GPU-acceleration of an Established Solar MHD code using OpenACC

Ronald M. Caplan, Jon A. Linker, Zoran Mikić, Cooper Downs, and Tibor Török

Slides available at: predsci.com/~caplanr
Outline

- Accelerated Computing
- OpenACC
- The MAS Code
- OpenACC Implementation
- Results
- Outlook
Accelerated Computing

An accelerator is a discrete piece of hardware designed for massively parallel computations.

Many brands/types of accelerators, here we focus on NVIDIA GPUs.

Why use accelerators?

1) Performance (FLOP/s and Memory Bandwidth)
2) Compact Performance
3) Saves Energy
4) Saves Money
Who uses accelerators?

... consists of 4,608 compute servers, each containing two 22-core IBM Power9 processors and six NVIDIA Tesla V100 GPU accelerators ...

ASTRONUM 2018
• Tues.  9:40 AM  M. Zingale
• Tues.  3:30 PM  M. Zhang
• Wed.  8:25 AM  N. Pogorelov - MS-FLUKSS
• Thurs. 1:55 PM  P. Woodward - PPMStar
Why *not* use accelerators?

- Not all algorithms suitable
- Hard to program
- Originally, only option was language extension APIs
  - CUDA (NVIDIA-specific)
  - OpenCL (more general)
- This involves rewriting large sections of code and maintaining at least two code bases.

```c
__global__ void saxpy(int N, float a,
                      float * restrict x,
                      float * restrict y){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < N) y[i] = a*x[i] + y[i];
}
...
const int BLOCK_SIZE=2048;
float *d_x,*d_y;
dim3 dimBlock(BLOCK_SIZE);
dim3 dimGrid((int)ceil((N+0.0)/dimBlock.x));
cudaMalloc((void **) &d_x, sizeof(float)*N);
cudaMalloc((void **) &d_y, sizeof(float)*N);
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);
saxpy<<<dimGrid,dimBlock>>>(N, a, d_x, d_y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
cudaFree(d_x);
cudaFree(d_y);
```

- Portability and longevity risk
  - What if GPUs go away?
OpenACC

More Science. Less Programming

- Directive-based API, began as off-shoot of OpenMP
- Uniform source code (no branches!)
- Low-risk (can compile to CPU as before)
- Vendor-independent (PGI, CRAY, GNU, OMNI, SunWay)
- Multiple Target Architectures (GPU, Multicore x86, FPGA, SunWay)
- Designed for rapid development, especially for pre-existing codes
- Used by >90% of GPU Industry codes run on Titan at ORNL

C: #pragma acc
FORTRAN: !$acc

BOOSTING INDUSTRY WITH OPENACC

Industrial users benefit from codes accelerated by directive-based programming standard

Over 100 apps* using OpenACC

- ANSYS Fluent
- Gaussian
- VASP
- LSDalton
- MPAS
- Gamera
- Numeca

* Applications in production and development
Example:
Accelerating SAXPY

```c
for (i=0; i<N; i++)
    y[i] = a*x[i] + y[i];
```

```c
#pragma acc enter data copyin(x,y)
#pragma acc parallel present(x,y)
{
    #pragma acc loop gang vector(32)
    for (i=0; i<N; i++)
        y[i] = a*x[i] + y[i];

    #pragma acc update_self(y)
    #pragma acc exit data delete(x,y)
}
```

```c
_global__ void saxpy(int N, float a,
                     float * restrict x,
                     float * restrict y){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < N) y[i] = a*x[i] + y[i];
}
```

```c
const int BLOCK_SIZE=2048;
float *d_x,*d_y;
dim3 dimBlock(BLOCK_SIZE);
dim3 dimGrid((int)ceil((N+0.0)/dimBlock.x));
...cudaMalloc( (void **) &d_x, sizeof(float)*N);
cudaMalloc( (void **) &d_y, sizeof(float)*N);
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);
saxpy<<<dimGrid,dimBlock>>>(N, 2.0, d_x, d_y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
cudaFree(d_x);
cudaFree(d_y);
```

```c
#pragma acc kernels
for (i=0; i<N; i++)
    y[i] = a*x[i] + y[i];
```
Established MHD code with over 15 years of development used extensively in solar physics research.

Written in FORTRAN 90 (~50,000 lines), parallelized with MPI.

