OCCA: An Extensible Portability Layer for Many-Core Programming

43rd SPEEDUP Workshop on High-Performance Computing
University of Geneva 09/04/14

Tim Warburton & David S. Medina
Department of Computational & Applied Mathematics @ Rice University
All are GPU enabled and are building accelerated micro/mini-apps. Q: which many-core programming model(s) should they learn?
Part 1: Medical procedure being modeled:
- Magnetic resonance guided laser interstitial thermal therapy: MRgLITT.
- Diego Rossinelli’s point is well taken: a real customer is very important!

Part 2: Pennes bioheat model:
- PDE modeling of tissue heating.

Part 3: Computational approach:
- Portable threading language: OCCA.
- Focus: on-node performance.

Part 4: The brainNek simulator
- Thread parallel implementation.
- Performance.
Magnetic Resonance Guided Laser Interstitial Thermal Therapy (MRgLITT)

c. 1966 observation on proton thermal emission properties.
c. 2006 experimental thermal therapy
### Magnetic Resonance Imaging

**Magnetic resonance imaging:**
- Strong magnets align patient’s proton spin axes.
- RF coils induce time varying EM emissions.
- MRI produces proton density estimation.

**Magnetic resonance thermometry:**
- Local temperature changes shift EM emission frequencies.
- Frequency shift is predictably linear for aqueous tissue.

### Key observations:
- 1-1 correspondence between signal frequency & location.
- 1-1 correspondence shifted by local heating.
MR-gated Laser Interstitial Thermal Therapies

**MRgLITT:**

- Surgeon drills a cranial hole & inserts an actively cooled diffusing interstitial laser fiber.
- The laser energy heats tissue.
- Treatment for brain, prostate, & brain lesions in brain, prostate, liver, & bone and for chronic epilepsy.
- Minimally invasive alternative to conventional surgery.

**Magnetic resonance imaging:**

- Plan, target, real-time monitoring, post treatment verification.

**Details:**

- Thermal dose (10-15W, 30-180s) used to verify delivery.
- <3cm diameter, can pull back fiber and heat multiple volumes.
Pennes Bioheat Model

c. 1948 biomedical model.
c. 1984 SEM numerical method.
Pennes Bioheat Equation

The Pennes bioheat equations model brain tissue heating.

\[ \rho c_p \frac{\partial T}{\partial t} - \nabla \cdot (k(T, x, \beta) \nabla T) + \omega(T, x, \beta) c_{\text{blood}} (T - T_a) = S(\beta, x, t) \text{ in } \Omega \]

\[ -k(T, x, \beta) \nabla T \cdot n = h(T - T_\infty) \quad \text{on } \partial \Omega_C \]

\[ -k(T, x, \beta) \nabla T \cdot n = G \quad \text{on } \partial \Omega_N \]

\[ T(x, 0) = u^0 \quad \text{in } \Omega \]

Pennes Bioheat Equation

The Pennes bioheat equations model brain tissue heating.

\[ \rho c_p \frac{\partial T}{\partial t} - \nabla \cdot (k(T,x,\beta)\nabla T) + \omega(T,x,\beta)c_{blood}(T - T_a) = S(\beta,x,t) \text{ in } \Omega \]

We use spectral elements for extensive preconditioning options.
The existing implementation used off the shelf packages to build a Pennes bioheat simulation tool for MRgLITT …

… leveraging $billions in software R&D and many, many years of collective software development.
Vision: FEM modeling in an online feedback for active control of MR-gLITT procedures.

Obstacles: need 100% reliability for live prediction capabilities.

Diagram courtesy David Fuentes

... but a typical simulation took ~12 hours on 12 CPU cores and scaling was an issue..
The medical practitioners cannot ship live MRTI data to a data center...
Vision: FEM modeling in an online feedback for active control of MR-gLITT procedures.

Obstacles: need 100% reliability for live prediction capabilities.

... but a typical simulation took ~12 hours on 12 CPU cores and scaling was an issue..

