

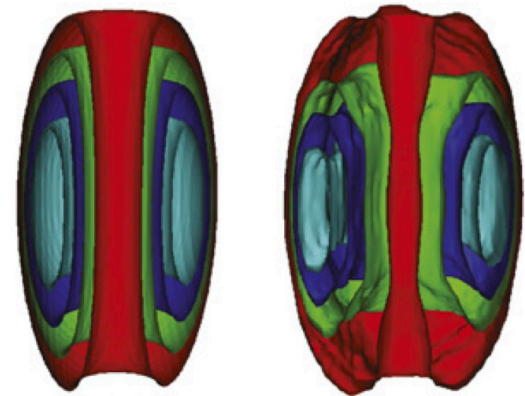
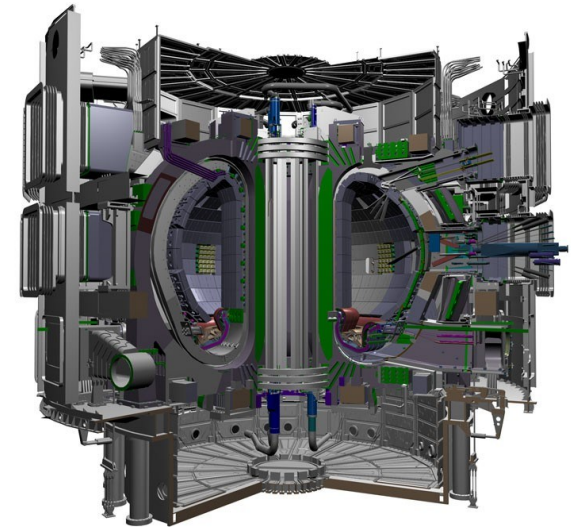
# Communication hierarchies in emerging many-core HPC architectures and one design based on OpenCL

Noah Reddell



# Motivation: Next-Generation Plasma Simulation

- Goal: model Ions, Electrons, and Neutrals as separate explicit fluids that interact through both surface and body forces including coupling to Electromagnetic fields.
- Allow for regimes where plasma is not in local thermodynamic equilibrium through higher moment's of Maxwell-Boltzmann equation or even **full kinetic representation**.
- Applications: fusion innovative confinement concepts, plasma actuators for supersonic craft, industrial apps., ICF, fundamental physics.



# Design Strategy

- Advanced plasma models involve increased computational demands.
- Architecture trends are dominated by increased core parallelism and increased relative penalty of data movement.
- Desire a code flexible for a wide class of problems and based on enduring standards and libraries.
  - C++, OpenCL, boost, HDF5, GlobalArrays

# Influencing Trends

- Integrated Circuit gate size approaching physical limits
- Clock speed has peaked
- Total machine power for HPC systems possibly the most significant cost factor
- Nearly all new architecture investment motivated by the consumer electronics / gaming sector.



# Contemporary GPU architecture

- 512 cores or active threads
- Several light weight cores per instruction unit
  - 16 cores comprising 'Streaming Multiprocessor'
- Memory Hierarchy
  - 2-8kB private memory per core (Fermi) as registers
  - Up to 48 kB shared local memory among 16 cores (Fermi)
  - 1-2GB Global DRAM
  - These resources shared among concurrent threads (6 a good number to hit) AND other active kernels.
- Separated from CPU by PCIe bus.

# Notables of next GPU architecture

- NVIDIA Kepler
- 28 nm process
- PCIe 3.0 Doubles BW over PCIe 2.0
  - 16 GB/s
- 1536 CUDA Cores
- 192 GB/s GPU Main Memory BW
- 3.1 TFLOPS FMA @ 195 W
  - 15.8 GFLOPS/W

# Upcoming CPU architectures

- Intel Many Integrated Cores
- 22nm process
- ~50 cores per chip
- Eight-wide SP / four-wide DP vector units based on existing SIMD and AVX instructions
- Significant development of auto-vectorizing compiler technology to transform scalar SIMD code into vector instructions.
- Shared memory capability carved out of L3 cache.



# OpenCL

- Open and royalty-free standard for multi-core and many-core programming supported by dozens of HPC companies.
- ...but implemented by ~~few~~ a growing base
  - NVIDIA CUDA SDK on Linux
  - AMD Streaming SDK on Linux
  - Apple OS X (AMD+NVIDIA GPU, Intel CPU)
  - Intel OpenCL SDK
  - IBM OpenCL Development (POWER7 CPU)
- Based on C99 with additions and omissions
- Data-parallel programming model
- Not very abstracted. Developer must deal with data locality and hardware specifics

# OpenCL kernels

Single-threaded C version:

```
void
trad_mul( int n,
          const float * a,
          const float * b,
          float * c )
{
    for( int i=0; i<n; i++ )
        c[i] = a[i] * b[i];
} // one instance loops over "n" work items
```

OpenCL version:

```
kernel void
cl_simple_mul(
    global const float * a,
    global const float * b,
    global float * c )
{
    int id = get_global_id(0);

    c[id] = a[id] * b[id];
} //instances executed for "n" work items
```

# OpenCL API

- Host side library calls are used to:
  - Compile and load kernels
  - Define work dimensions
  - Move memory between buffers and host RAM
  - Establish queuing sequence for kernels
  - Check status of running jobs

# NERSC and NCSA OpenCL support

- **Dirac** – NERSC - 44 nodes NVIDIA C2050 ‘Fermi’, 4 nodes NVIDIA C1060 Tesla. OpenCL through NVIDIA CUDA SDK
- **Accelerator Cluster** – NCSA – 40 nodes with a mix of NVIDIA GPU, AMD GPU, AMD CPUs. OpenCL through NVIDIA CUDA SDK or AMD Stream SDK
- **Lincoln** – NCSA – 192 nodes with NVIDIA Tesla S1070.

# Jaguar upgrade to Titan

- Demonstrates continued commitment to many-core / GPU based nodes. (est. 2013)
- 7k – 18k NVIDIA Tesla GPUs
- 16-core AMD CPU
- Integrated OpenCL platform please...





# Data movement is precious

- PCIe 2.0 x16
  - 8 GB/s host to GPU and 8 GB/s GPU to host
- Using NVIDIA Fermi M2050 specs:
- Peak performance: 515 Gflop/s DP
- $515 \text{ Gflop/s} \div 8 \text{ GB/s} \times 8 \text{ bytes/double}$ 
  - 515 floating point operations needed per operand transferred between GPU and host to hide PCIe bottleneck
- GPU Memory bandwidth: 148 GB/s to cores
  - 28 floating point operations needed per operand transferred between GPU GRAM and core to hide memory interface bottleneck

# Data movement severely affects power

- Energy requirements for one FLOP around 50 pJ for FMA with operands held locally in registers or L1 cache.
- But moving data and other overhead accounts for about 1k-10k pJ per FLOP.
  - In one study 1.3k pJ for local DRAM, 14k pJ for distributed (MPI) memory access.<sup>1</sup>

1. Kogge, 2011 Hardware Evolution Trends of Exascale Computing.

# Tight parallelism important too

- Especially for GPU implementations
- NVIDIA and AMD GPUs both are based on architectures with many more functional units than instruction units and schedulers.
- Algorithms that support all kernel instances simultaneously performing the same operation on different data perform best
- CPU implementations must auto-vectorize to utilize SSE3 and AVX hardware.

