NREL
NATIONAL RENEWABLE ENERGY LABORATORY

Accelerating Kernels from WRF on GPUs

John Michalakes, NREL

Manish Vachharajani, University of Colorado

John Linford, Virginia Tech

Adrian Sandu, Virginia Tech

PEEPS Workshop, June 22, 2010

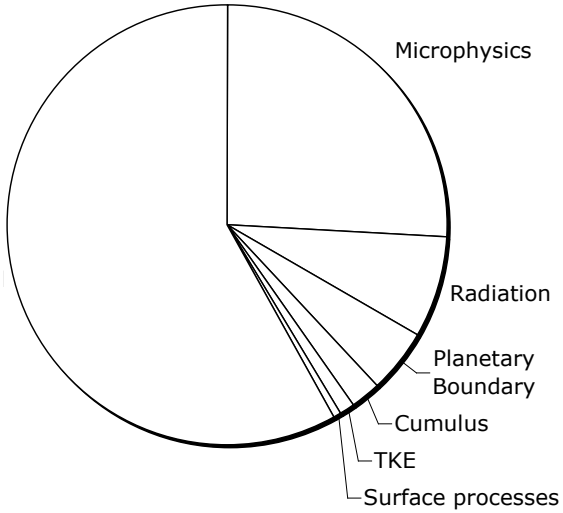
WRF Overview

- Large collaborative effort to develop next-generation community non-hydrostatic model
  - 4000+ registered users
  - Applications
    - Numerical Weather Prediction
    - High resolution climate
    - Air quality research/prediction
    - Wildfire
    - Atmospheric Research
- Software designed for HPC
  - Ported to and in use on virtually all types of system in the Top500
  - 2007 Gordon Bell finalist
- Why accelerators?
  - Cost performance
  - Need for strong scaling

http://www.wrf-model.org

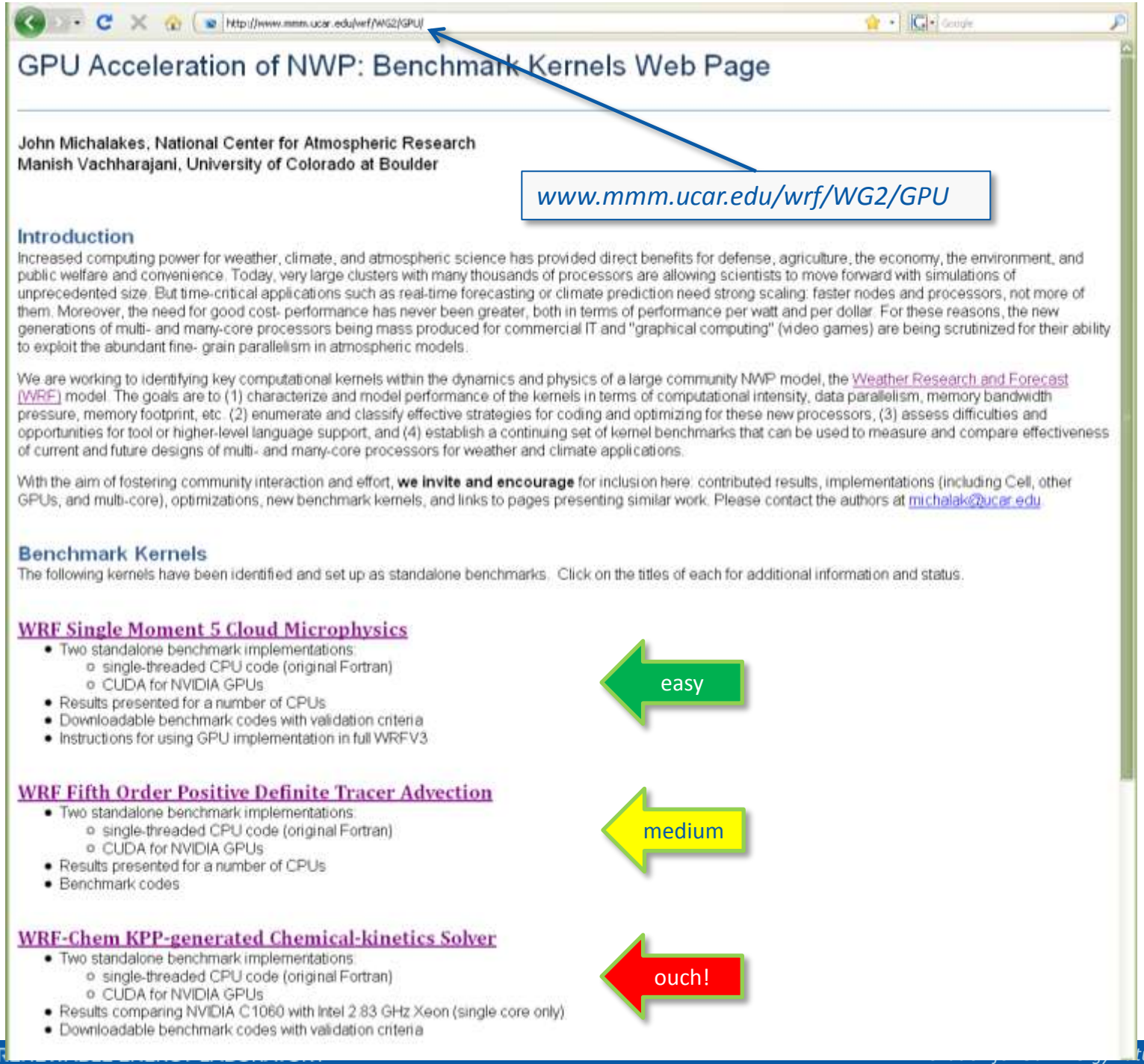
WRF Overview

- Software
  - ~0.5 million lines mostly Fortran
  - MPI and OpenMP
  - All single (32-bit) precision
- Dynamics
  - CFD over regular Cartesian 3D grid
  - Explicit finite-difference
  - 2D decomposition in X and Y
- Physics
  - Computes forcing terms as updates to tendencies of state variables
  - Column-wise, perfectly parallel in horizontal dimensions
  - ¼ of total run time is microphysics



microphysics	26%
other physics	20%
dynamics	44%
other	10%

Percentages of total run time (single processor profile)


