

Hybrid Multi-core Programming for Exascale Computing

John Levesque

Cray CTO Office

Director of Cray's Supercomputing Center of Excellence

Electronic Structures Workshop

February 6, 2012

Key Challenges to Get to an Exascale

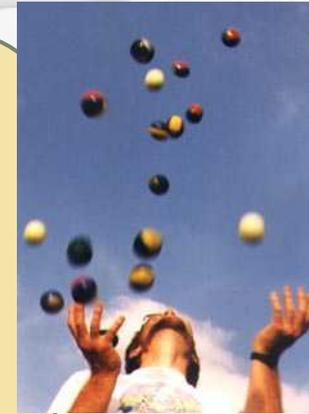
Power

- Traditional voltage scaling is over
- Power now a major design constraint
- Cost of ownership
- Driving significant changes in architecture



Concurrency

- A billion operations per clock
- Billions of refs in flight at all times
- Will require *huge* problems
- Need to exploit *all* available parallelism



Programming Difficulty

- Concurrency and new micro-architectures will significantly complicate software
- Need to hide this complexity from the users



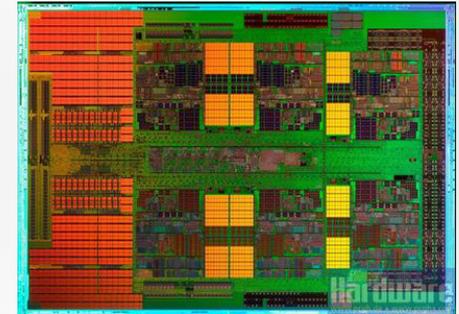
Resiliency

- Many more components
- Components getting less reliable
- Checkpoint bandwidth not scaling



Improving Processor Efficiency

- Multi-core was a good first response to power issues
 - Performance through parallelism
 - Modest clock rate
 - Exploit on-chip locality
- However, conventional processor architectures are optimized for single thread performance rather than energy efficiency
 - Fast clock rate with latency(performance)-optimized memory structures
 - Wide superscalar instruction issue with dynamic conflict detection
 - Heavy use of speculative execution and replay traps
 - Large structures supporting various types of predictions
 - Relatively little energy spent on actual ALU operations
- Could be much more energy efficient with multiple simple processors, exploiting vector/SIMD parallelism and a slower clock rate
- But serial thread performance is really important (Amdahl's Law):
 - If you get great parallel speedup, but hurt serial performance, then you end up with a niche processor (less generally applicable, harder to program)