Available for use at the Community Coordinated Modeling Center (CCMC).

Predicted Corona of the August 21st, 2017 Total Solar Eclipse.

Simulation of the Feb. 13th, 2009 CME.
MAS: ‘Full’ MHD Model Equations

\[ \frac{\partial \mathbf{A}}{\partial t} = \mathbf{v} \times (\nabla \times \mathbf{A}) - \frac{c^2 \eta}{4 \pi} \nabla \times \nabla \times \mathbf{A} \]

\[ \frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{v}) \]

\[ \frac{\partial T}{\partial t} = -\nabla \cdot (T \mathbf{v}) - (\gamma - 2) (T \nabla \cdot \mathbf{v}) + \frac{\gamma}{2k} \left( \frac{m_p}{\rho} \right) \left[ -\nabla \cdot (\mathbf{q}_1 + \mathbf{q}_2) \right] - \frac{\rho^2}{m_p^2} Q(T) + H \]

\[ \mathbf{q}_1 = -f(r) \beta_{\text{tor}}(T) \kappa_0 T^{5/2} \mathbf{b} \cdot \nabla T \]

\[ \mathbf{q}_2 = (1 - f(r)) \frac{k}{(\gamma - 1)} \frac{\rho}{m_p} T \mathbf{v} \mathbf{b} \]

\[ \frac{\partial \varepsilon_{\pm}}{\partial t} = -\nabla \cdot \left( \varepsilon_{\pm} [\mathbf{v} \pm \mathbf{v}_A] \right) - \frac{\varepsilon_{\pm}}{2} \nabla \cdot \mathbf{v} \]

\[ \frac{\partial \mathbf{v}}{\partial t} = -\nabla \cdot \mathbf{v} \mathbf{v} + \frac{1}{\rho} \left( \frac{1}{c} \mathbf{J} \times \mathbf{B} - \nabla p - \nabla \left( \frac{\varepsilon_+ + \varepsilon_-}{2} \right) + \rho \mathbf{g} \right) + \frac{1}{\rho} \nabla \cdot (\nu \rho \nabla \mathbf{v}) + \frac{1}{\rho} \nabla \cdot \left( S \rho \nabla \frac{\partial \mathbf{v}}{\partial t} \right) \]

\[ \frac{\partial z_{\pm}}{\partial t} = - (\mathbf{v} \pm \mathbf{v}_A) \cdot \nabla z_{\pm} - \frac{z_{\pm}}{2 \lambda_{\perp}} \left( \frac{z_{\perp}}{2} \right) + \frac{z_{\perp}}{4} (\mathbf{v} \pm \mathbf{v}_A) \cdot \nabla (\ln \rho) + \frac{z_{\perp}}{2} (\mathbf{v} \pm \mathbf{v}_A) \cdot \nabla (\ln |\mathbf{v}_A|) \]
In the low corona outside of active regions, the plasma beta is very small (i.e. dynamics dominated by magnetic field).

This allows a simplified “zero-beta” model to be useful in many cases (e.g. modeling the initial configuration and onset dynamics of a CME eruption).

Since the core algorithms are the same as the full model, this makes an ideal target for our initial OpenACC implementation (stepping stone).
MAS: Algorithm Summary

- Finite difference on non-uniform spherical grid
- Parallelized with MPI
- Explicit and implicit time-stepping algorithms
- Implicit time-step (backward-Euler) solved with Preconditioned Conjugate Gradient
- Two communication-free preconditioners: **PC1** and **PC2**
- For ‘hard’ solves, **PC2** faster than **PC1** for ‘easy’ solves, **PC1** faster than **PC2**
Zero-Beta Unstable Flux Rope Eruption

Run information

Physical code time duration: **198 seconds**
Number of time-steps: **695**

$160 \times 267 \times 246 \sim 10.5$ million points

Spherical Domain with $r_{\text{max}} = 10 R_\odot$

Detailed run information

<table>
<thead>
<tr>
<th>N</th>
<th>$\Delta_{\text{min}}$</th>
<th>$\Delta_{\text{max}}$</th>
<th>max</th>
<th>$\Delta_{t+1}-\Delta_t / \Delta_{t+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>160</td>
<td>800 km</td>
<td>530000 km</td>
<td>9%</td>
</tr>
<tr>
<td>$\theta$</td>
<td>267</td>
<td>0.066°</td>
<td>9.45°</td>
<td>11%</td>
</tr>
<tr>
<td>$\phi$</td>
<td>246</td>
<td>0.067°</td>
<td>14.61°</td>
<td>10%</td>
</tr>
<tr>
<td>$t$</td>
<td>695</td>
<td>0.001 sec</td>
<td>0.17 sec</td>
<td>11%</td>
</tr>
</tbody>
</table>