http://www.mmm.ucar.edu/wrf/WG2/GPU/
GPU Acceleration of NWP: Benchmark Kernels Web Page
John Michalakes, National Center for Atmospheric Research
Manish Vachharajani, University of Colorado at Boulder
www.mmm.ucar.edu/wrf/WG2/GPU
Introduction
Increased computing power for weather, climate, and atmospheric science has provided direct benefits for defense, agriculture, the economy, the environment, and public welfare and convenience. Today, very large clusters with many thousands of processors are allowing scientists to move forward with simulations of unprecedented size. But time-critical applications such as real-time forecasting or climate prediction need strong scaling: faster nodes and processors, not more of them. Moreover, the need for good cost- performance has never been greater, both in terms of performance per watt and per dollar. For these reasons, the new generations of multi- and many-core processors being mass produced for commercial IT and "graphical computing" (video games) are being scrutinized for their ability to exploit the abundant fine- grain parallelism in atmospheric models.
We are working to identifying key computational kernels within the dynamics and physics of a large community NWP model, the Weather Research and Forecast (WRF) model. The goals are to (1) characterize and model performance of the kernels in terms of computational intensity, data parallelism, memory bandwidth pressure, memory footprint, etc. (2) enumerate and classify effective strategies for coding and optimizing for these new processors, (3) assess difficulties and opportunities for tool or higher-level language support, and (4) establish a continuing set of kernel benchmarks that can be used to measure and compare effectiveness of current and future designs of multi- and many-core processors for weather and climate applications.
With the aim of fostering community interaction and effort, we invite and encourage for inclusion here: contributed results, implementations (including Cell, other GPUs, and multi-core), optimizations, new benchmark kernels, and links to pages presenting similar work. Please contact the authors at michalak@ucar.edu
Benchmark Kernels
The following kernels have been identified and set up as standalone benchmarks. Click on the titles of each for additional information and status.
WRF Single Moment 5 Cloud Microphysics
Two standalone benchmark implementations
single-threaded CPU code (original Fortran)
CUDA for NVIDIA GPUs
Results presented for a number of CPUs
Downloadable benchmark codes with validation criteria
Instructions for using GPU implementation in full WRFV3
easy
WRF Fifth Order Positive Definite Tracer Advection
Two standalone benchmark implementations
single-threaded CPU code (original Fortran)
CUDA for NVIDIA GPUs
Results presented for a number of CPUs
Benchmark codes
medium
WRF-Chem KPP-generated Chemical-kinetics Solver
Two standalone benchmark implementations
single-threaded CPU code (original Fortran)
CUDA for NVIDIA GPUs
Results comparing NVIDIA C1060 with Intel 2.83 GHz Xeon (single core only)
Downloadable benchmark codes with validation criteria
ouch!

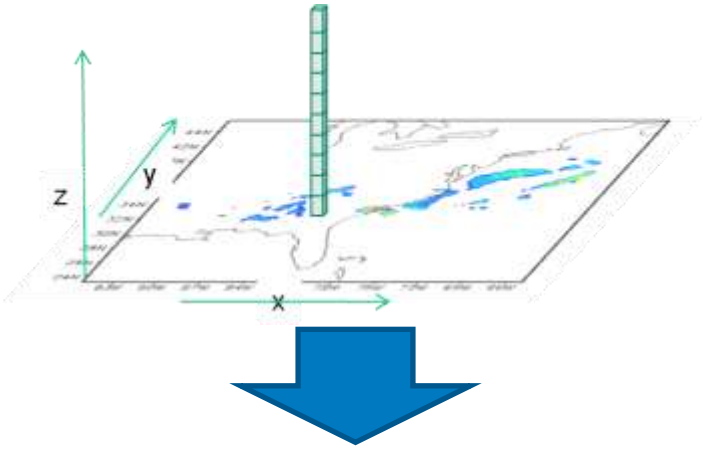
Kernel 1: Microphysics

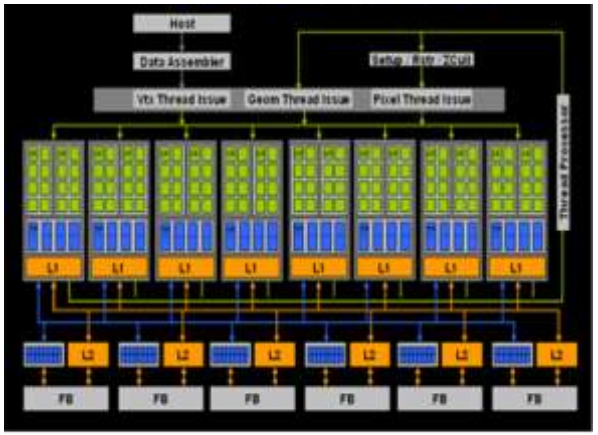
- WRF Single Moment 5-Tracer (WSM5)* scheme
- Represents condensation, precipitation, and thermodynamic effects of latent heat release
- Operates independently up each column of 3D WRF domain
- Large memory footprint: 40 32-bit floats per cell
- Expensive:
  - Called every time step
  - 2400 floating point multiply-equiv. per cell per invocation

*Hong, S., J. Dudhia, and S. Chen (2004). Monthly Weather Review, 132(1):103-120.