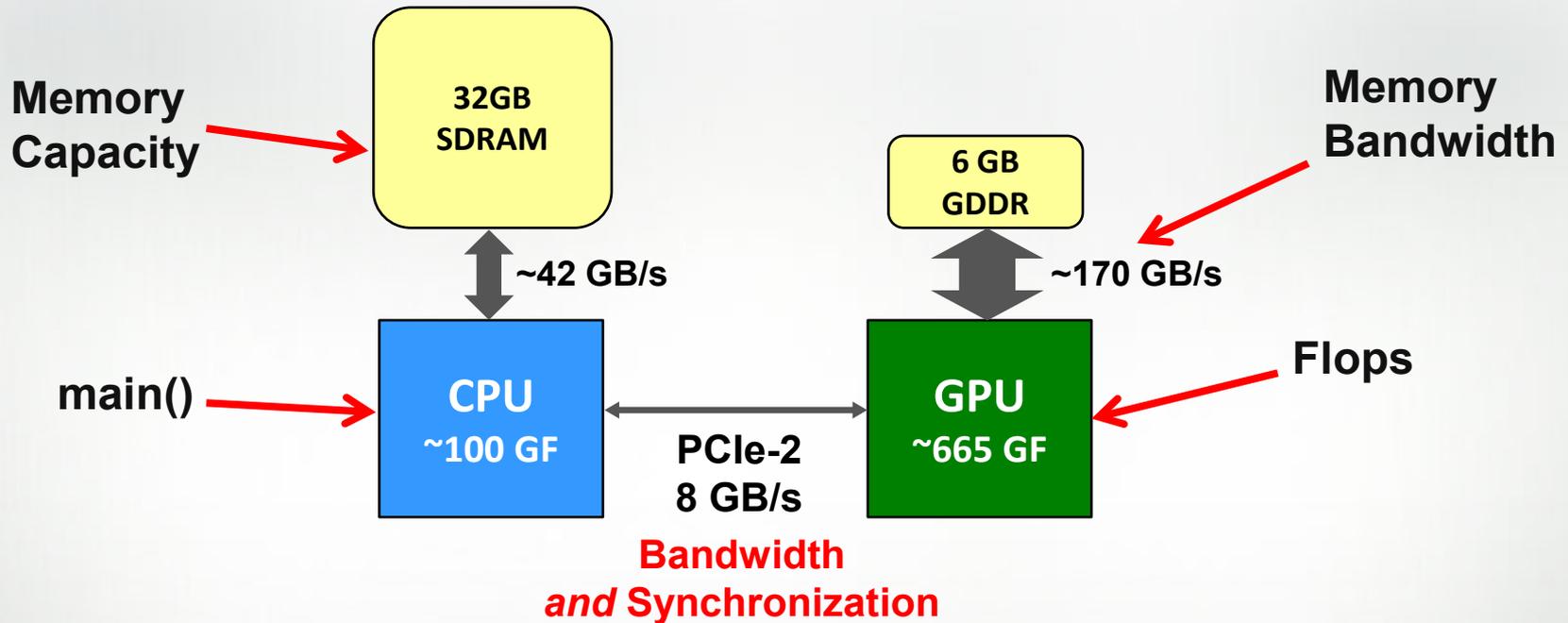


Conclusion: Heterogeneous Computing

- To achieve scale and sustained performance per {\$,watt}, must adopt:
 - ...a *heterogeneous* node architecture
 - fast cores for serial code
 - many power-efficient cores for parallel code
 - ...a deep, explicitly managed memory hierarchy
 - to better exploit locality, improve predictability, and reduce overhead
 - ...a microarchitecture to exploit parallelism at all levels of a code
 - distributed memory, shared memory, vector/SIMD, multithreaded
 - (related to the “concurrency” challenge—leave no parallelism untapped)
- Sounds a lot like GPU accelerators...
- NVIDIA Fermi™ has made GPUs feasible for HPC
 - Robust error protection and strong DP FP, plus programming enhancements
- Expect GPUs to make continued and significant inroads into HPC
 - Compelling technical reasons
 - High volume market
 - It looks like they can credibly support both masters (graphics and compute)
- *Two issues w/ GPU acceleration: **STRUCTURAL** and **PROGRAMMING***