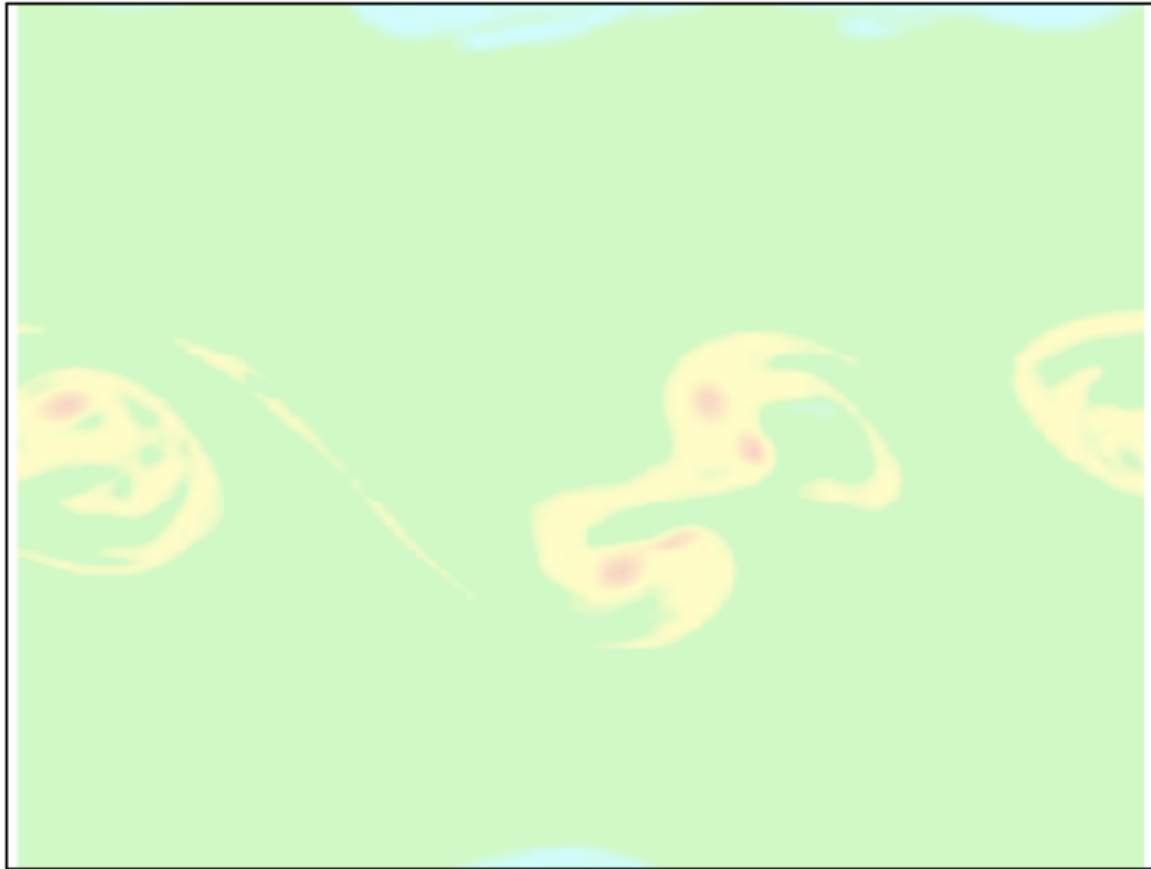
# Motivation for new fluid solver

- Design for many-core architectures
- Move memory less
  - The common denominator in impediments to exascale: power and bandwidth
- Use established packages where possible
- Support problem sizes larger than GPU DRAM capacity.
- Support High-order methods
  - Immediate focus on explicit methods
  - Targeting locally-implicit methods for stiff problems
  - Maintain flexibility for fully implicit approach
- Be useful to others in the community

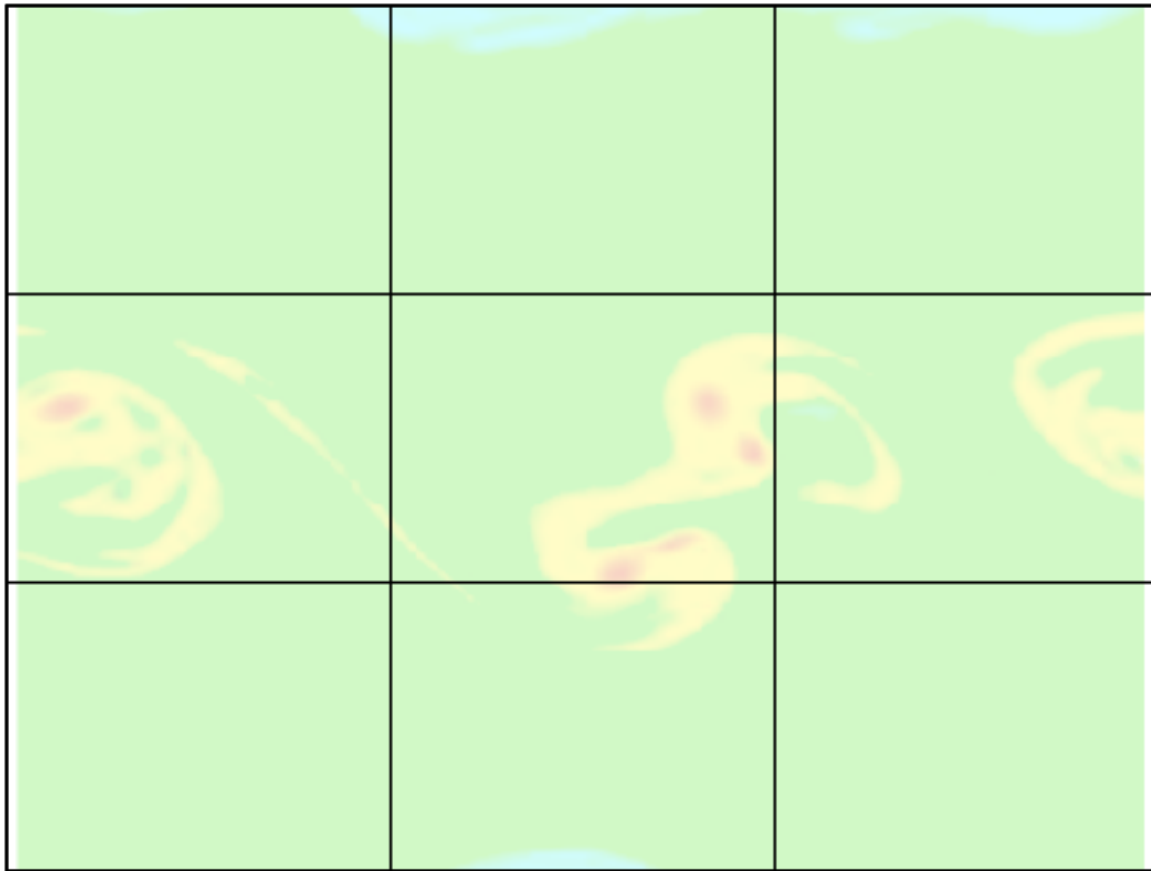
# Finite Volume Methods

- Structured solution for hyperbolic PDEs applied to domains decomposed into cells, or volumes.
- Explicit approach yields parallel algorithm with only nearby-neighbor communication.
- 1<sup>st</sup>-order, 2<sup>nd</sup>-order, and higher-order extensions developed.