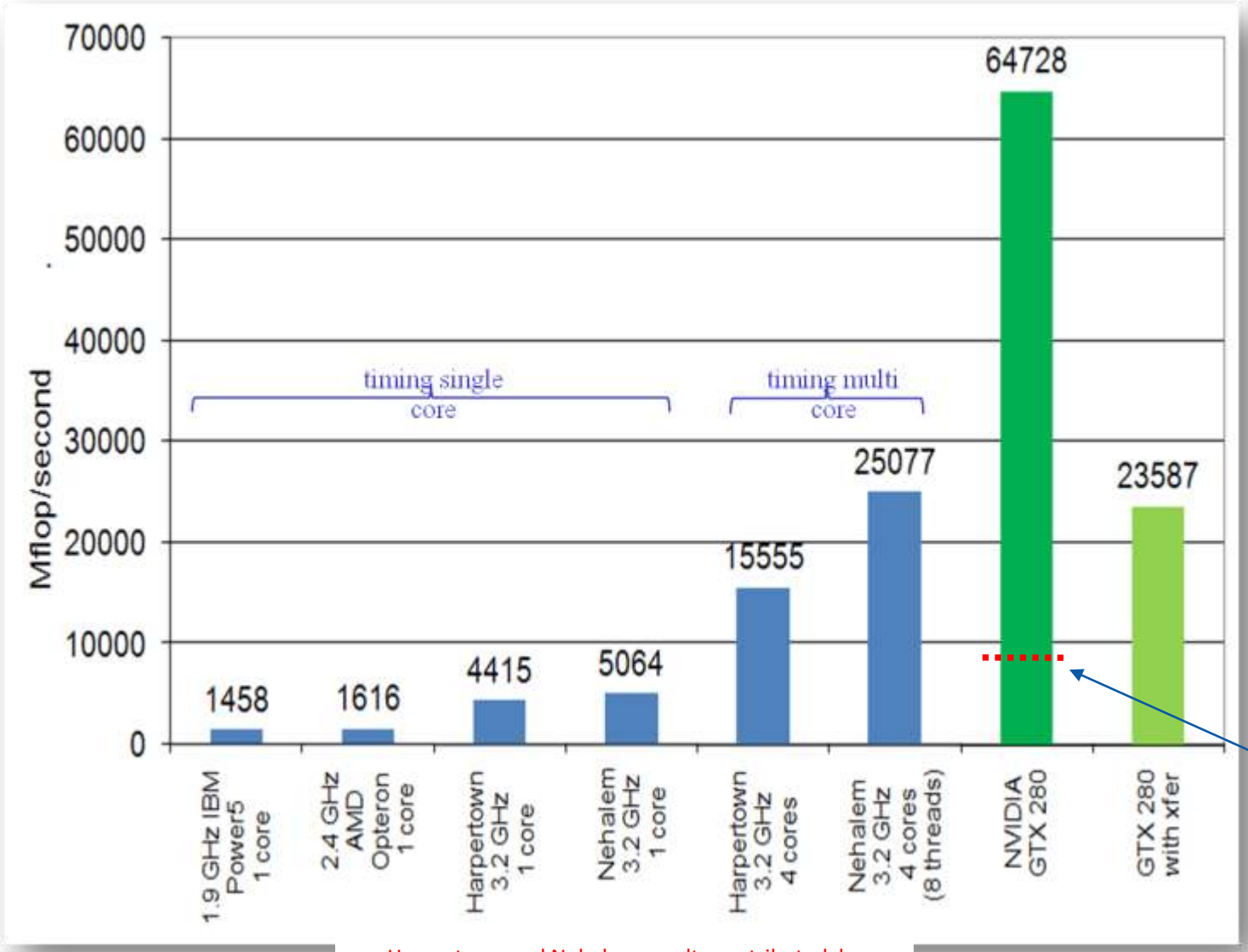
Kernel 1: Microphysics

- Manual conversion, writing 15-hundred line Fortran90 module into CUDA C
- Remove outer loops over i, j horizontal dimensions, keep only vertical k loops
- Each resulting column assigned to a thread
- Benchmark workload: Standard WRF test case (Eastern U.S. Storm, Jan. 24, 2000)





Kernel 1: WSM5 Microphysics



Harpertown and Nehalem results contributed by Roman Dubtsov, Intel

Kernel 1: WSM5 Microphysics

- WSM5 Microphysics adapted to NVIDIA's CUDA for GPU
  - 15-25% of WRF cost effectively removed along with load imbalance
  - CUDA version distributed with WRFV3
  - Users have seen 1.2-1.3x improvement

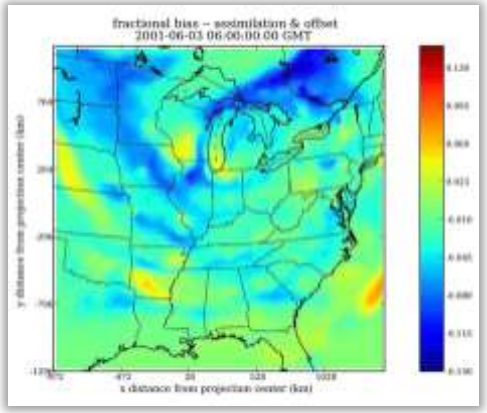
- PGI have acceleration directives show comparable speedups and overheads from transfer cost

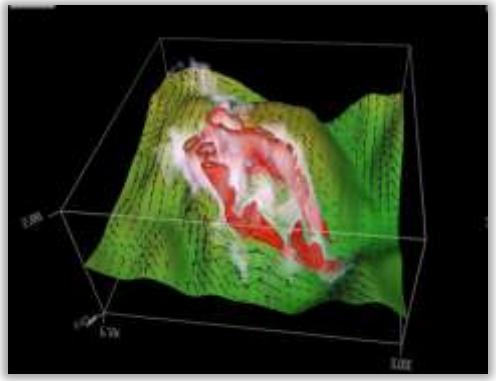
WRF CONUS 12km benchmark
Courtesy Brent Leback and Craig Toepfer, PGI

	Original PGI 9.0-4 Host	Accelerated PGI 9.0-4 Host+GPU	CUDA C w/PGI 9.0-4 Host+GPU
total seconds	1679.71	1413.48	1413.46
microphysics	*(276.72)*	*(29.79)*	*(26.35)*

Kernel 3: WRF-Chem*

- WRF model coupled to atmospheric chemistry for air quality research and air pollution forecasting





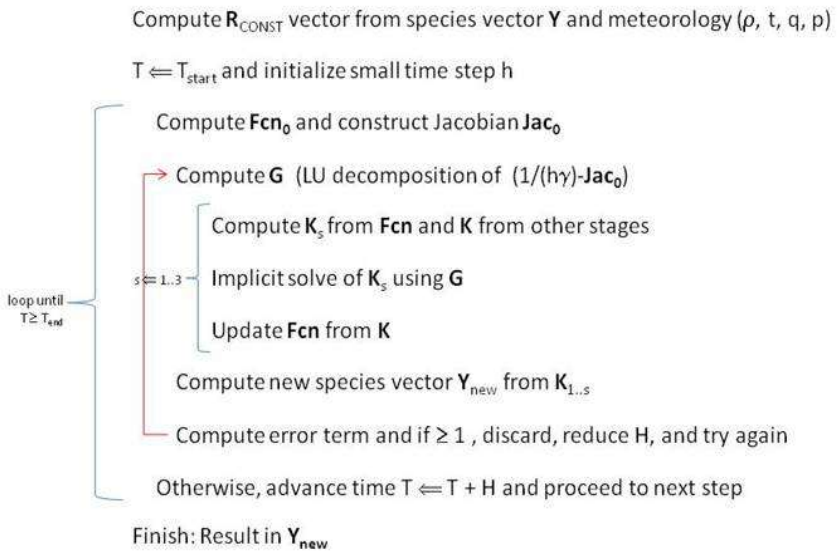
*Grell *et al.*, WRF Chem Version 3.0 User's Guide, http://ruc.fsl.noaa.gov/wrf/WG11

**Hairer E. and G. Wanner. *Solving ODEs II: Stiff and Differential-Algebraic Problems*, Springer 1996.

***Damian, *et al.* (2002). Computers & Chemical Engineering 26, 1567-1579.

Kernel 3: WRF-Chem*

- WRF model coupled to atmospheric chemistry for air quality research and air pollution forecasting
- RADM2-SORG chemical kinetics solver:
  - Time evolution of tens to hundreds of chemical species being produced and consumed at varying rates in networks of reactions
  - Rosenbrock** solver for stiff system of ODEs at each cell
  - Series of Newton iterations, each step of which is solved implicitly
  - Many times cost of core meteorology
    - WRF domain is very small: 160M floating point operations per time step
    - Chemistry on same domain increases cost 40x