The medical practitioners cannot ship live MRTI data to a data center...
Goal: simulator requirements

Computational Constraints:

- Time is of the essence: simulate in seconds not hours.
- Run on local hardware: avoid transferring sensitive medical data.
- Portable: must run on any available CPU or GPU with whatever API is available.
- Adjust to evolving processor design and programming models.

Testing:

- Verify against synthetic solutions.
- Verify against synthetic agar phantom data.
- Verify against recorded patient therapy data.
- Predictive capability?
OCCA2: portable multi-threading API

Switching gears…

Draft implementation: (https://github.com/tcew/OCCA2)
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?
Can we insulate against uncertainty and fragmentation?
Can we avoid maintaining multiple code bases?

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 Smorgasbord

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

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

... and a range of thread programming models (with vendor favorite pairs).
Many-core Smorgasbord

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

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

<table>
<thead>
<tr>
<th>API</th>
<th>Type</th>
<th>Front-ends</th>
<th>Kernel</th>
<th>Back-ends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kokkos</td>
<td>ND arrays</td>
<td>C++</td>
<td>Custom</td>
<td>CUDA &amp; OpenMP</td>
</tr>
<tr>
<td>VexCL</td>
<td>Vector class</td>
<td>C++</td>
<td>-</td>
<td>CUDA &amp; OpenCL</td>
</tr>
<tr>
<td>RAJA</td>
<td>Library</td>
<td>C++</td>
<td>C++ Lambdas</td>
<td>CUDA, OpenMP, OpenACC</td>
</tr>
<tr>
<td>OCCA2</td>
<td>API &amp; Macro Kernel Language</td>
<td>C,C,C#, (F90), Python, MATLAB, Julia</td>
<td>Custom Unified Kernel Language</td>
<td>CUDA, OpenCL, pThreads, COI &amp; OpenMP</td>
</tr>
<tr>
<td>CU2CL</td>
<td>Source-to-source</td>
<td>App</td>
<td>CUDA</td>
<td>OpenCL</td>
</tr>
<tr>
<td>OmpSs</td>
<td>Source-to-source compiler</td>
<td>C</td>
<td>OpenMP, Cilk, MPI, OpenCL</td>
<td>OpenCL, MPI, Insieme IR runtime</td>
</tr>
<tr>
<td>Ocelot</td>
<td>PTX Translator</td>
<td>CUDA</td>
<td>CUDA</td>
<td>OpenCL</td>
</tr>
</tbody>
</table>

OCCA emphasis: lightweight and extensible.
OCCA2: unified threading model

Portability & extensibility: device independent kernel language and native host APIs.
OCCA2: code & API stack

OCCA2 communicates with several different run-times.

Windows, Linux, & OS X support to varying degrees.

* goodly subset of OpenCL and CUDA kernel languages.
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 wrapper) 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 detects changes in compiler options, source code ...

* where appropriate.
#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] << '
';
OCCA2: example adding two vectors

```cpp
#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] << std::endl;
    }
}
```

OCCA2 kernel language syntax & macros: http://www.github.com/tcew/OCCA2
Example HOST & DEVICE code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
All the HOST codes use the same kernel.
Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
Native Host Codes

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 "stdlib.h"

int main(int argc, char **argv)
{
    entries = 5;
    device = occa.device("OpenCL", 0, 0);

    for(i = 0; i < 5; ++i)
        printf("%d = %f
", 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

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

```python
import ctypes
import occa

# Dynamic range?
entries = 5
device = occa.device('OpenCL', 0, 0);
a = [i for i in xrange(entries)]
b = [1 - i for i in xrange(entries)]
ab = [0 for i in xrange(entries)]
o_a = occa.malloc('single', a)
o_b = occa.malloc('single', b)
o_ab = occa.malloc('single', ab)
addVectors = occa.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);
ab = o_ab(:);
```

All the HOST codes use the same kernel.

Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
OCCA2: C++ API behind the scenes

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

- `occa::kernel occa::device::buildKernel()`
- `occa::kernel::operator() to run kernel`
- `clEnqueueNDRangeKernel`
- `cuLaunchKernel`
- `call dynamically linked fn`

**Notes**

OCCA uses the CUDA Driver API for run-time compilation of OCCA kernels as CUDA kernels.
To do: improve OpenMP loop partitioning.