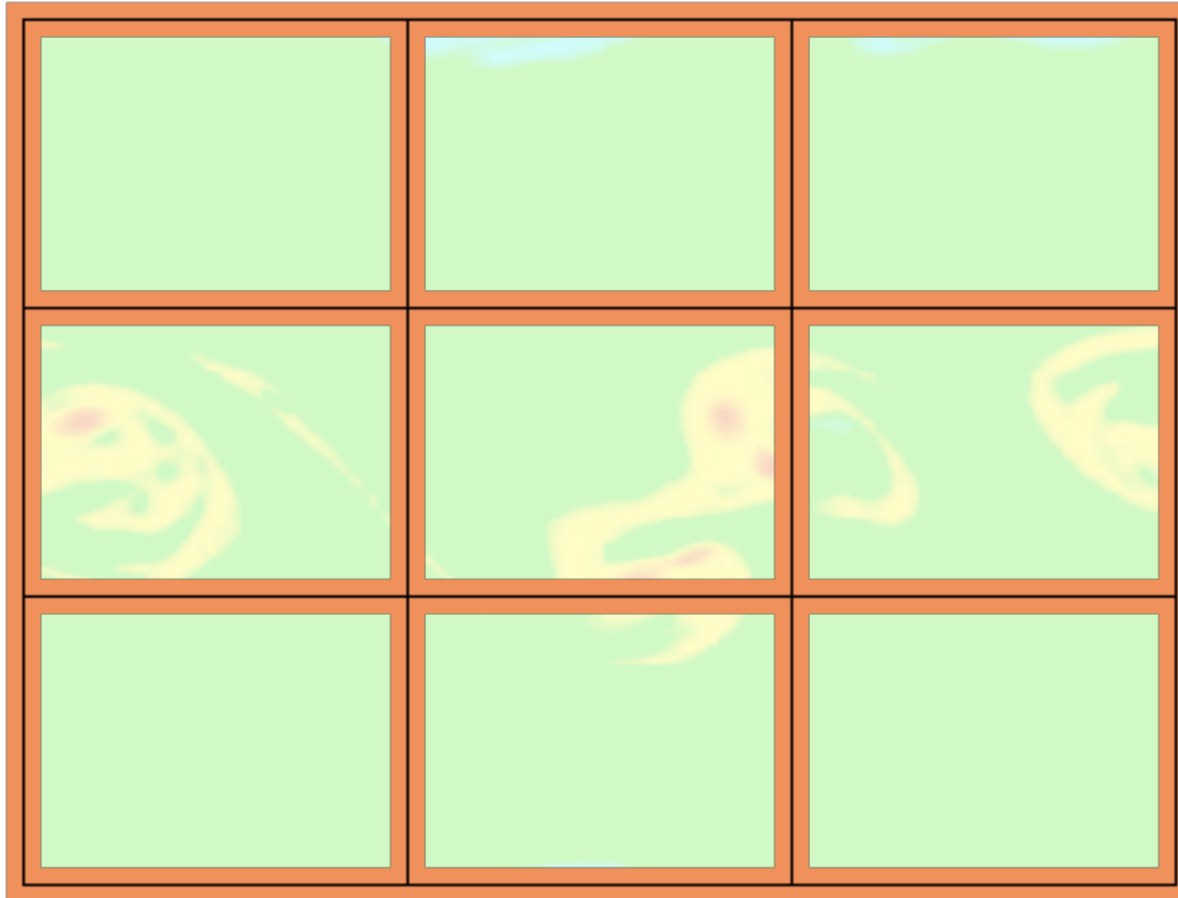
# Multi-level domain decomposition



# Multi-level domain decomposition

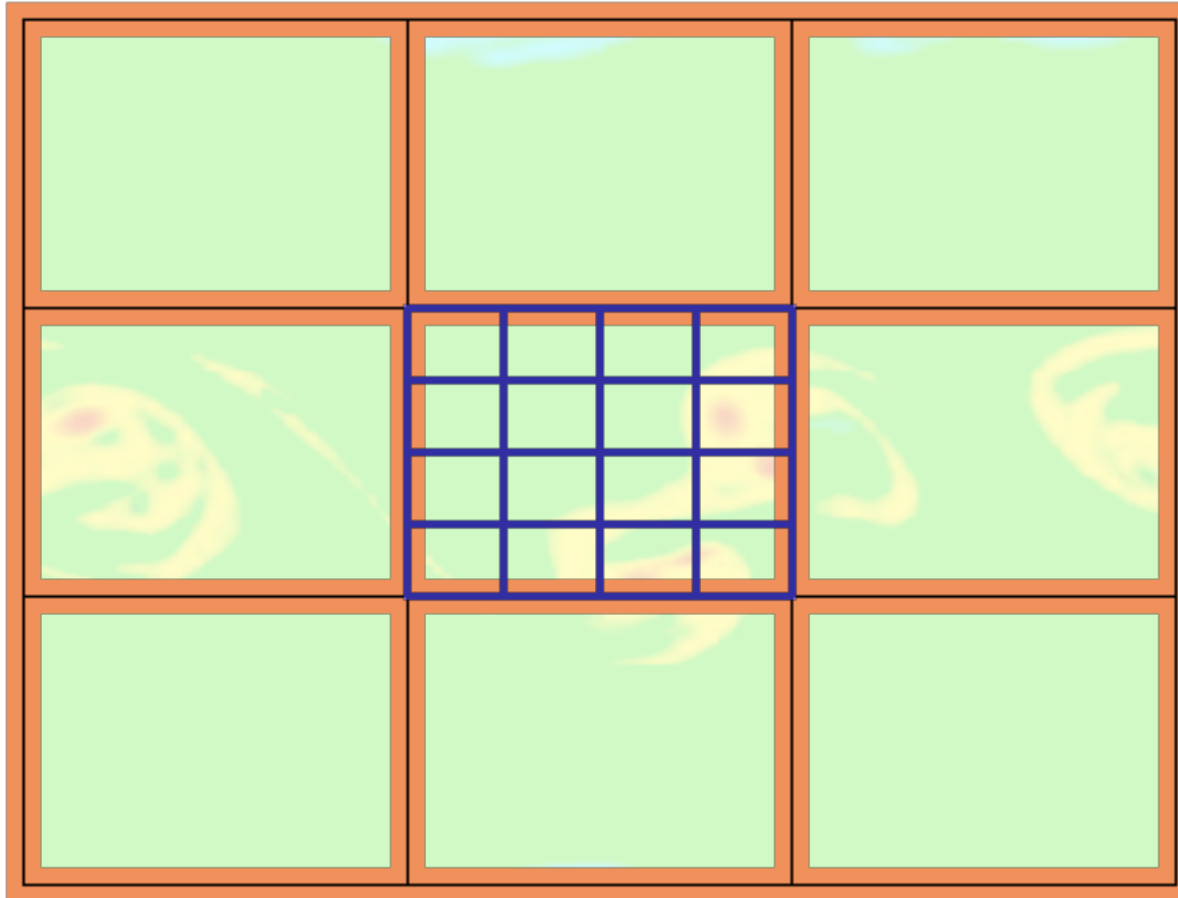


# Multi-level domain decomposition

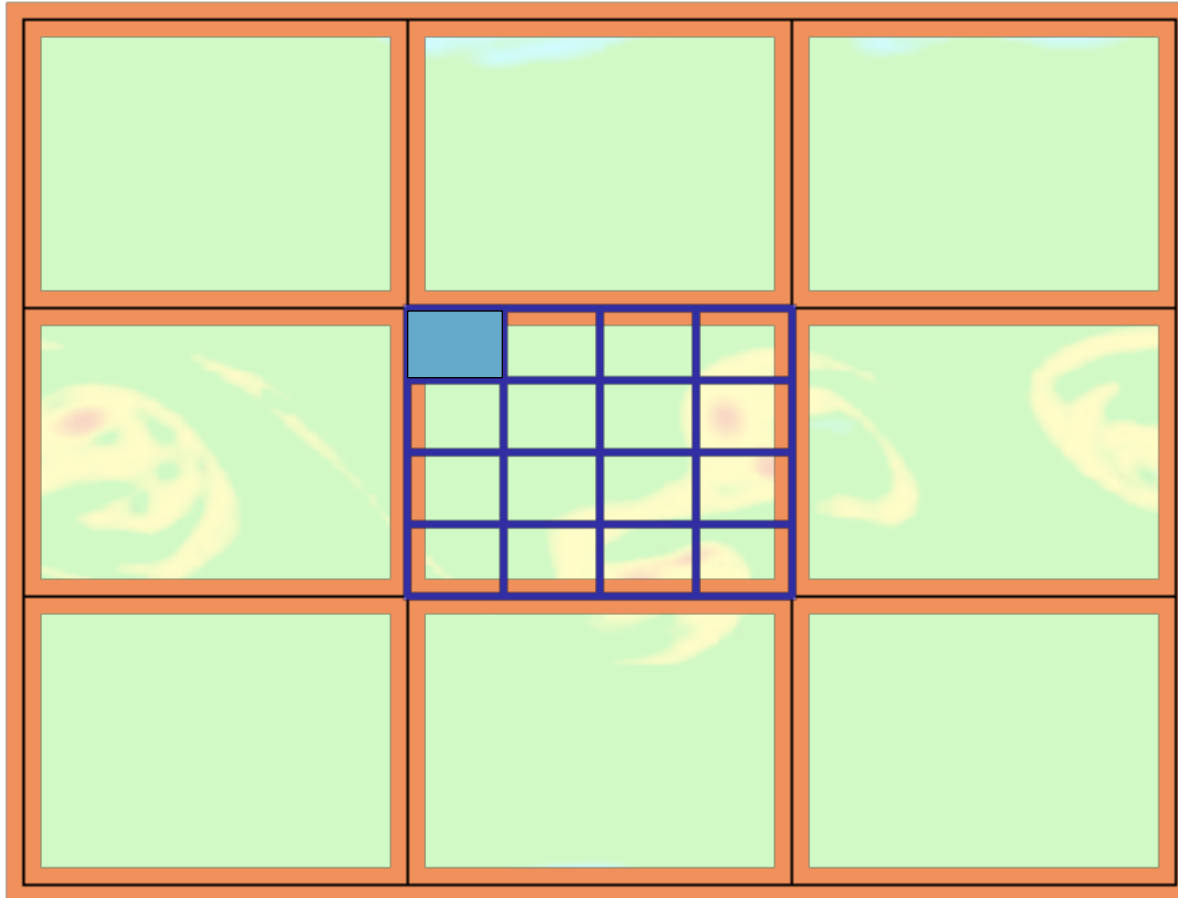