- Y(NVAR) – input vector of 59 active species concentrations
- Temporaries Ynew(NVAR) , Yerr(NVAR), and K(NVAR*3)
- Fcn(NVAR) – dY_i / dt
- RCONST(NREACT) – array of 159 reaction rates.
- Jac0(LU_NONZERO), Ghimj(LU_NONZERO) store 659 non-zero entries of Jacobian
- Integer arrays for indexing sparse Jacobian matrix (stored in GPU constant memory)

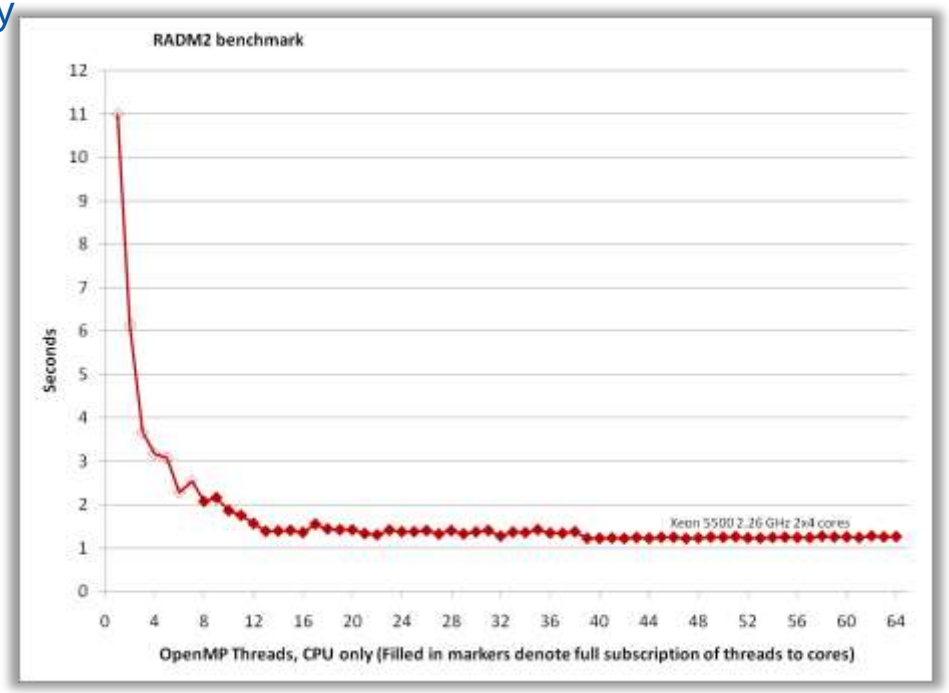
*Grell *et al.*, WRF Chem Version 3.0 User's Guide, http://ruc.fsl.noaa.gov/wrf/WG11

**Hairer E. and G. Wanner. *Solving ODEs II: Stiff and Differential-Algebraic Problems*, Springer 1996.

***Damian, *et al.* (2002). Computers & Chemical Engineering 26, 1567-1579.

Kernel 3: WRF-Chem*

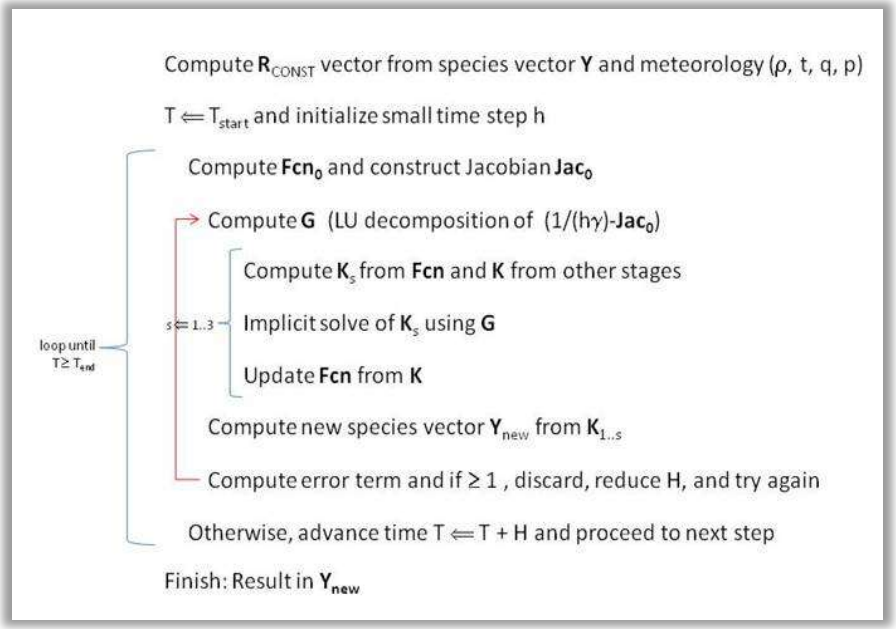
- WRF model coupled to atmospheric chemistry for air quality research and air pollution forecasting
- RADM2-SORG chemical kinetics solver:
  - Time evolution of tens to hundreds of chemical species being produced and consumed at varying rates in networks of reactions
  - Rosenbrock** solver for stiff system of ODEs at each cell
  - Series of Newton iterations, each step of which is solved implicitly
  - Many times cost of core meteorology
    - WRF domain is very small: 160M floating point operations per time step
    - Chemistry on same domain increases cost 40x
- Parallelism
  - The computation itself is completely serial
  - Independent computation at each cell
  - Seemingly ideal for massively threaded acceleration



Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 using CUDA (first attempt)

- Convert KPP generated Fortran to C
- Convert entire solver for one cell into CUDA
- Spawn kernel as one-thread-per-cell over domain
- Results:
  - Too much for CUDA compiler
  - Entire kernel constrained by most resource-intensive step
  - Disappointing performance



Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 using CUDA (first attempt)

- Convert KPP generated Fortran to C
- Convert entire solver for one cell into CUDA
- Spawn kernel as one-thread-per-cell over domain
- Results:
  - Too much for CUDA compiler
  - Entire kernel constrained by most resource-intensive step
  - Disappointing performance

```
Radm2sorg <<<gridDim, blockDim >>>( … )
```

Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 using CUDA (first attempt)

