SOS14 Panel Discussion

Software programming environments and tools for GPUs

Where are they today?
What do we need for tomorrow?

Savannah, GA

09 March 2010

John A. Turner, ORNL

Allen Malony, Univ. of Oregon

Simone Melchionna, EPFL

Mike Heroux, SNL

DOE INCITE applications span a broad range of science challenges

Projects	2006	2007	2008	2009
Accelerator physics	1	1	1	1
Astrophysics	3	4	5	5
Chemistry	1	1	2	4
Climate change	3	3	4	5
Combustion	1	1	2	2
Computer science	1	1	1	1
Fluid Dynamics			1	1
Fusion	4	5	3	5
Geosciences		1	1	1
High energy physics		1	1	
Life sciences	2	2	2	4
Materials science	2	3	3	4
Nuclear physics	2	2	1	2
Industry	2	3	3	3
Total Projects:	22	28	30	38
CPU Hours:	36,156,000	75,495,000	145,387,000	469,683,000

Application Challenges

- Node-level Concurrency
  - finding sufficient concurrency to amortize or hide data movement costs
  - thread creation and management
- Data movement
  - managing memory hierarchy
  - minimizing data movement between CPU and GPU
- Programming models
  - expressing the concurrency and data movement mentioned above
  - refactoring applications undergoing active development
  - maintaining code portability and performance
- Compilers and tools
  - availability, quality, usability
  - ability to use particular language features (e.g. C++ templates)

Software programming environments and tools for GPUs

(Where are they today? What do we need for tomorrow?)

- How are you programming GPUs today?
  - In preparing apps for hybrid / accelerated / many-core platforms, what is your preferred path for balancing performance, portability, and “future-proofing”?
  - What is the typical effort required to refactor applications for hybrid / accelerated / many-core platforms?
- What role does performance modeling play, if any?
  - What role can / should it play?

Software programming environments and tools for GPUs

(Where are they today? What do we need for tomorrow?)

- What programming environment tools are key to preparing applications for coming platforms?
  - What tools are most in need of improvement & investment?
- Do the application modifications for existing and near-term platforms (Roadrunner, GPU-accelerated systems) better prepare them for other future platforms envisioned in the Exascale roadmap?

Software programming environments and tools for GPUs

(Where are they today? What do we need for tomorrow?)

- What is difficult or missing from your current programming models?
- What important architectural features must be captured in new programming models?
- What are the possible future programming models 10 and 20 years from now?

Software programming environments and tools for GPUs

(Where are they today? What do we need for tomorrow?)

- In previous major shifts such as vector to distributed-memory parallel, some applications did not survive or required such major re-factoring that for all intents and purposes they were new codes.
  - What fraction of applications would you predict will fail to make this transition?
  - Will the survival rate be significantly different between “science” apps and “engineering” apps?
  - Software has become more complex since the last major transition, but also tends to make greater use of abstraction. In the end does this help or hinder applications in moving to these architectures?
  - Is there an aspect to survival that could be viewed as a positive rather than a negative? That is, does the “end of the road” for an application really present an opportunity to re-think the physics / math / algorithms / implementations?