**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.

```cpp
_for (occaOuterId2 = 0; occaOuterId2 < occaOuterDim2; ++occaOuterId2){
  for(occaOuterId1 = 0; occaOuterId1 < occaOuterDim1; ++occaOuterId1){
    for(occaOuterId0 = 0; occaOuterId0 < occaOuterDim0; ++occaOuterId0){
      occaInnerFor2{ // Work-item implicit loop
        occaInnerFor1{
          occaInnerFor0{
            // GPU kernel scope
          }
        }
      }
    }
  }
}
```
**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.

```
#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
          }}
        }}
      }}
```
To do: improve OpenMP loop partitioning.

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.
OCCA2: overview of the kernel model

Goal: one implementation for each kernel.

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

Confusing CUDA kernel
[ hidden parallel for loops ]

Confusing OpenCL kernel
[ hidden parallel for loops ]

__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;
}
OpenCL and CUDA kernels use an implicit parallel loop structure.

OCCA exposes these loops.

OCCA kernels compile at runtime to serial, OpenCL, CUDA, OpenMP, or Pthreads.

Minimum requirement:
1 outer loop & 1 inner loop.

Loop bounds are specified

The map between thread array <> parallel loop is made explicit.
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...).
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…).
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...).
An experimental version of OCCA2 is available from github

Code: https://github.com/tcew/OCCA2
Tutorial: https://github.com/tcew/OCCA2Tutorial
OCCA: github & road map

OCCA road map [6-month refresh period]:

- **OCCA 1.0:**
  - C++ front-end.
  - OpenMP, CUDA, OpenCL back-ends.

- **OCCA 2.0:**
  - Added native C#/C/F90/Python/MATLAB front-ends.
  - Added COI and pThreads back-ends.
  - Improved software engineering best practices.

- **OCCA 3.0 in progress:**
  - Added kernel pre-processor.
  - Translates OpenCL & CUDA kernels to OCCA intermediate representation.
  - New simplified OCCA kernel language support.
  - Next: kernel inlining in source code…

Code: [https://github.com/tcew/OCCA2](https://github.com/tcew/OCCA2)
Tutorial: [https://github.com/tcew/OCCA2Tutorial](https://github.com/tcew/OCCA2Tutorial)
brainNek: accelerated spectral elements for bioheat modeling

c. 1948 biomedical model.
c. 1966 proton thermal emission properties.
c. 1984 spectral element method.
c. 2006 experimental thermal therapy

c. 2013 many-core parallel computing.
Sampling of GPU-FEM Literature

**GPU-FEM**: mostly low order and focus on storing stiffness matrices

- Göddeke, Strzodka, & Turek, (2005)
  “Accelerating double precision FEM simulations with GPUs”.
  “Efficient nonlinear FEM for soft tissue modeling and its GPU implementation …”
- Cecka, Lew, & Darve, (2011)
  “Assembly of finite element methods on graphics processors”.
  “Finite-element sparse matrix vector multiplication on graphic processing units.”.
- Ribbrock et al (2011)
  “Efficient finite element geometric multigrid solvers for unstructured grids on GPUs”.
- Zhisong Fu et al. (2014)
  “Architecting the Finite Element Method Pipeline for the GPU”.

**GPU-SEM**: matrix-free but typically for explicit time stepping wave equations

- Komatitsch, Michéa, & Erlebacher, (2009)
  “Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA”.
  “Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs”.
- Michéa, & Komatitsch, (2010)
  “Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards”.

We focus on matrix-free PCG for spectral element discretization of elliptic problems. OAS or AGMG preconditioning.
High Order Finite Elements: specifics

Spectral elements* following Paul Fischer’s Nek5k roadmap.

Matrix-free preconditioned conjugate gradient for elliptic solver:

- $O(N)$ operations per node per iteration.
- $(N+1)^2$ threads evaluate the action of the stiffness matrix on each element.
- Fast spectral overlapping additive Schwarz preconditioner.
  - $(N+3)^2$ threads evaluate the action of the preconditioner on each element.
- Aggregation based (FEM) algebraic multigrid preconditioning

* See for instance Patera ’84 for SEM background
Dr Fuentes’ implementation used existing packages to build a Pennes bioheat simulation tool for MRgLITT.