PCG Solver Iterations per Time Step (mean)

<table>
<thead>
<tr>
<th>SI Predictor</th>
<th>SI Corrector</th>
<th>Viscosity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1</td>
<td>72</td>
<td>75</td>
</tr>
<tr>
<td>PC2</td>
<td>30 $\rightarrow$ 34</td>
<td>31 $\rightarrow$ 35</td>
</tr>
</tbody>
</table>
Profile code

- PCG over 90% of run-time
- Viscosity is hardest solve

Analyze algorithms for GPU-compatibility

- Most PCG steps and explicit time-stepping “vector-friendly”
- Preconditioners
  - **PC1**: directives only (**portable**)
  - **PC2**: cuSparse (**not portable**)

Test performance through “Proof-of-concepts”

- DIFFUSE: Explicit finite-difference
- POT3D: PCG+PC1/PC2

Based on results of POT3D, we only accelerate **PC1** in MAS
OpenACC Implementation: Examples

**CPU ↔ GPU Data transfers**
- allocate and initialize “y” ...
- !$acc enter data copyin (y)
- use “y” in OpenACC compute regions ...
- !$acc update self (y)
- CPU version of “y” updated for I/O, etc. ...
- !$acc exit data delete(y)

**Basic Loop**
```
!$acc parallel default(present)
!$acc loop
    do i=1,n
        y(i) = a*x(i) + y(i)
    enddo
!$acc end parallel
```

**Reductions**
```
!$acc kernels loop present(y)
!$acc& reduction(+:sum)
    do j=1,m
        sum = sum + y(j)
    enddo
```

**FORTRAN Array-syntex**
```
!$acc kernels default(present)
    y(:) = a*x(:) + y(:)
!$acc end kernels
```
OpenACC Implementation: Multi-GPU

Multiple GPUs with MPI

**MPI-2**
(assumes linear affinity)

```
call MPI_Comm_rank (MPI_COMM_WORLD, iproc, ierr)
ngpus_per_node = 4
igpu = MODULO(iproc, ngpus_per_node)
!$acc set device_num(igpu)
```

**MPI-3**
(code shown assumes #GPUs/node = #ranks/node)

```
call MPI_Comm_split_type (MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, & 0, MPI_INFO_NULL, comm_shared, ierr)
call MPI_Comm_size (comm_shared, nprocsh, ierr)
call MPI_Comm_rank (comm_shared, iprocsh, ierr)
igpu = MODULO(iprocsh, nprocsh)
!$acc set device_num(igpu)
```

Use GPU data directly with MPI calls (“CUDA-aware MPI”)

```
!$acc host_data use_device(y) if_present
call MPI_Allreduce (MPI_IN_PLACE, y, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD, ierr)
!$acc end host_data
```
OpenACC Implementation: Effort Summary

Factors to consider:

- Optional CPU code simplifications
- Some CPU changes are temporary compiler bug work-arounds, or waiting for future OpenACC features
- Full code not accelerated (zero-beta only!)

Details:

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total lines in original code</td>
<td>52,600</td>
</tr>
<tr>
<td>Total lines in accelerated code</td>
<td>55,460</td>
</tr>
<tr>
<td>Total $!acc/$!acc&amp; lines added</td>
<td>776 (1.5%)</td>
</tr>
<tr>
<td>Total modified lines</td>
<td>2,451 (4.7%)</td>
</tr>
</tbody>
</table>

Single portable source for GPU and CPU!
OpenACC Implementation: Difficulties

Difficulties...

Compiler Issues
- Documentation lag
- Implementation lag
- Bugs

System issues
- Compiler licenses/updates
- Library versions and setup
- Hardware setups

!$acc cache(a;y(i-1:i+1))

> I'm sorry, I'm afraid I can't do that... yet
“Time-to-solution”
Includes I/O, comm, setup, etc. (Queue times excluded, but important!)