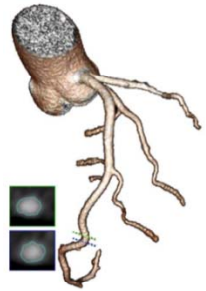
Allen Malony slides

High Performance Simulations of Complex Biological Flows on CPUs and GPUs

Simone Melchionna
EPFL

EPFL, CH Harvard, US CNR, Italy

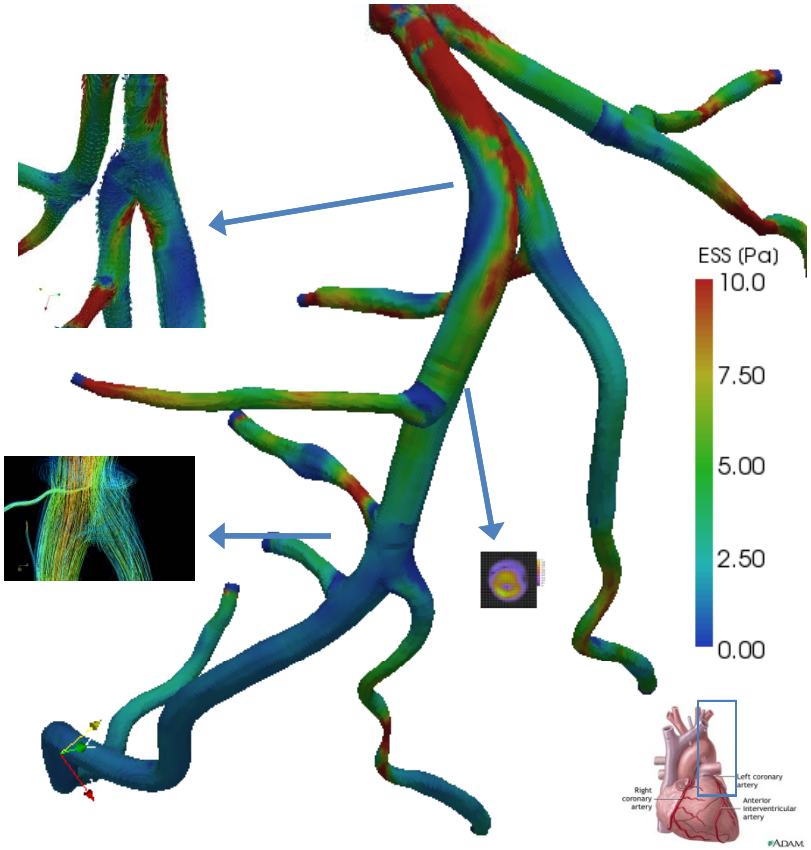
Multi-scale Hemodynamics

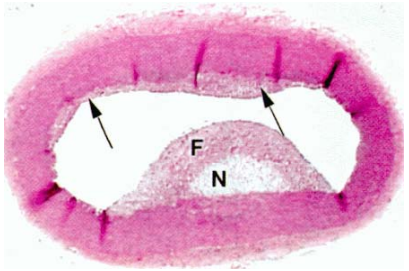


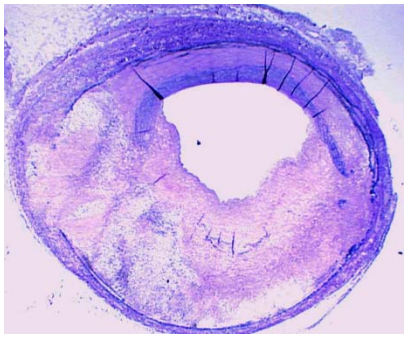
Level-set to 3D reconstruction

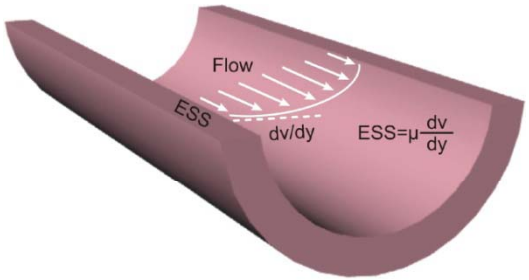
Mesh generation

Domain partitioning



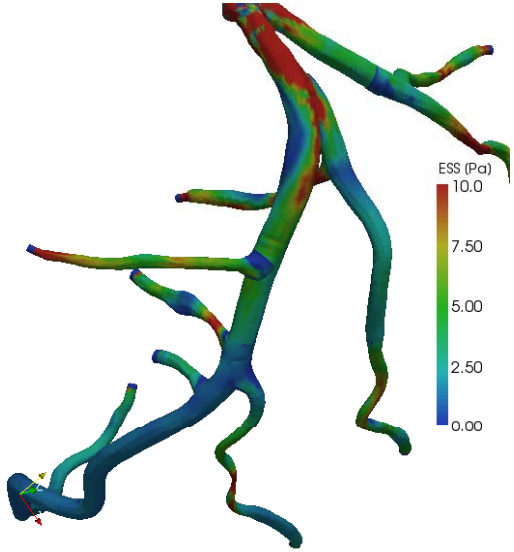


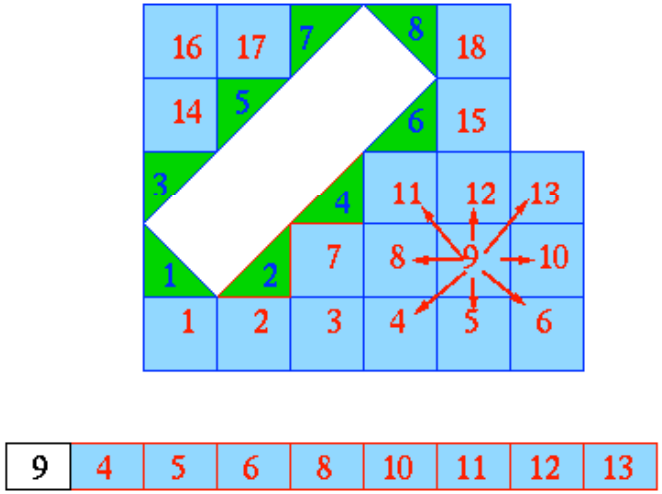


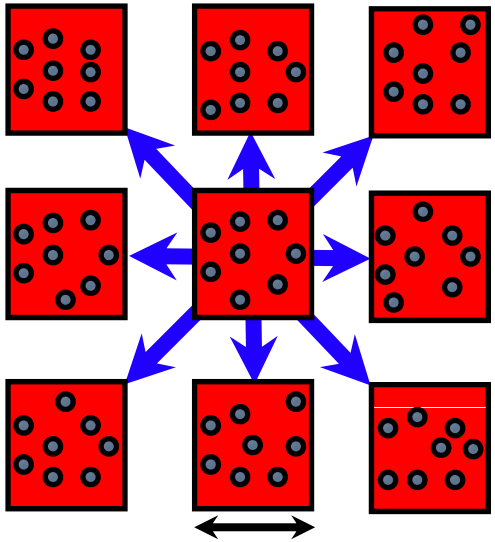


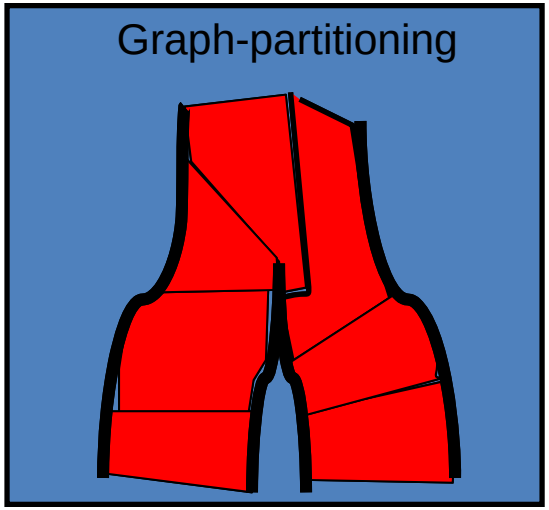
Vascular Remodeling

MUPHY: MULti-PHYsics simulator









Multiscale computing: GPU challenges

Heterogeneous Multiscale Simulation: does not necessarily fit Heterogeneous Computing and data-parallel accelerators. Method modifications have long latency for code modifications.

Great constraints on programmers: 1) Avoid conditionals 2) Avoid non-coalesced Global Memory access 3) Lack of inter-block comms 4) Data layout critical.

Rethinking consolidated simulation methods: a great practice.

Hybrid programming model: efficient ☺, accessible ☹, vulnerable ☹

For Multiscale sims several components do not map on threads/coalesced mem access. E.g. bisection search, ordered traversing of matrix elements.

Development is not an incremental porting: CPU-GPU bandwidth critical step.

Wish list

Overcome the heroic stage: more accessible programming model and lift constraints on programmers. More room for science! Less strain on global code rewriting (scripting & meta-programming). Allow for incremental porting.