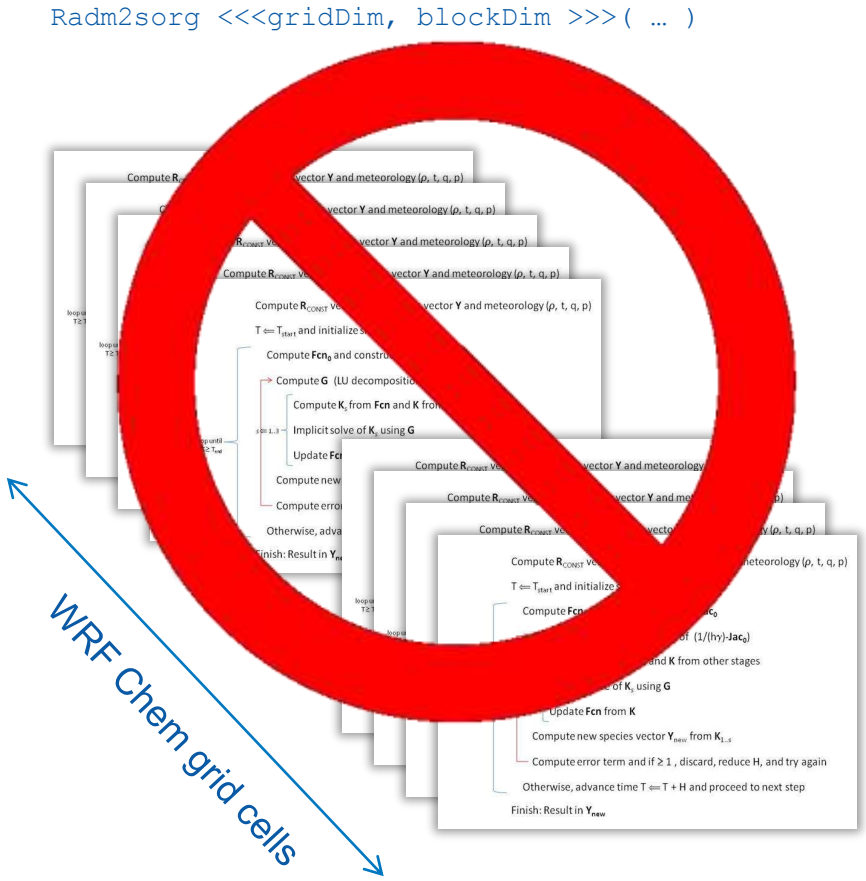
- Computation and storage at each grid cell per invocation:
  - 600K fp ops
  - 1M load/stores
  - 1800 dbl. prec. words
  - Array layout is cell-index outermost
- This means
  - Low computational intensity
  - Massive temporal working set
  - Outstrips shared memory and available registers per thread
- Result
  - Latency to GPU memory is severe bottleneck
  - Non-coalesced access to GPU memory is also a bandwidth limitation

```
Radm2sorg <<<gridDim, blockDim >>>( … )
```

Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 Improvements

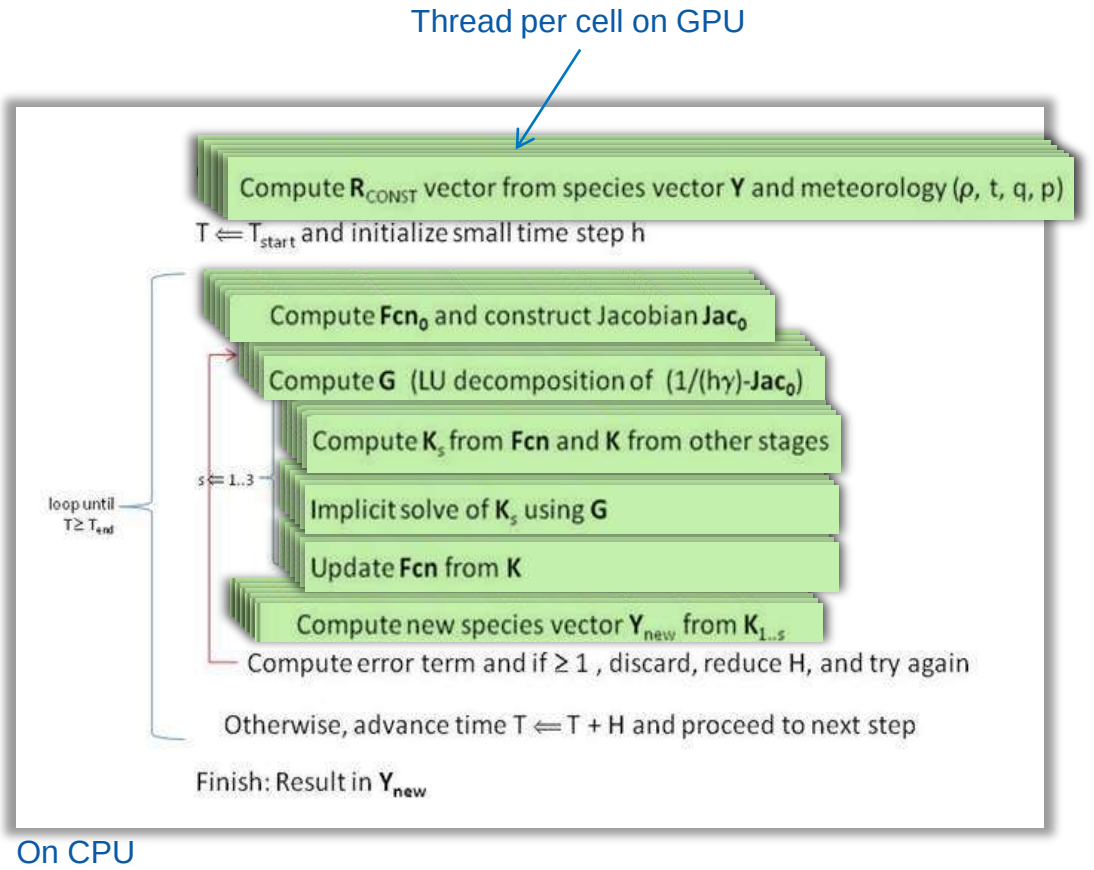
- Rewrite code to break up single RADM2 kernel into steps
  - Outer loop given back to CPU
  - Smaller footprint
  - Individual kernels can be invoked according to what's optimal for that step in terms of
    - Number of threads
    - Use of shared memory
  - No performance downside: kernel invocation latency is small
  - Most difficult in terms of effort



Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 Improvements

- Rewrite code to break up single RADM2 kernel into steps
  - Outer loop given back to CPU
  - Smaller footprint
  - Individual kernels can be invoked according to what's optimal for that step in terms of
    - Number of threads
    - Use of shared memory
  - No performance downside: kernel invocation latency is small
  - Involves a complete rewrite



Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 Improvements

- Store indirection vectors into sparse data structures in GPU constant memory (easy)

```
SUBROUTINE KppSolveIndirect( JVS, X )
! ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Sparse solve subroutine using indirect addressing
! ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  USE KPP_ROOT_Parameters
  USE KPP_ROOT_JacobianSP
      INTEGER i, j
      KPP_REAL JVS(KPP_LU_NONZERO), X(KPP_NVAR), sum
      DO i=1,NVAR
         DO j = LU_CROW(i), LU_DIAG(i)-1
             X(i) = X(i) - JVS(j)*X(LU_ICOL(j));
         END DO
      END DO
      DO i=NVAR,1,-1
        sum = X(i);
        DO j = LU_DIAG(i)+1, LU_CROW(i+1)-1
          sum = sum - JVS(j)*X(LU_ICOL(j));
        END DO
        X(i) = sum/JVS(LU_DIAG(i));
      END DO
END SUBROUTINE KppSolveIndirect
```

Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 Improvements

- Store indirection vectors into sparse data structures in GPU constant memory (easy)
- Unroll loops over sparse arrays
  - Exposes reuse to compiler to exploit 16K register file on each stream multiprocessor
  - More effective than putting datastructures in shared memory, even when they do fit
  - Free: KPP can do this automatically

rolled

```
SUBROUTINE KppSolveIndirect( JVS, X )
! ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Sparse solve subroutine using indirect addressing
! ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  USE KPP_ROOT_Parameters
  USE KPP_ROOT_JacobianSP
      INTEGER i, j
      KPP_REAL JVS(KPP_LU_NONZERO), X(KPP_NVAR), sum
      DO i=1,NVAR
         DO j = LU_CROW(i), LU_DIAG(i)-1
            X(i) = X(i) - JVS(j)*X(LU_ICOL(j));
         END DO
      END DO
      DO i=NVAR,1,-1
        sum = X(i);
        DO j = LU_DIAG(i)+1, LU_CROW(i+1)-1
          sum = sum - JVS(j)*X(LU_ICOL(j));
        END DO
        X(i) = sum/JVS(LU_DIAG(i));
      END DO
END SUBROUTINE KppSolveIndirect
```

KPP

unrolled

```
  . . .
X(36) = X(36)-JVS(188)*X(11)
X(37) = X(37)-JVS(195)*X(27)-JVS(196)*X(28)
X(38) = X(38)-JVS(205)*X(24)-JVS(206)*X(27)-JVS(207)*X(28)-JVS(208)*X(29)
X(39) = X(39)-JVS(217)*X(8)-JVS(218)*X(9)-JVS(219)*X(22)-JVS(220)*X(23)
X(40) = X(40)-JVS(230)*X(24)
X(41) = X(41)-JVS(239)*X(8)
X(42) = X(42)-JVS(246)*X(32)-JVS(247)*X(35)-JVS(248)*X(36)-JVS(249)*X(37)- . . .
X(43) = X(43)-JVS(268)*X(16)-JVS(269)*X(21)-JVS(270)*X(23)-JVS(271)*X(24)- . . .
        &-JVS(275)*X(31)-JVS(276)*X(32)-JVS(277)*X(35)-JVS(278)*X(36)- . . .
  . . .
```

thousands of lines of this

Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010

RADM2 Improvements

- Store indirection vectors into sparse data structures in GPU constant memory (easy)
- Unroll loops over sparse arrays
  - Exposes reuse to compiler to exploit 16K register file on each stream multiprocessor
  - More effective than putting datastructures in shared memory, even when they do fit
  - Free: KPP can do this automatically
- Reorder arrays so cell-index innermost to give 2x improvement in bandwidth through coalescing (somewhat easy using macros)

rolled

```
SUBROUTINE KppSolveIndirect( JVS, X )
! ---------------------------------------------
! Sparse solve subroutine using indirect addressing
! ---------------------------------------------
  USE KPP_ROOT_Parameters
  USE KPP_ROOT_JacobianSP
      INTEGER i, j
      KPP_REAL JVS(KPP_LU_NONZERO), X(KPP_NVAR), sum
      DO i=1,NVAR
         DO j = LU_CROW(i), LU_DIAG(i)-1
            X(i) = X(i) - JVS(j)*X(LU_ICOL(j));
         END DO
      END DO
      DO i=NVAR,1,-1
        sum = X(i);
        DO j = LU_DIAG(i)+1, LU_CROW(i+1)-1
          sum = sum - JVS(j)*X(LU_ICOL(j));
        END DO
        X(i) = sum/JVS(LU_DIAG(i));
      END DO
END SUBROUTINE KppSolveIndirect
```