- We use best available compiler, compiler version, instruction sets, library versions, and *algorithm* for each hardware

**Why is this fair?**

We’re not benchmarking hardware

Want to test the maximum “effective” performance on each system for solving our problem, using our code
# Hardware and Environments

## NASA NAS Pleiades & Electra vs Local Workstation vs Local Desktop

<table>
<thead>
<tr>
<th></th>
<th>NASA NAS Pleiades &amp; Electra</th>
<th>Local Workstation</th>
<th>Local Desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Intel 2018.0.128</td>
<td>GNU 5.4.0</td>
<td></td>
</tr>
<tr>
<td>MPI Library</td>
<td>SGI MPT 2.15r20</td>
<td>OpenMPI 1.10.2</td>
<td></td>
</tr>
<tr>
<td>Family</td>
<td>Sandy Bridge</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ivy Bridge</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Haswell</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Broadwell</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Skylake</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Set</td>
<td>AVX</td>
<td>AVX2</td>
<td>AVX2</td>
</tr>
<tr>
<td>Model</td>
<td>E5-2670</td>
<td>E5-2680v3</td>
<td>Gold 6148</td>
</tr>
<tr>
<td></td>
<td>E5-2680v2</td>
<td>E5-2680v4</td>
<td></td>
</tr>
<tr>
<td>Clock Rate</td>
<td>2.6 GHz</td>
<td>2.5 GHz</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td></td>
<td>2.8 GHz</td>
<td>2.4 GHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.5 GHz</td>
<td>2.4 GHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4 GHz</td>
<td>2.5 GHz</td>
<td>3.6 GHz</td>
</tr>
<tr>
<td>#Sockets x #Cores</td>
<td>2x8</td>
<td>2x10</td>
<td>2x20</td>
</tr>
<tr>
<td></td>
<td>2x12</td>
<td>2x14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2x20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Mem Bandwidth</td>
<td>51.2 GB/s</td>
<td>59.7 GB/s</td>
<td>68 GB/s</td>
</tr>
<tr>
<td></td>
<td>68 GB/s</td>
<td>76.8 GB/s</td>
<td>128 GB/s</td>
</tr>
<tr>
<td></td>
<td>128 GB/s</td>
<td>68 GB/s</td>
<td>76.8 GB/s</td>
</tr>
</tbody>
</table>

## NVIDIA PSG vs SDSC Comet vs Local Desktop

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA PSG</th>
<th>SDSC Comet</th>
<th>Local Desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>PGI 18.3</td>
<td>PGI 18.4</td>
<td></td>
</tr>
<tr>
<td>MPI Library</td>
<td>OpenMPI 1.10.7</td>
<td>OpenMPI 2.1.2</td>
<td></td>
</tr>
<tr>
<td>CUDA Library</td>
<td>CUDA 9.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver Version</td>
<td>396.26</td>
<td>367.48</td>
<td>396.26</td>
</tr>
<tr>
<td># GPUs x Model</td>
<td>4xV100</td>
<td>4xP100</td>
<td>1xTitanXP</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>1.38 GHz</td>
<td>1.33 GHz</td>
<td>1.58 GHz</td>
</tr>
<tr>
<td># CUDA DP Cores/GPU</td>
<td>2560</td>
<td>1792</td>
<td>120</td>
</tr>
<tr>
<td>Mem Bandwidth/GPU</td>
<td>900 GB/s</td>
<td>732 GB/s</td>
<td>547.6 GB/s</td>
</tr>
</tbody>
</table>