Overcome programming models vulnerabilities (long modification latency) and increase level of abstraction

More libraries and coarse/fine grained algorithmics. Profiling & debugging tools. More testbeds, more HW emulators, more mature ties with community. Avoid the xN temptation.

Establish models and best practice for established simulation methods. Put forward a GPU landscape on the simulation methods space. Establish models on data locality (method rethinking!).

Moving object simulation methods: establish new roadmaps.

Mike Heroux: Programming Environments for GPUs

- In preparing apps for hybrid / accelerated / many-core platforms, what is your preferred path for balancing performance, portability, and "future-proofing"?
  We have developed a node API using C++ template meta-programming. It is similar to CUDA (actually Thrust) with the OpenCL memory model (Trilinos NodeAPI).

- What is the typical effort required to refactor applications for hybrid / accelerated / many-core platforms?
  We started with a clean slate (Tpetra) instead of refactoring Epetra.

- What programming environment tools are key to preparing applications for coming platforms. What tools are most in need of improvement & investment?
  We need a portable programming model, higher level than OpenCL. CUDA level is sufficient. Something like CUDA SDK is essential.

- What role can performance modeling play, if any?
  Some, but working example suite is even better, or miniapps.

- Do the application modifications for existing and near-term platforms (Roadrunner, GPU-accelerated systems) better prepare them for other future platforms envisioned in the Exascale roadmap?
  Yes, if not in reusable software, at least in strategy and design.

Mike Heroux: Programming Environments for GPUs

- In previous major shifts such as vector to distributed-memory parallel, some applications did not survive or required such major re-factoring that for all intents and purposes they were new codes.
  - What fraction of applications would you predict will fail to make this transition?
    Many will make it, at least as "petascale kernels".
  - Will the survival rate be significantly different between "science" apps and "engineering" apps?
    Science "forward problems" appear to be more scalable than engineering forward problems. But engineering apps are more ready for advanced modeling and simulation, UQ, etc.
  - Software has become more complex since the last major transition, but also tends to make greater use of abstraction. In the end does this help or hinder applications in moving to these architectures?
    Proper abstraction can be a huge advantage. Caveat: Abstraction can solve any problem except too much abstraction.
  - Is there an aspect to survival that should be viewed as a positive rather than a negative? That is, does the "end of the road" for an application really present an opportunity to re-think the physics / math / algorithms at a deeper level?
    Yes, but parallelism cannot overcome inferior algorithm complexity.

Trilinos Node API Example Kernels: `axpy()` and `dot()`

```
template <class WDP>
void
Node::parallel_for(int beg, int end,
                   WDP workdata    );
```

```
template <class WDP>
WDP::ReductionType
Node::parallel_reduce(int beg, int end,
                      WDP workdata    );
```

```
template <class T>
struct AxpyOp {
  const T * x;
  T * y;
  T alpha, beta;
  void execute(int i)
  { y[i] = alpha*x[i] + beta*y[i]; }
};
```

```
template <class T>
struct DotOp {
  typedef T ReductionType;
  const T * x, * y;
  T identity()       { return (T)0;      }
  T generate(int i)  { return x[i]*y[i]; }
  T reduce(T x, T y) { return x + y;     }
};
```

```
AxpyOp<double> op;
op.x = ...;  op.alpha = ...;
op.y = ...;  op.beta  = ...;
node.parallel_for< AxpyOp<double> >
                 (0, length, op);
```

```
DotOp<float> op;
op.x = ...;  op.y = ...;
float dot;
dot = node.parallel_reduce< DotOp<float> >
                          (0, length, op);
```

Work with Chris Baker, ORNL

Some Additional Thoughts

- New Programming Models/Environments:
  - Must compete/cooperate with MPI.
  - MPI advantage: Most code is serial.
- New PM/E:
  - Truly useful if it solve the ubiquitous markup problem.
  - Conjecture:
    - Data-intensive apps will drive new PM/Es.
    - Our community will adopt as opportunities arise.
- Source of new PM/Es: Industry.
  - Best compilers are commercial.
  - Runtime system is critical.