Fuentes’ implementation leveraged $billions in software R&D and many, many years of collective software development.
brainNek: code structure

brainNek is optimized for many-core computing

Intensive computation is done by optimized “OCCA” kernel code executed as OpenCL, CUDA, or OpenMP.

The thin interface layers are “no touch” pass-through layers.
We tune for the best # of slabs for each 2D sweep.

Variants of Micikevicius’s register rolling approach is limited by register pressure.
We built an OCCA based partitioned preconditioned conjugate gradient solver.

1. \[ a_L = A_L Z p_G \]
2. \[ b_G = Z^t a_L \]
   \[ \gamma_k = p_G \cdot b_G \]
   \[ \alpha_k = \delta_k / \gamma_k \]
3. \[ x_G = x_G + \alpha_k p_G \]
   \[ r_G = r_G - \alpha_k b_G \]
   compute \( \| r_G \|^2 \)
   if \((\| r_G \| \leq \tau)\) break;
4. \[ z_P = M^{-1} \hat{Z} r_G \]
5. \[ z_G = \hat{Z}^t z_P \]
   \[ \delta_{k+1} = z_G \cdot r_G \]
   \[ \beta_k = \delta_{k+1} / \delta_k \]
6. \[ p_G = z_G + \beta_k p_G \]
brainNek: kernel partition of PCG

We built an OCCA based partitioned preconditioned conjugate gradient solver

1. \( a_L = A_L Z p_G \)  
   Matrix-free operator

2. \( b_G = Z' a_L \)  
   \( \gamma_k = p_G \cdot b_G \)  
   \( \alpha_k = \delta_k / \gamma_k \)  
   Synchronization point

3. \( x_G = x_G + \alpha_k p_G \)  
   \( r_G = r_G - \alpha_k b_G \)  
   compute \( ||r_G||^2 \)
   if \( (||r_G|| \leq \tau) \) break;  
   Synchronization point

4. \( z_P = M^{-1} \hat{Z} r_G \)  
   Matrix-free precon.

5. \( z_G = \hat{Z}' z_P \)  
   \( \delta_{k+1} = z_G \cdot r_G \)  
   \( \beta_k = \delta_{k+1} / \delta_k \)  
   Synchronization point

6. \( p_G = z_G + \beta_k p_G \)

Six PCG kernels structured to reduce redundant data movement and avoid race conditions. 37% of the time is spent in gathers, reductions and level-one vector operations.
Six PCG kernels structured to reduce redundant data movement and avoid race conditions.

37% of the time is spent in gathers, reductions and level-one vector operations.

brainNek: kernel partition of PCG

We built an OCCA based partitioned preconditioned conjugate gradient solver

1. \( a_L = A_L Z p_G \)
2. \( b_G = Z' a_L \)
   \( \gamma_k = p_G \cdot b_G \)
   \( \alpha_k = \delta_k / \gamma_k \)
3. \( x_G = x_G + \alpha_k p_G \)
   \( r_G = r_G - \alpha_k b_G \)
   compute \( ||r_G||^2 \)
4. if \( (||r_G|| \leq \tau) \) break;
5. \( z_P = M^{-1} \hat{Z} r_G \)
6. \( z_G = \hat{Z}' z_P \)
7. \( \delta_{k+1} = z_G \cdot r_G \)
   \( \beta_k = \delta_{k+1} / \delta_k \)
8. \( p_G = z_G + \beta_k p_G \)
Two undergraduates developed the GUI interface, interfaced with segmented brain materials, added the laser forcing, Robin boundary conditions...

The GUI enables a surgeon to place, orient and slide the laser applicator as well as change the heating duration and laser power interactively.

The simulator is implemented in a combination of C++/OCCA/OpenGL.
The simulator is implemented in a combination of C++/OCCA/OpenGL.

brainNek Mini-app

Two undergraduates developed the GUI interface, interfaced with segmented brain materials, added the laser forcing, Robin boundary conditions...