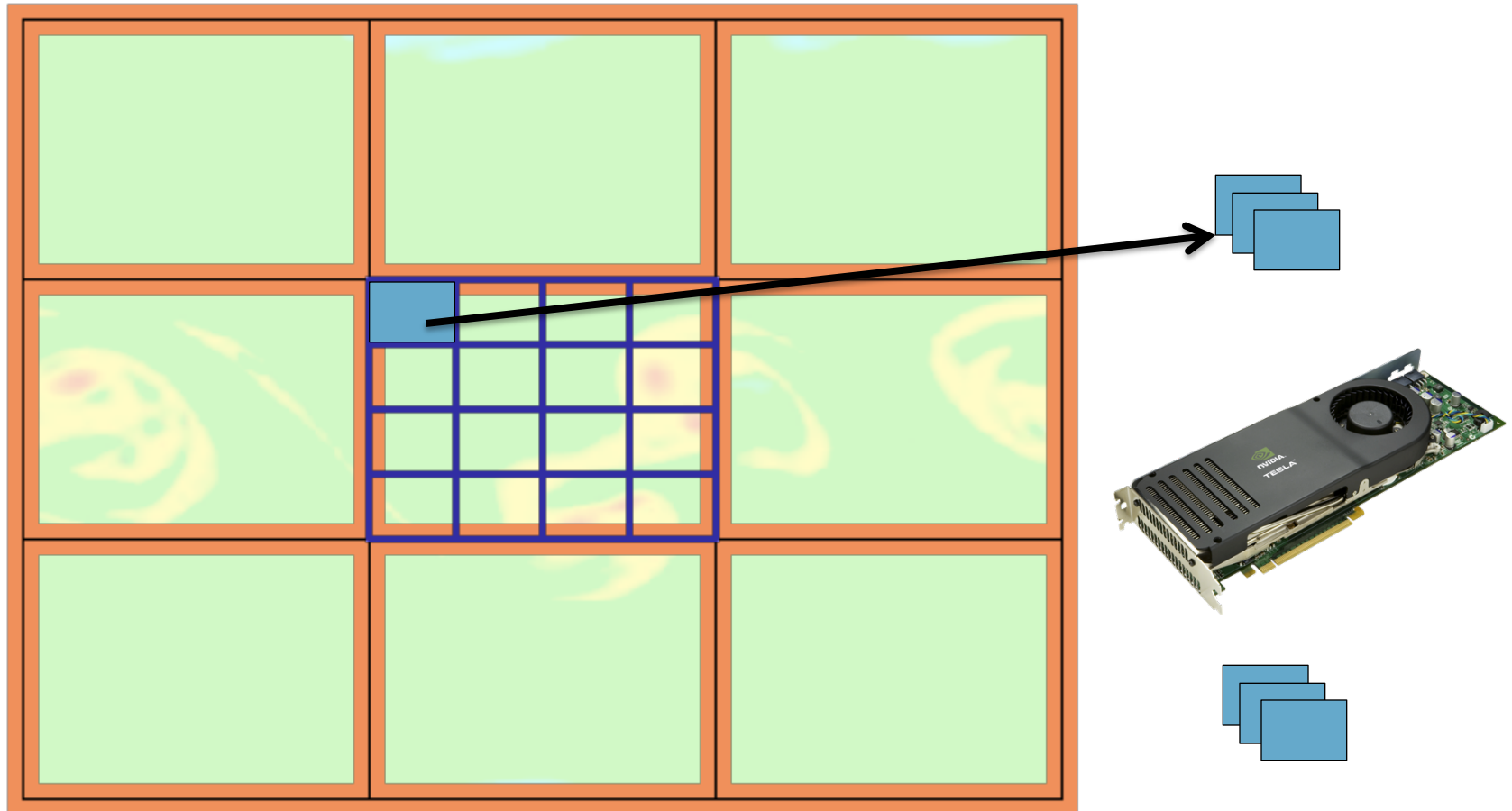
# Multi-level domain decomposition



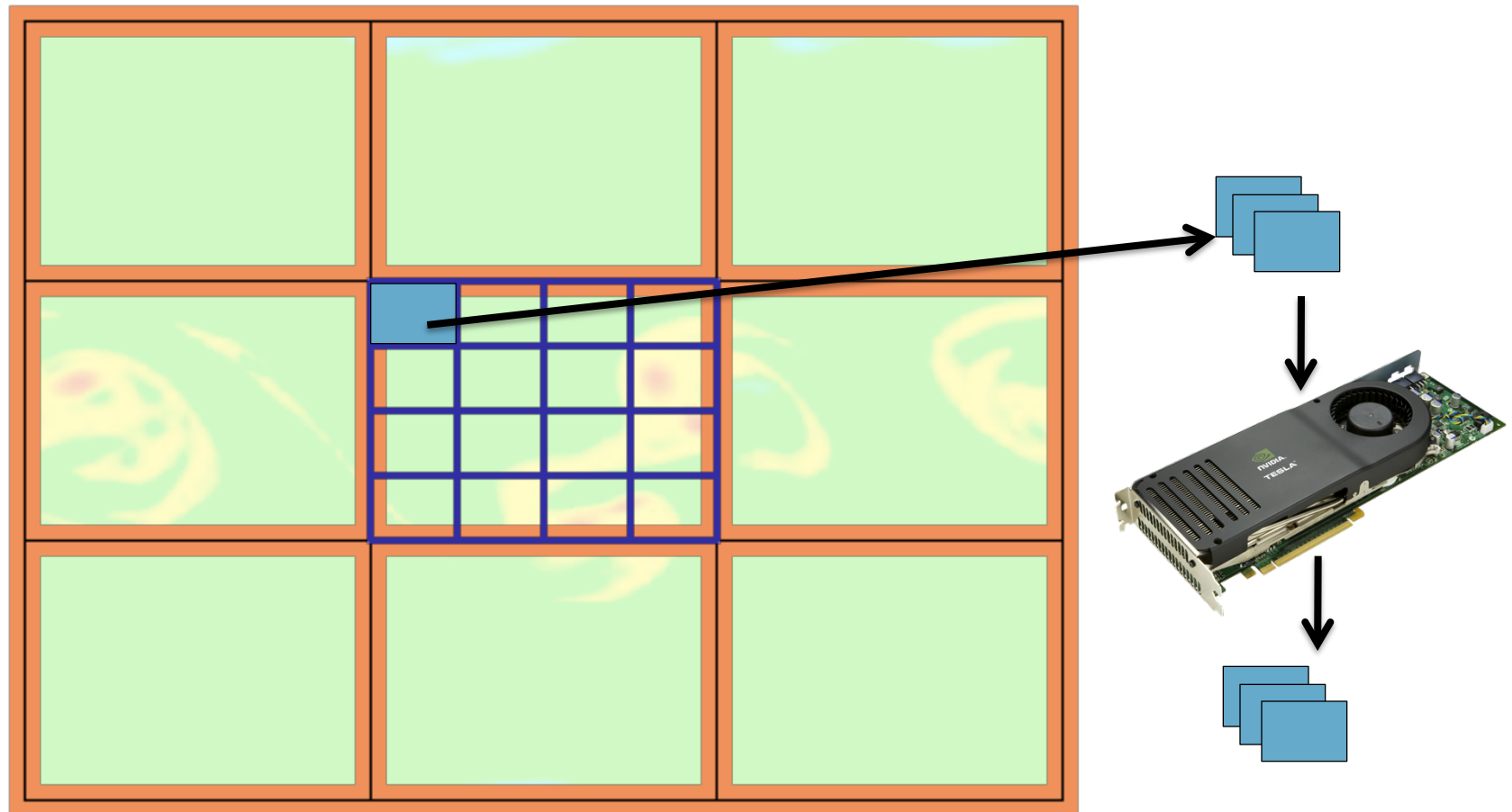
# Multi-level domain decomposition



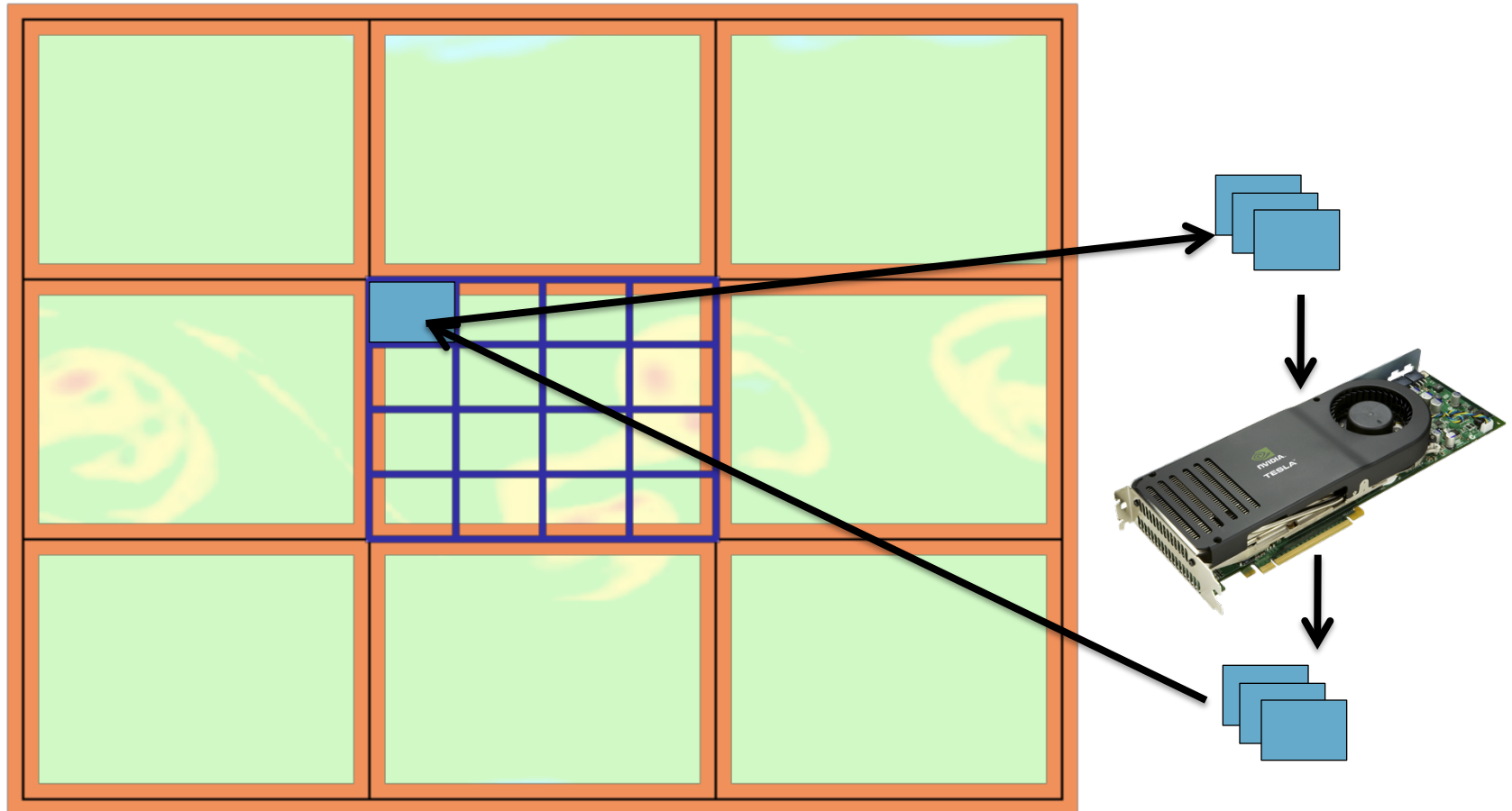
# Multi-level domain decomposition