## Compiler Flags:

- **Intel (CPU):** `-O3 -heap-arrays`
  `-fp-model precise`
  `-xCORE_AVX`#

- **GNU (CPU):** `-O3 -mtune=native`

- **PGI (GPU):** `-O3`
  `-ta=tesla:cuda9.1,cc`##
Timing Results

4x PCIe GPUs per node

RDMA
## Timing Results “In-house” Single Server

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Wall Clock (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon E5-2680v3 (2x12-core)</td>
<td>5.12</td>
</tr>
<tr>
<td>Xeon Gold 6148 (2x20-core)</td>
<td>6.88</td>
</tr>
<tr>
<td>P100 (1xGPU)</td>
<td>3.08</td>
</tr>
<tr>
<td>V100</td>
<td>3.58</td>
</tr>
<tr>
<td>P100 (2xGPU)</td>
<td>2.27</td>
</tr>
<tr>
<td>V100</td>
<td>1.88</td>
</tr>
<tr>
<td>P100 (4xGPU)</td>
<td>1.23</td>
</tr>
<tr>
<td>V100</td>
<td>0.92</td>
</tr>
<tr>
<td>P100 (8xGPU)</td>
<td>0.63</td>
</tr>
<tr>
<td>V100</td>
<td>0.62</td>
</tr>
</tbody>
</table>

**CPU Models:**
- Blue: CPU (PC1)
- Light Blue: CPU (PC2)

**GPU Models:**
- Green: GPU P100 (PC1)
- Orange: GPU V100 (PC1)
Timing Results “In-house” Single Desktop

Wall Clock: (est) ~1 hour
<1.5x Cost
>10x Speed

- Xeon E5-1650v4 (1x6-cores @3.6GHz) ~$3000
- Xeon E5-2680v3 (2x12-cores @2.5GHz) ~$7000
- GeForce TitanXP (1xGPU Pascal) +$1200

~$9000

Intel Broadwell
Intel Haswell

CPU (PC1)
CPU (PC2)
GPU (PC1)
Alternative Algorithms: Super Time-Stepping

- Want vectorizable PC as good as **PC2** in reducing iterations
- Geometric/algebraic multigrid attractive choice but requires massive code changes
- At ASTRONUM 2016 we tested RKL2 Super Time-Stepping (**STS**) (Meyers et al 2014) in MAS as an alternative to **PCG** for viscosity
- Performance of the **STS** method was great, but had accuracy issues
- Since the **STS** algorithm is highly vectorizable, it's worth testing an **OpenACC** implementation for the current problem (where viscosity is most time-consuming)
Alternative Algorithms: Super Time-Stepping

Viscosity: PCG+PC2

Viscosity: STS

Magnetic Energy

Kinetic Energy

\[ |J \times B| \]
Alternative Algorithms: Super Time-Stepping

CPU: STS $\approx$ PC2

GPU: STS $<$ PC1

- CPU STS exhibits better scaling, but similar run times to PC2
- GPU STS $\sim$ twice as fast as PC1, but similar scaling
<table>
<thead>
<tr>
<th></th>
<th>1x</th>
<th>1.6x</th>
<th>2.9x</th>
<th>5.3x</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PC1</strong></td>
<td><img src="image1" alt="V100" /></td>
<td><img src="image2" alt="P100" /></td>
<td><img src="image3" alt="Intel Skylake" /></td>
<td><img src="image4" alt="Intel Haswell" /></td>
</tr>
<tr>
<td>(2.3 hours)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CPU: PC2</strong></td>
<td><img src="image1" alt="V100" /></td>
<td><img src="image2" alt="P100" /></td>
<td><img src="image3" alt="Intel Skylake" /></td>
<td><img src="image4" alt="Intel Haswell" /></td>
</tr>
<tr>
<td><strong>GPU: PC1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2.3 hours)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>STS\textsubscript{visc}</strong></td>
<td><img src="image1" alt="V100" /></td>
<td><img src="image2" alt="P100" /></td>
<td><img src="image3" alt="Intel Skylake" /></td>
<td><img src="image4" alt="Intel Haswell" /></td>
</tr>
<tr>
<td>(1.1 hours)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can fit 4 of these in one desktop (~20 min)!
Summary and Outlook

For this run (representative of many similar cases), we can move from HPC cluster to “in-house”

Future improvements

- Vectorizable Preconditioners
- **PC2** with single-precision
- Make **STS** method accuracy-robust

Next steps

- Heliospheric runs (**PC1** faster than **PC2** on **CPU**)!
- Thermodynamic (coronal) runs (on GPU-cluster like Summit)
Questions?

This work was supported by
- NSF’s Frontiers in Earth System Dynamics
- NASA’s Living with a Star program
- Air Force Office of Scientific Research

Computer allocations provided by
- NASA NAS (Pleiades/Electra)
- NVIDIA Cooperation (PSG)
- XSEDE/SDSC (Comet)