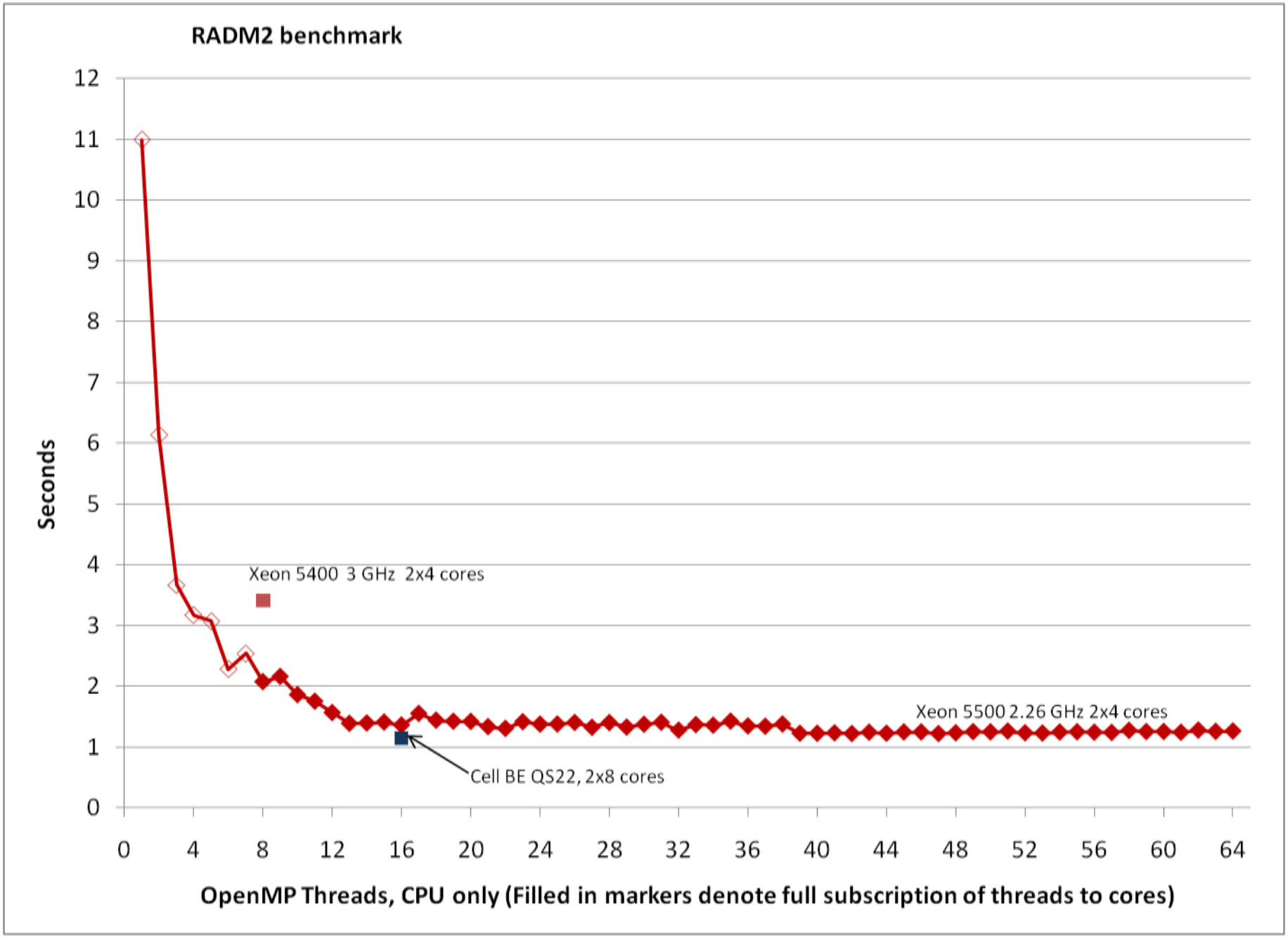
KPP

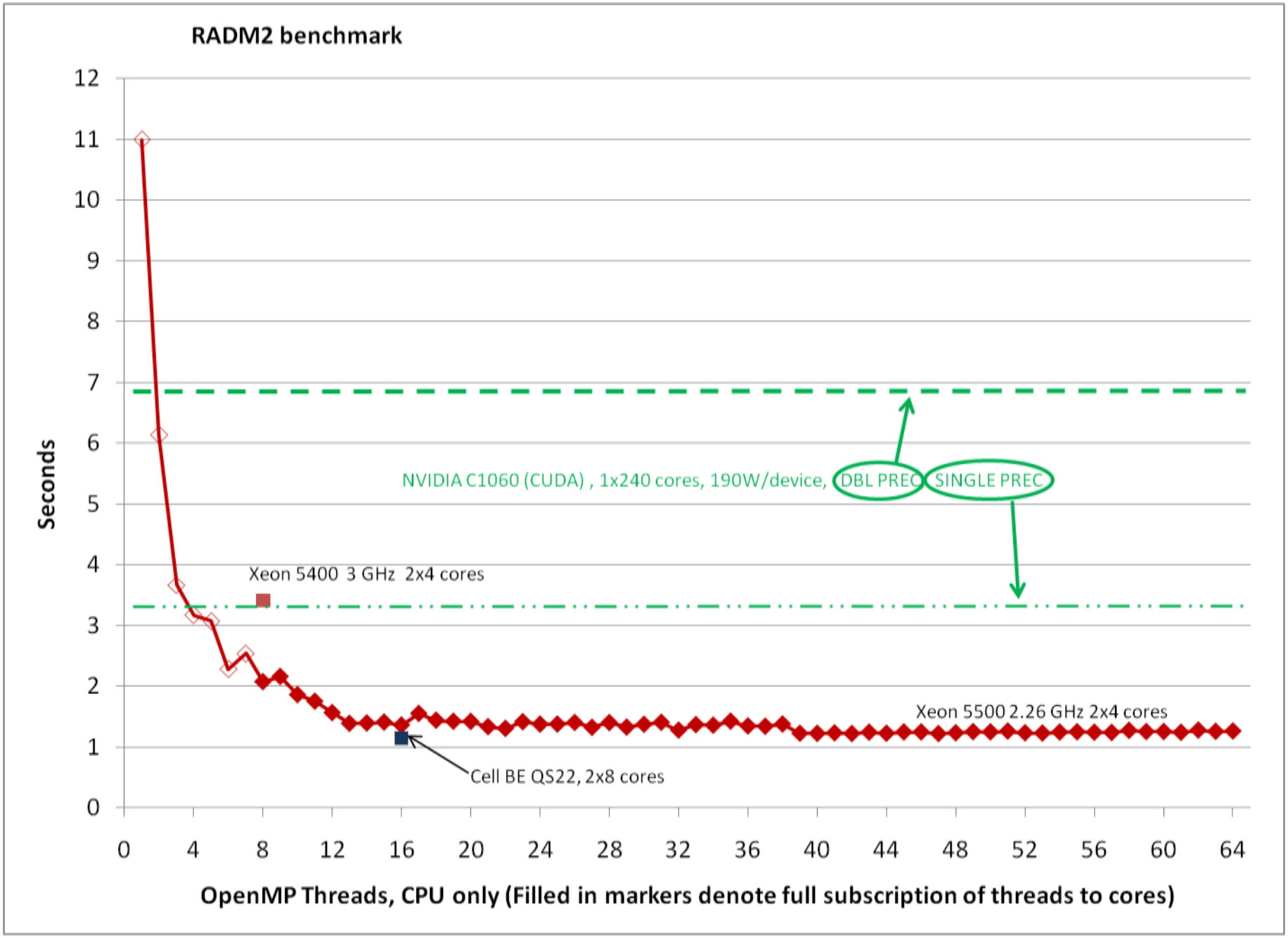
unrolled

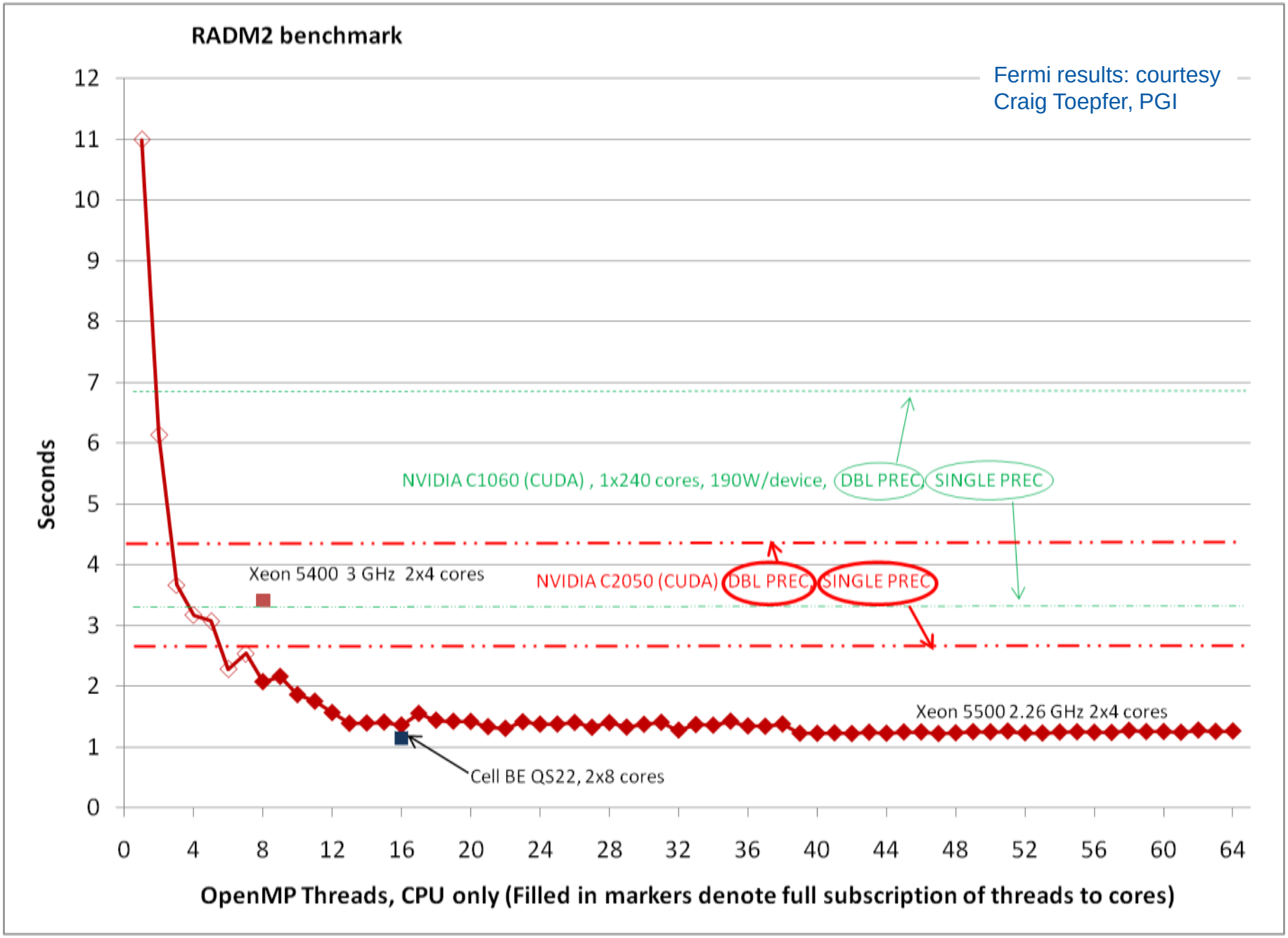
```
. . .
X(36) = X(36)-JVS(188)*X(11)
X(37) = X(37)-JVS(195)*X(27)-JVS(196)*X(28)
X(38) = X(38)-JVS(205)*X(24)-JVS(206)*X(27)-JVS(207)*X(28)-JVS(208)*X(29)
X(39) = X(39)-JVS(217)*X(8)-JVS(218)*X(9)-JVS(219)*X(22)-JVS(220)*X(23)
X(40) = X(40)-JVS(230)*X(24)
X(41) = X(41)-JVS(239)*X(8)
X(42) = X(42)-JVS(246)*X(32)-JVS(247)*X(35)-JVS(248)*X(36)-JVS(249)*X(37)- . . .
X(43) = X(43)-JVS(268)*X(16)-JVS(269)*X(21)-JVS(270)*X(23)-JVS(271)*X(24)- . . .
          &-JVS(275)*X(31)-JVS(276)*X(32)-JVS(277)*X(35)-JVS(278)*X(36)- . . .
. . .
```

thousands of lines of this

Linford, Michalakes, Vachharajani, Sandu. *Special Issue, High Performance Computing with Accelerators.* Trans. Parallel and Distributed systems. To appear. 2010


RADM2 benchmark
Seconds
12
11
10
9
8
7
6
5
4
3
2
1
0
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
Xeon 5400 3 GHz 2x4 cores
Xeon 5500 2.26 GHz 2x4 cores
Cell BE QS22, 2x8 cores
OpenMP Threads, CPU only (Filled in markers denote full subscription of threads to cores)


RADM2 benchmark
Seconds
12
11
10
9
8
7
6
5
4
3
2
1
0
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
NVIDIA C1060 (CUDA) , 1x240 cores, 190W/device,
DBL PREC
SINGLE PREC
Xeon 5400 3 GHz 2x4 cores
Xeon 5500 2.26 GHz 2x4 cores
Cell BE QS22, 2x8 cores
OpenMP Threads, CPU only (Filled in markers denote full subscription of threads to cores)


RADM2 benchmark
Fermi results: courtesy
Craig Toepfer, PGI
Seconds
12
11
10
9
8
7
6
5
4
3
2
1
0
NVIDIA C1060 (CUDA) , 1x240 cores, 190W/device,