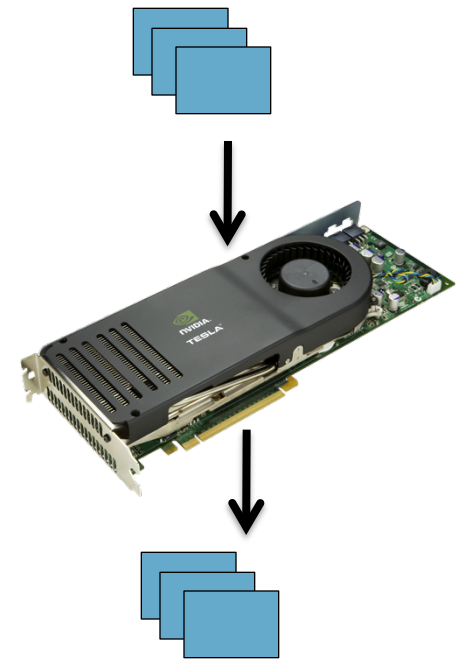
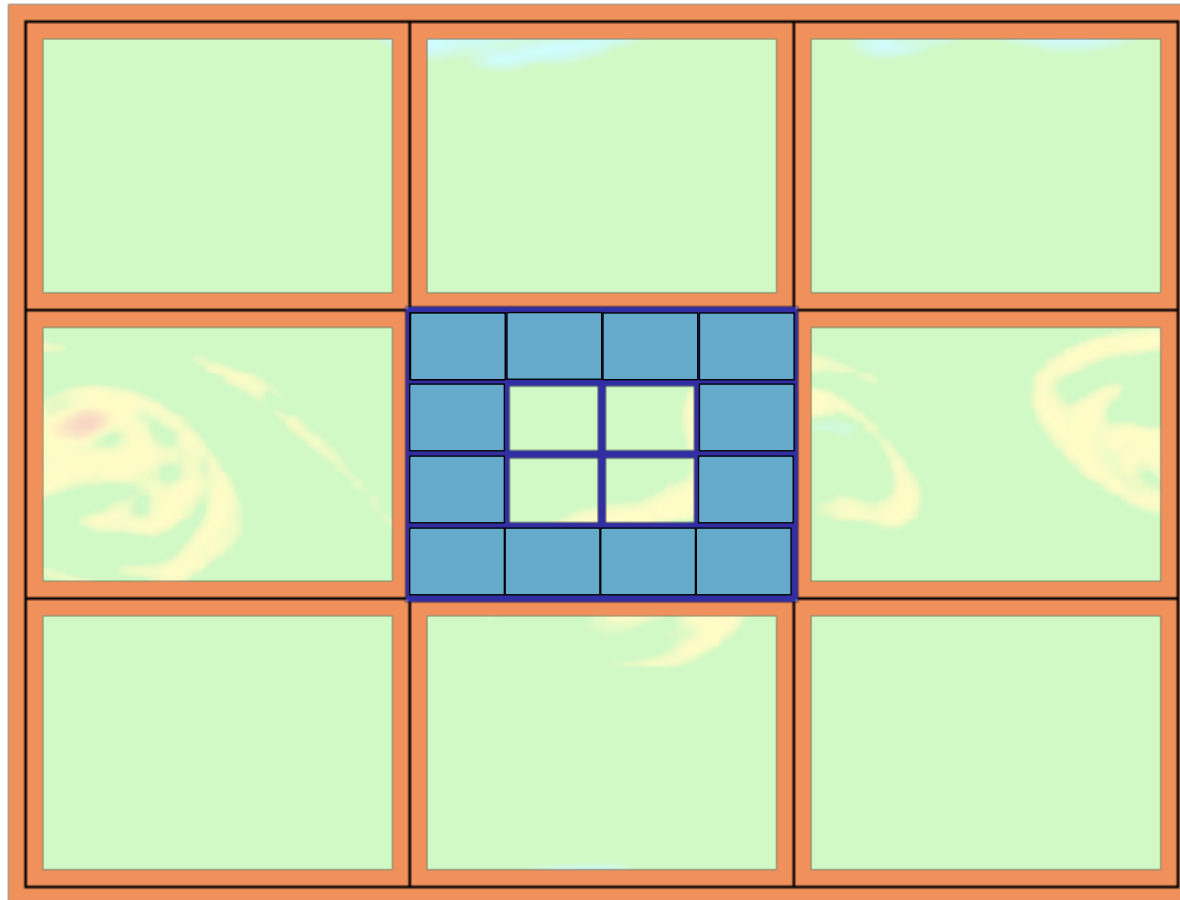
# Multi-level domain decomposition



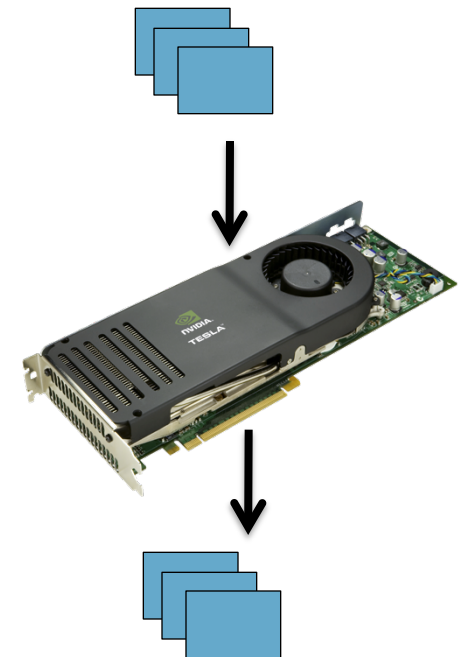
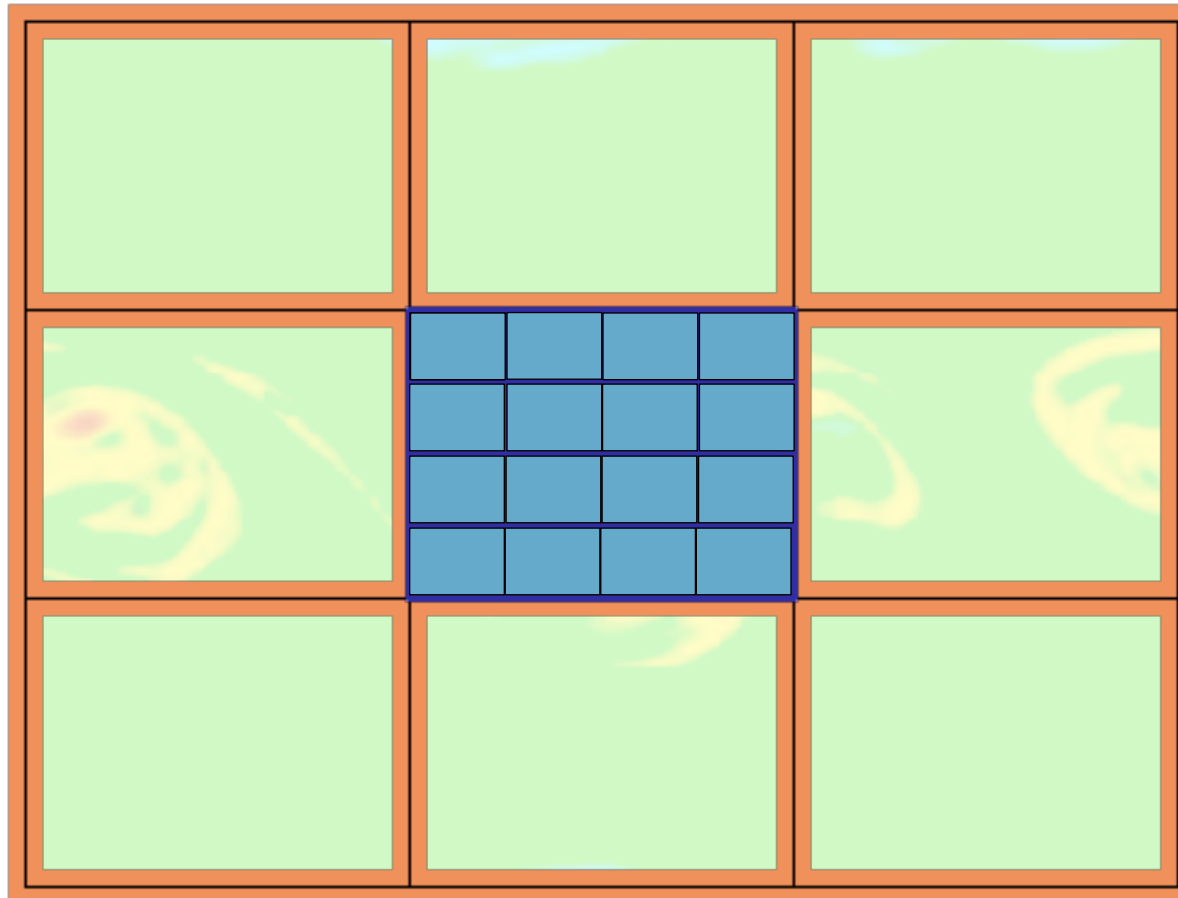
# Multi-level domain decomposition



