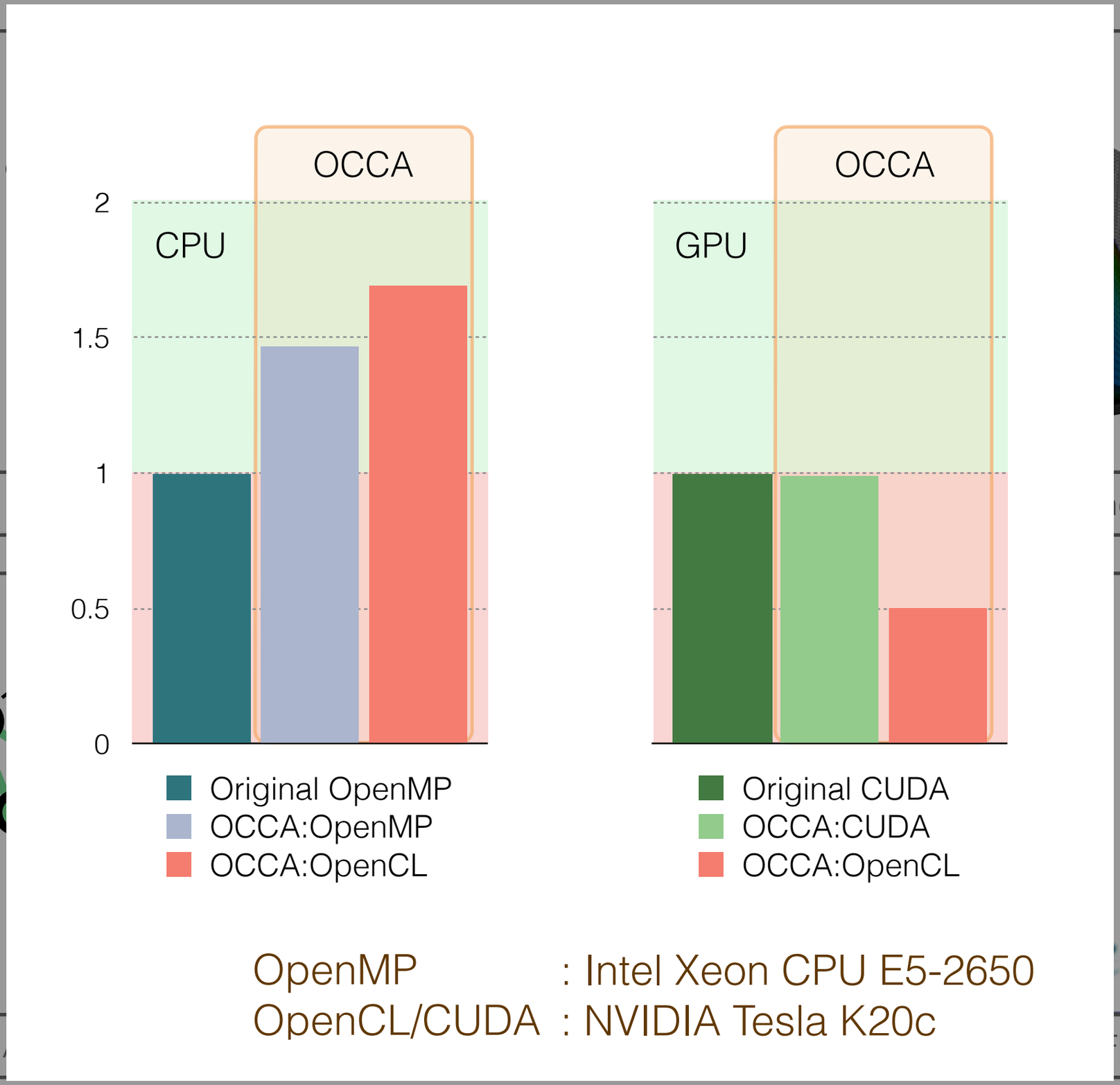
Algebraic Multigrid

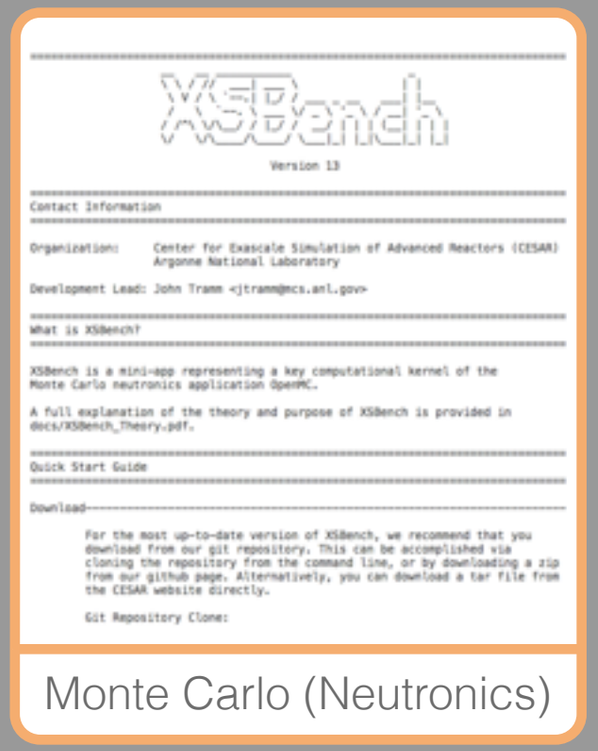


OpenMP : Intel Xeon CPU E5-2640
OpenCL/CUDA : NVIDIA Tesla K10

+ GPU Benchmarks and others

Applications using OCCA





Monte Carlo (Neutronics)

DG Shallow Water

+ GPU Benchmarks and others

Currently Working On

OCCA Features:

- Loop-carried dependency analysis for kernel generation + testing
- Support offsets in kernel calls

OKL/OFL Features

- Tiling loop labels (`tile(X)`, `tile(X,Y)`, `tile(X,Y,Z)`)
- Possibly loop-collapsing

Testing:

- Verify C and Fortran parsers (subset of ROSE benchmarks)
- Validate OKL performance on a subset of Rodinia tests:
  - 13 Dwarfs: Structured grid, Unstructured grid, Graph traversal, dense LA, …
  - 2 out of 13 tackled
- Embedded OKL-C and OFL-Fortran