DBL PREC
SINGLE PREC
Xeon 5400 3 GHz 2x4 cores
NVIDIA C2050 (CUDA)
DBL PREC
SINGLE PREC
Xeon 5500 2.26 GHz 2x4 cores
Cell BE QS22, 2x8 cores
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
OpenMP Threads, CPU only (Filled in markers denote full subscription of threads to cores)

Code transformations

- GPU device memory latency
  - Fusing loops to get rid of temporary arrays
  - Unrolling loops over sparse data structures to expose register reuse
  - Rewriting code to use shared-memory, if working set fits
  - ~~Pipelining tasks between cores on the GPU~~. Not possible.
- GPU device memory bandwidth
  - Array & loop index reordering to improve coalesced memory access
- Host-GPU transfer costs
  - Organizing host-GPU transfers to minimize movement
  - Asynchronous data transfers
  - Using pinned memory on host to speed up host-GPU transfers
  - Hand-coding to access array sections for MPI communications
- Misc.
  - Breaking up code into multiple kernel invocations

Some final thoughts on programming models

- What's good about GPU programming
  - Forces programmer to think in terms of simple tasks performed over large numbers of lightweight threads
  - We'll have to think that way for peta-/exascale-systems anyway
  - Programs converted to GPU often perform better on multi-core too
- What's bad about GPU programming
  - The memory hierarchy must be programmed explicitly
  - The co-processor model must also be programmed explicitly
  - Restructuring for performance is manual, costly, and blind.
  - Does the investment pay off in performance? Will the program be usable in 5 years?