# Multi-level domain decomposition



# Multi-level domain decomposition



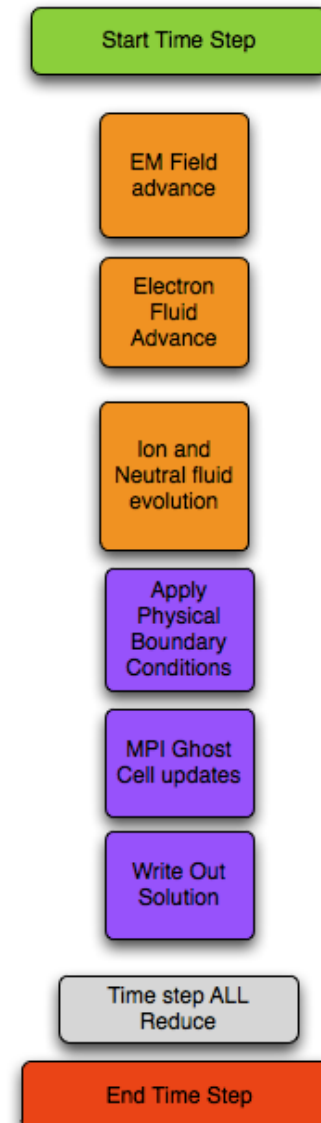
(Plus pinned memory efforts.)

# Course and fine parallelism

Transformed our previous plasma simulation code WARPX which relies solely on MPI for parallelism to one that makes better use of modern architectures.

Starting point: 1 MPI process per core, all memory sharing through MPI Send/Receive, all steps execute in series.

Common story for many HPC codes.

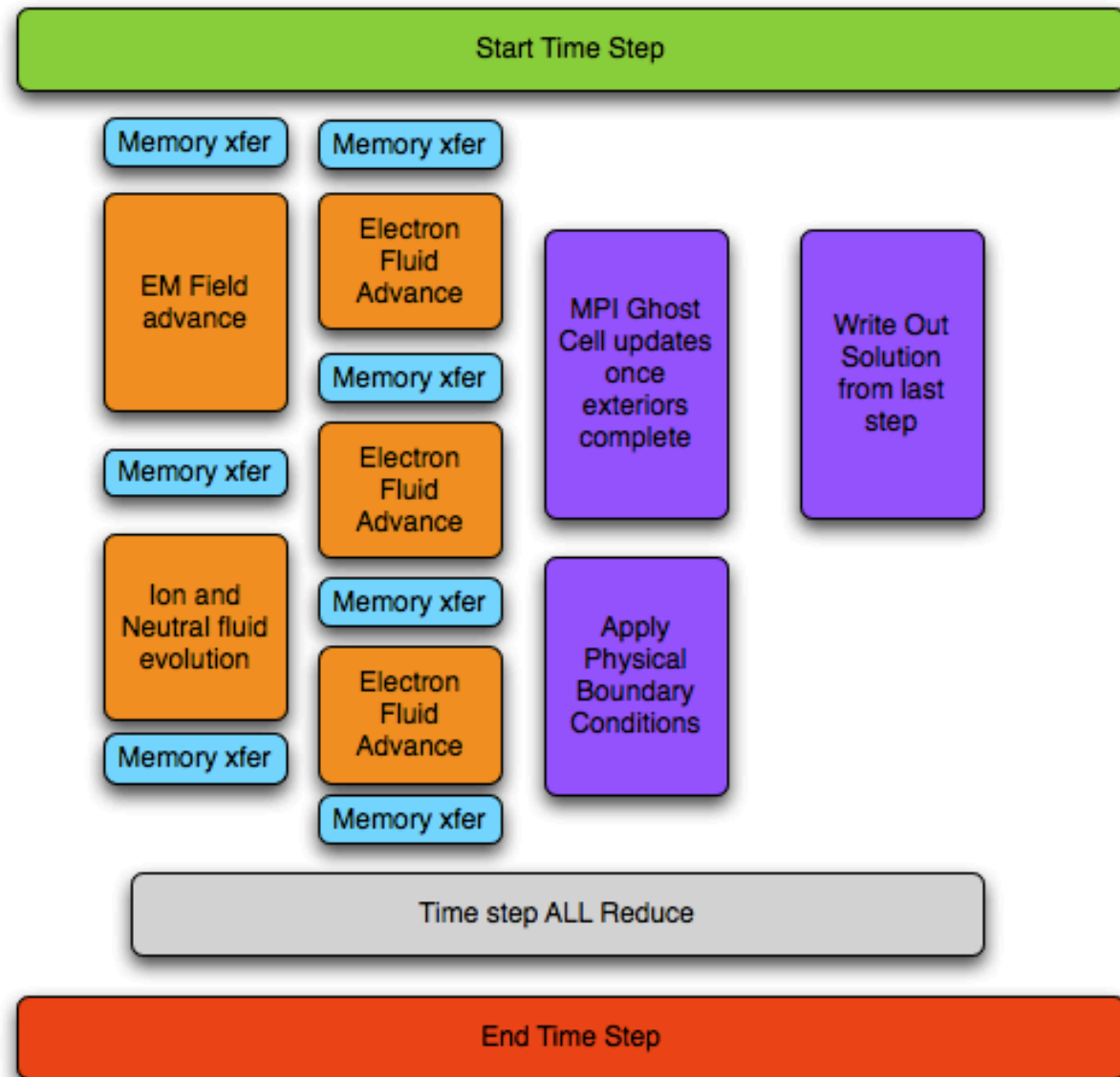




# Course and fine parallelism

First attempt at OpenCL implementation uses multiple kernels and thus more GPU global memory transfer than necessary.

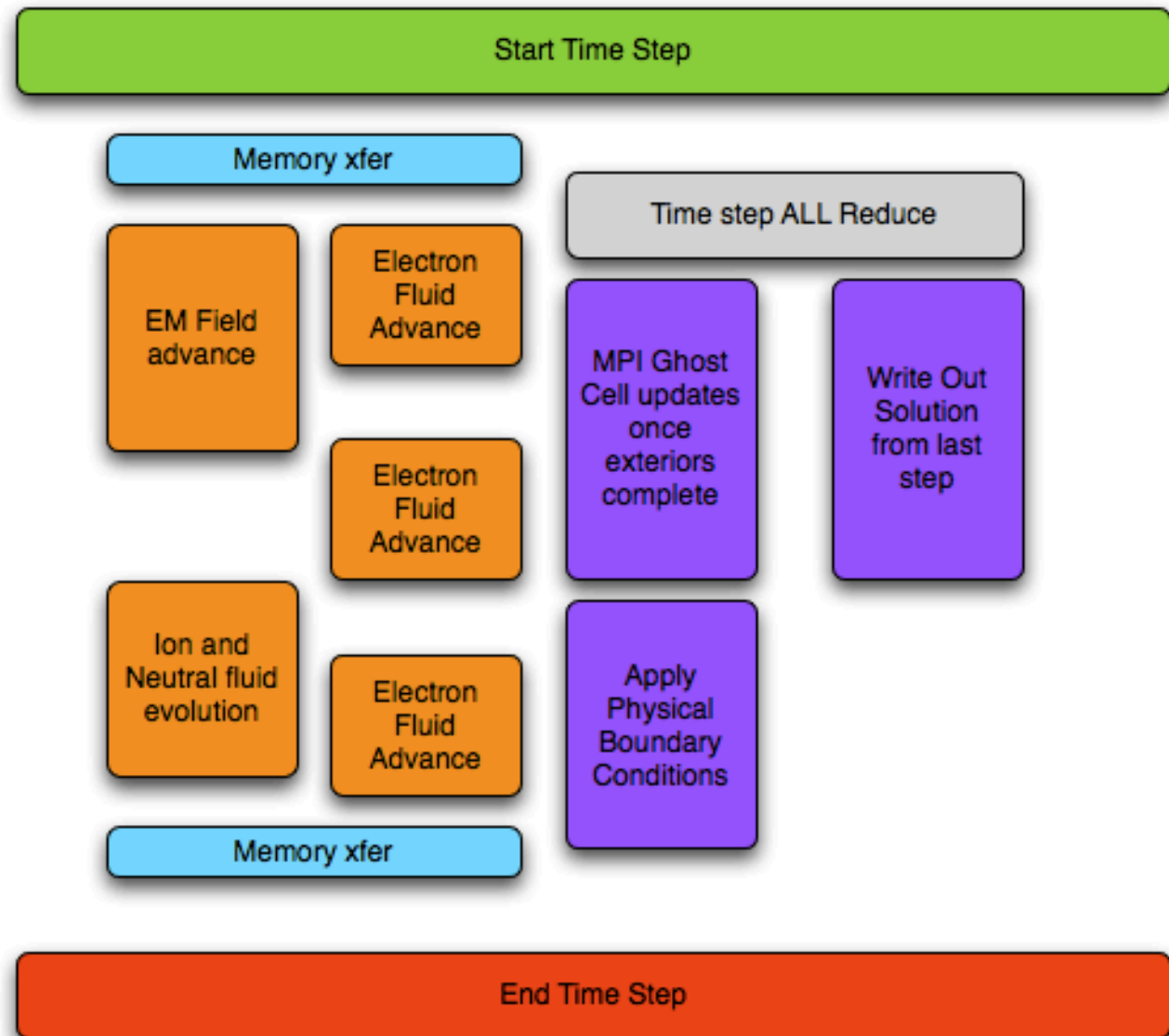
Mix of GPU and CPU computing. OpenCL, pthreads, and MPI parallel execution models combined.