The GUI enables a surgeon to place, orient and slide the laser applicator as well as change the heating duration and laser power interactively.
brainNek: benchmarking

Two platforms: AMD 7990 & NVIDIA Titan

Two high end “gamer” GPUs.

2x2048 FPU
2x288 GB/s
2x4.1 TFLOP

2688 FPU
288 GB/s
4.5 TFLOP

Image: www.maximumpc.com
Two competing issues impact performance: 1. occupancy  2. flops per load/store

\[ a_L = A_L Z_p G \]

\[ z_p = M^{-1} \hat{Z} r_G \]

We would have to change the kernel partitioning to get more than a 50% speed up for \( N>4 \).

PCG streaming operations lower overall performance.
brainNek: flop intensive PCG kernels

With OCCA we can run in OpenCL or CUDA mode on NVIDIA GPUs.

A motivating issue for OCCA.

DGTD, FDTD, FV, FEM, SEM solvers exhibit similar issues.
With OCCA we can run in OpenCL or CUDA mode on NVIDIA GPUs.

**NVIDIA Titan (CUDA v OpenCL)**

Floating Point Performance

\[ \mathbf{a}_L = \mathbf{A}_L \mathbf{Zp}_G \]

**Polynomial Order**

1. \( a_L = A_L Zp_G \)

A motivating issue for OCCA.

DGTD, FDTD, FV, FEM, SEM solvers exhibit similar issues.
### brainNek: performance

Comparison of expert coded SEM elliptic solver performance for OAS preconditioned PCG iterative solves: degree 8 elements

<table>
<thead>
<tr>
<th>Code Base</th>
<th>Parallel Model</th>
<th>Processor</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (DP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nek5K</td>
<td>MPI</td>
<td>Intel E5430 (4core)</td>
<td>Est. 6.3</td>
</tr>
<tr>
<td>Nek5k</td>
<td>MPI</td>
<td>Intel SandyBridge</td>
<td>Est. 13</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenMP</td>
<td>Intel i7-3930K (6core)</td>
<td>15 gcc,icpc</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenCL</td>
<td>Intel i7-3930K (6core)</td>
<td>12</td>
</tr>
<tr>
<td>GPU (DP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NekBone</td>
<td>CUDA</td>
<td>NVIDIA K40</td>
<td>NDA</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:CUDA</td>
<td>NVIDIA Titan</td>
<td>94</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenCL</td>
<td>AMD HD 7970</td>
<td>114</td>
</tr>
<tr>
<td>GPU (SP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:CUDA</td>
<td>NVIDIA Titan</td>
<td>196</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenCL</td>
<td>AMD HD 7970</td>
<td>267</td>
</tr>
</tbody>
</table>

Performance aggregated over all kernels in PCG solver including low flop streaming operations.
brainNek: performance

We performed a virtual patient therapy and timed the procedure.

<table>
<thead>
<tr>
<th>Device</th>
<th>Code</th>
<th>Mode</th>
<th>Run Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia GTX 590 (1 GPU)</td>
<td>brainNek</td>
<td><em>OCCA/CL</em></td>
<td>24</td>
</tr>
<tr>
<td>AMD Radeon HD 7970</td>
<td>brainNek</td>
<td><em>OCCA/CL</em></td>
<td>17</td>
</tr>
<tr>
<td>Intel Xeon E5520 (4 cores x2)</td>
<td>brainNek</td>
<td><em>OCCA/CL</em></td>
<td>308</td>
</tr>
<tr>
<td>Intel Xeon E5675 (6 cores x2)</td>
<td>libmesh</td>
<td>CPU</td>
<td>&gt;12 hours (~70% in assembly)</td>
</tr>
</tbody>
</table>

*brainNek on dual socket 8 core Sandy Bridge with 8 wide AVX FPUs runs >2x faster.*
Summary

• Online patient specific MRgLITT modeling is feasible.

• *brainNek* demoed to neurosurgeons.

• In progress: improved tissue model and laser fluence model.

• OCCA2 can deliver portability without impacting performance.

• Focusing on algorithms and hardware gave substantial speedup [lost in GPU noise].

brainNek: close collaboration with domain experts at MD Anderson Cancer Center

Thank You