# Course and fine parallelism

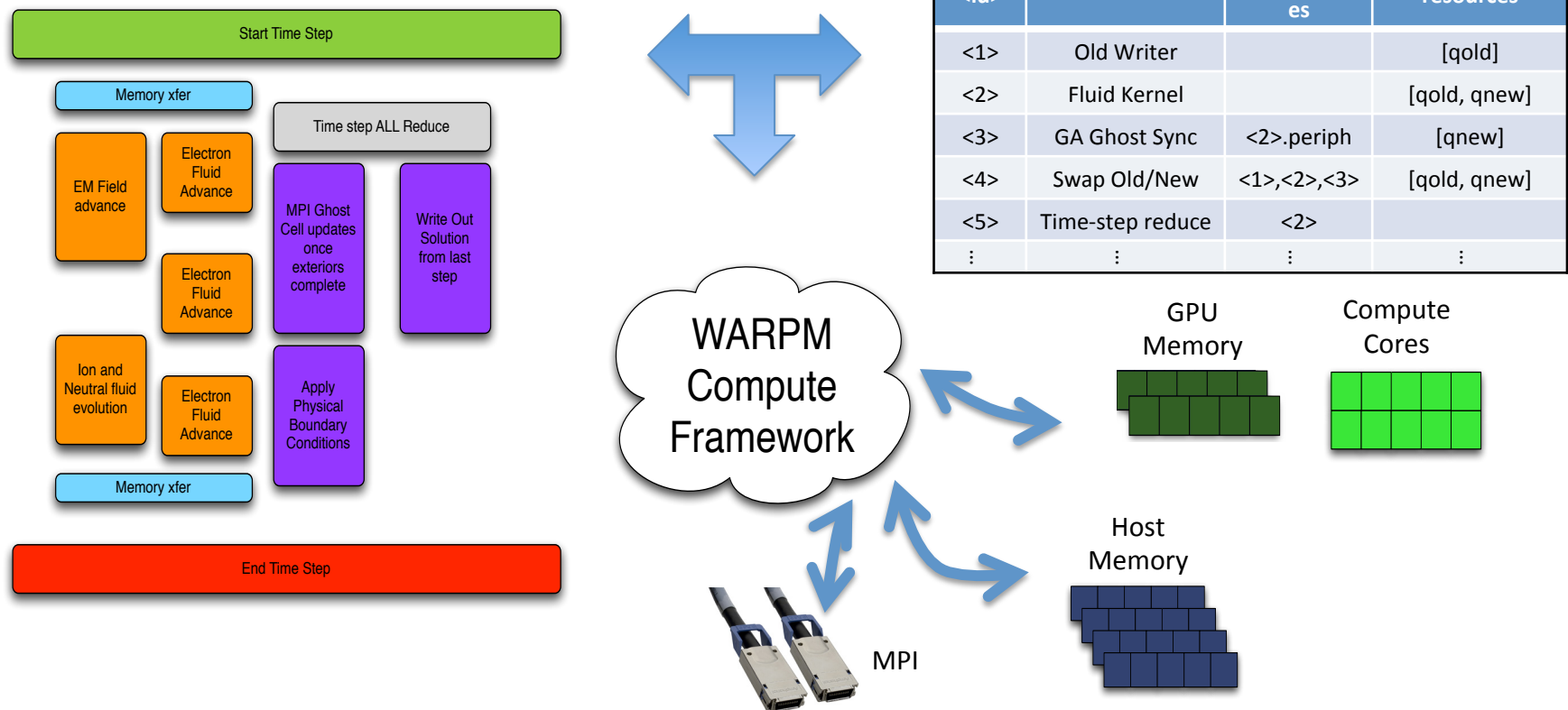
WARPM targets consolidated kernel model

- Framework supporting local/explicit numerical methods for fluids and kinetic systems of hyperbolic equations.
- ‘M’ for many-core
- Motivated by eliminating unnecessary memory transfers
- Natural support for heterogeneous computing on HPC clusters



# WARPM Framework

Orchestrates computational kernels, memory movement, and disk i/o based on user specified dependencies



Users supply the computational model and evaluation sequence.

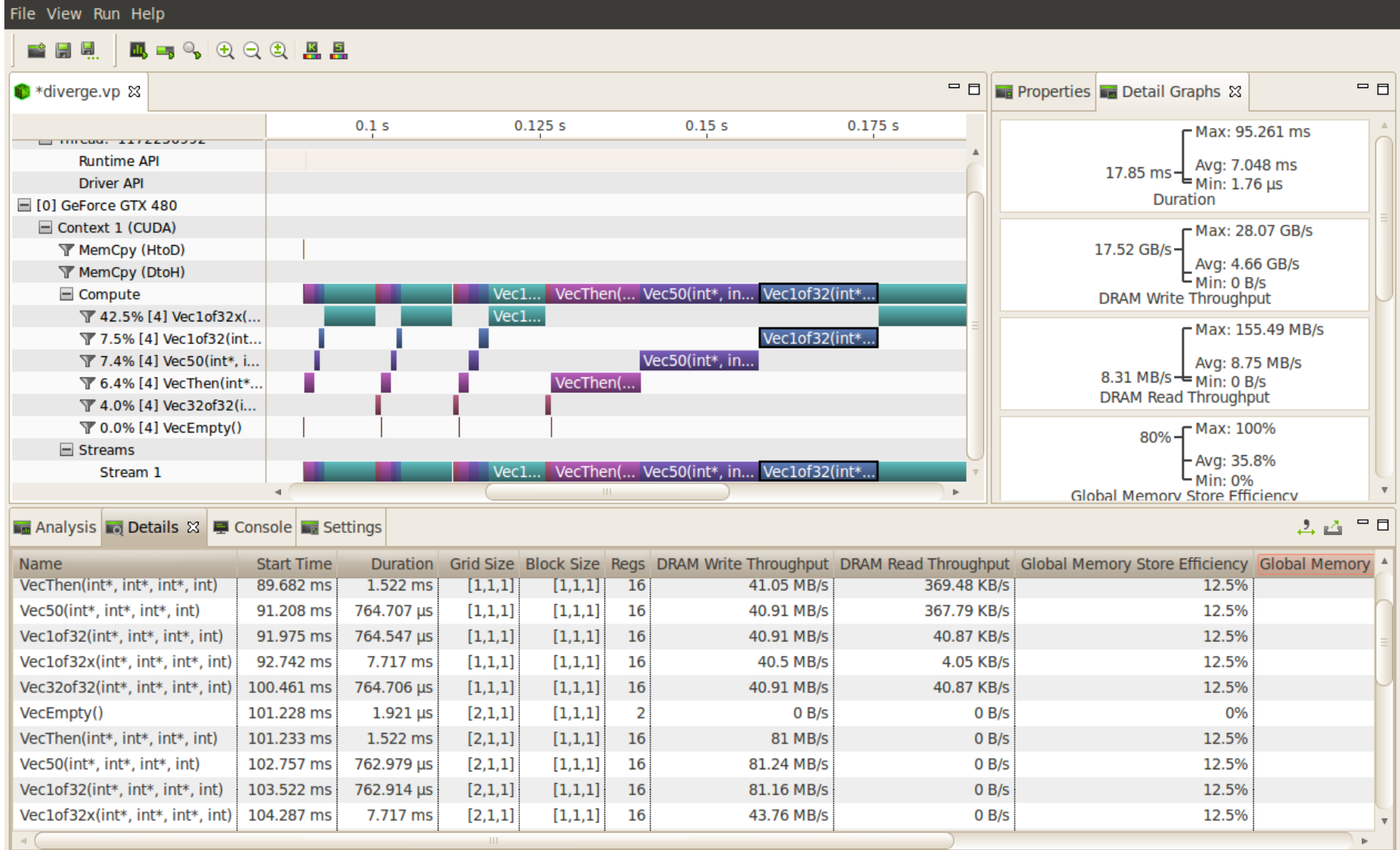
# Runtime Dynamic Code Assembly

- Relative low overhead for the OpenCL compiler vs. typical simulation run.
- OpenCL scheme inherently supports runtime compilation.
- WARPM dynamically gathers and stitches together source code based on the user's simulation parameters.
  - Numerical method, geometry, equation system

# OpenCL Performance Assessment

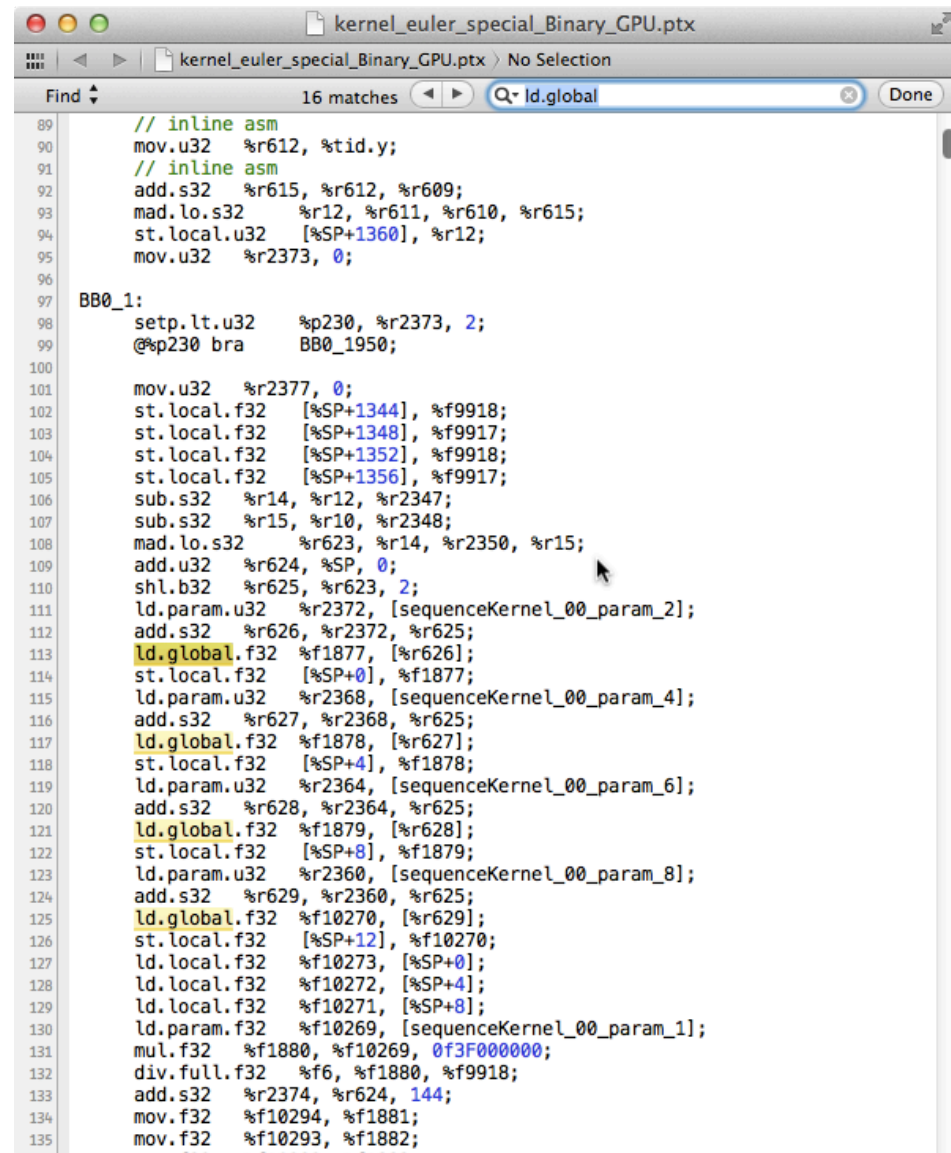
- NVIDIA Parallel Profiler
- Review the intermediate or assembly OpenCL compiler output.
- TRACE timing with tools such as TAU

# OpenCL Performance Assessment



# OpenCL Performance Assessment

- NVIDIA Parallel Profiler
- Review the intermediate or assembly OpenCL compiler output.
- TRACE timing with tools such as TAU



The screenshot shows a window titled 'kernel\_euler\_special\_Binary\_GPU.ptx'. The search bar at the top contains 'ld.global' and indicates '16 matches'. The assembly code is displayed with line numbers on the left. The code includes several instructions, with 'ld.global.f32' instructions highlighted in yellow. The instructions are as follows:

```
89 // inline asm
90 mov.u32 %r612, %tid.y;
91 // inline asm
92 add.s32 %r615, %r612, %r609;
93 mad.lo.s32 %r12, %r611, %r610, %r615;
94 st.local.u32 [%SP+1360], %r12;
95 mov.u32 %r2373, 0;
96
97 BB0_1:
98 setp.lt.u32 %p230, %r2373, 2;
99 @%p230 bra BB0_1950;
100
101 mov.u32 %r2377, 0;
102 st.local.f32 [%SP+1344], %f9918;
103 st.local.f32 [%SP+1348], %f9917;
104 st.local.f32 [%SP+1352], %f9918;
105 st.local.f32 [%SP+1356], %f9917;
106 sub.s32 %r14, %r12, %r2347;
107 sub.s32 %r15, %r10, %r2348;
108 mad.lo.s32 %r623, %r14, %r2350, %r15;
109 add.u32 %r624, %SP, 0;
110 shl.b32 %r625, %r623, 2;
111 ld.param.u32 %r2372, [sequenceKernel_00_param_2];
112 add.s32 %r626, %r2372, %r625;
113 ld.global.f32 %f1877, [%r626];
114 st.local.f32 [%SP+0], %f1877;
115 ld.param.u32 %r2368, [sequenceKernel_00_param_4];
116 add.s32 %r627, %r2368, %r625;
117 ld.global.f32 %f1878, [%r627];
118 st.local.f32 [%SP+4], %f1878;
119 ld.param.u32 %r2364, [sequenceKernel_00_param_6];
120 add.s32 %r628, %r2364, %r625;
121 ld.global.f32 %f1879, [%r628];
122 st.local.f32 [%SP+8], %f1879;
123 ld.param.u32 %r2360, [sequenceKernel_00_param_8];
124 add.s32 %r629, %r2360, %r625;
125 ld.global.f32 %f10270, [%r629];
126 st.local.f32 [%SP+12], %f10270;
127 ld.local.f32 %f10273, [%SP+0];
128 ld.local.f32 %f10272, [%SP+4];
129 ld.local.f32 %f10271, [%SP+8];
130 ld.param.f32 %f10269, [sequenceKernel_00_param_1];
131 mul.f32 %f1880, %f10269, 0f3F000000;
132 div.full.f32 %f6, %f1880, %f9918;
133 add.s32 %r2374, %r624, 144;
134 mov.f32 %f10294, %f1881;
135 mov.f32 %f10293, %f1882;
```

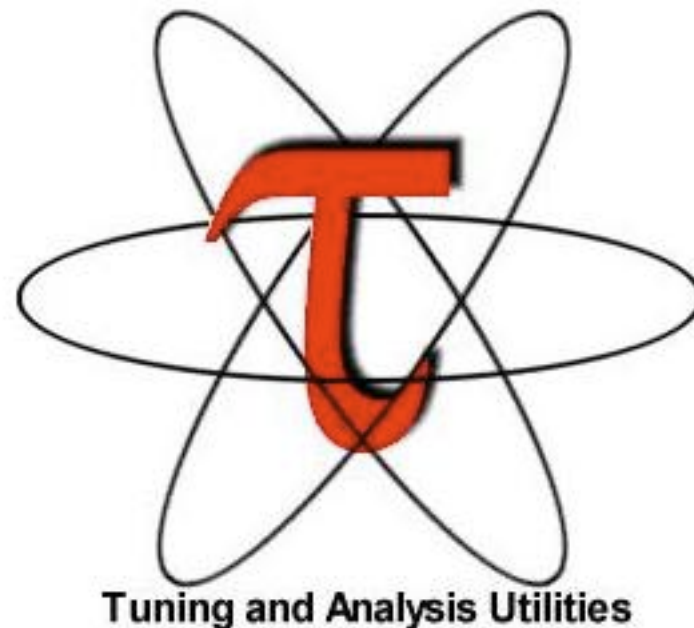
# OpenCL Performance Assessment

- NVIDIA Parallel Profiler
- Review the intermediate or assembly OpenCL compiler output.
- TRACE timing with tools such as TAU



# OpenCL Performance Assessment

- NVIDIA Parallel Profiler
- Review the intermediate or assembly OpenCL compiler output.
- TRACE timing with tools such as TAU



# Summary

- WARPM has proven to be an effective new scientific computing framework that is well situated for emerging computing architectures. Development will continue.
- WARPM provides a framework that can be readily utilized by users to solve their own computational models.
- Introduces three technologies
  - OpenCL kernel code that is dynamically assembled from code modules based on user simulation selections at run time.
  - Domain decomposition and boundary-first computation to overlap computation, memory movement, and MPI communication.
  - Task parallelism and dependency representation.

# Future Work

- Higher order FV and DG kernel implementations
- Kernel optimization efforts
- Performance analysis
- Study plasma physics

# Thank you

- All of this is possible with support from my Advisor Prof. Uri Shumlak
- And the DOE Computational Science Graduate Research Fellowship



# Backup Slides