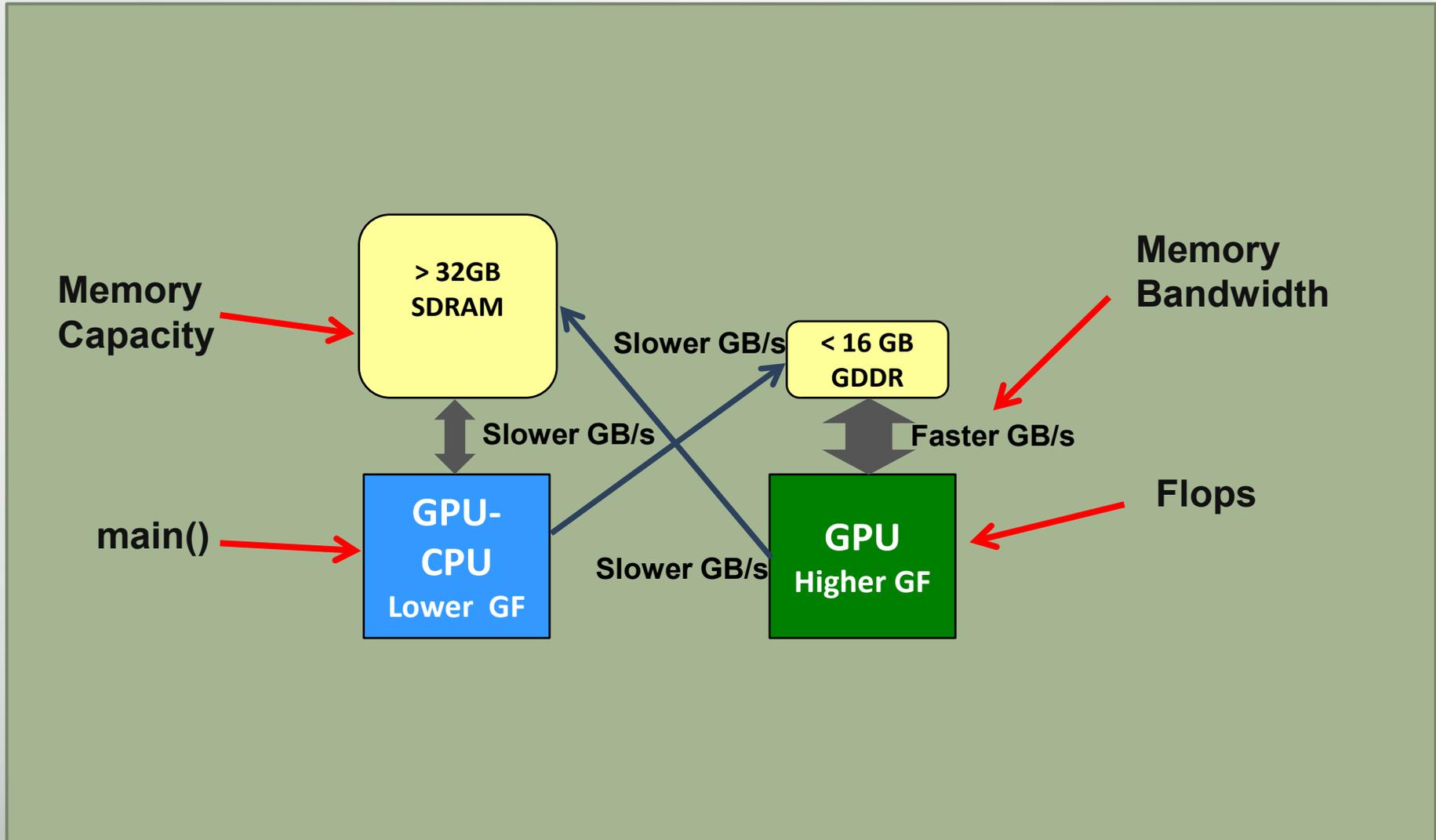


Structural Issues with Accelerated Computing

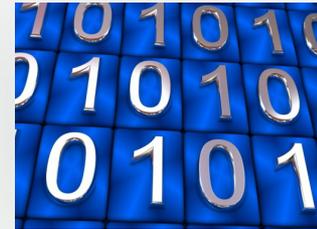


- This is a short-lived situation
 - NVIDIA Denver and AMD Fusion
- Try to keep kernel data structures resident in GPU memory
 - Avoids copying b/w CPU and GPU; work on GPU-network communication
- May limit breadth of applicability over next 2-3 years

Structural Issues with Accelerated Computing Even with fused products

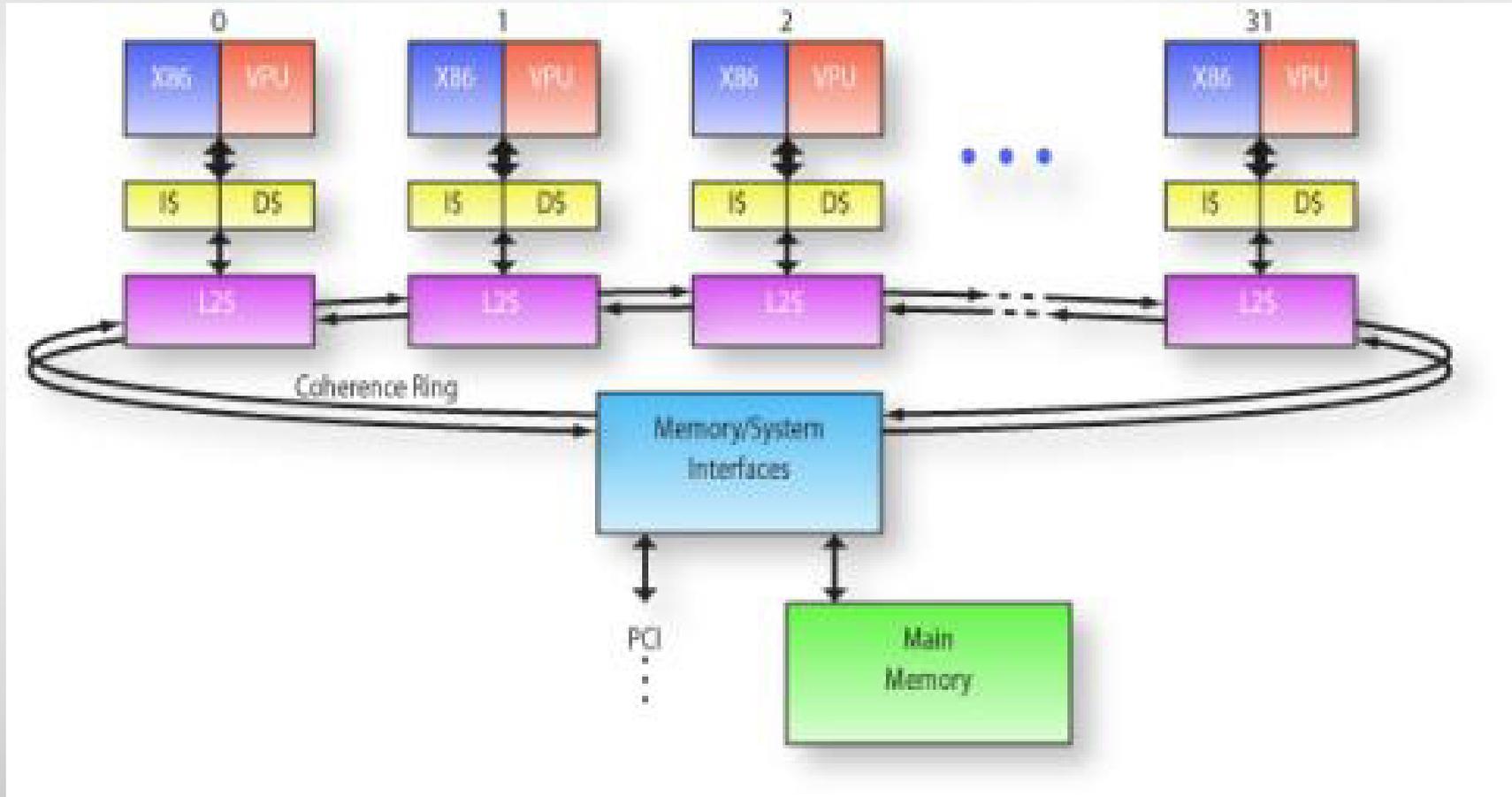


Programming Issues with Accelerated Computing

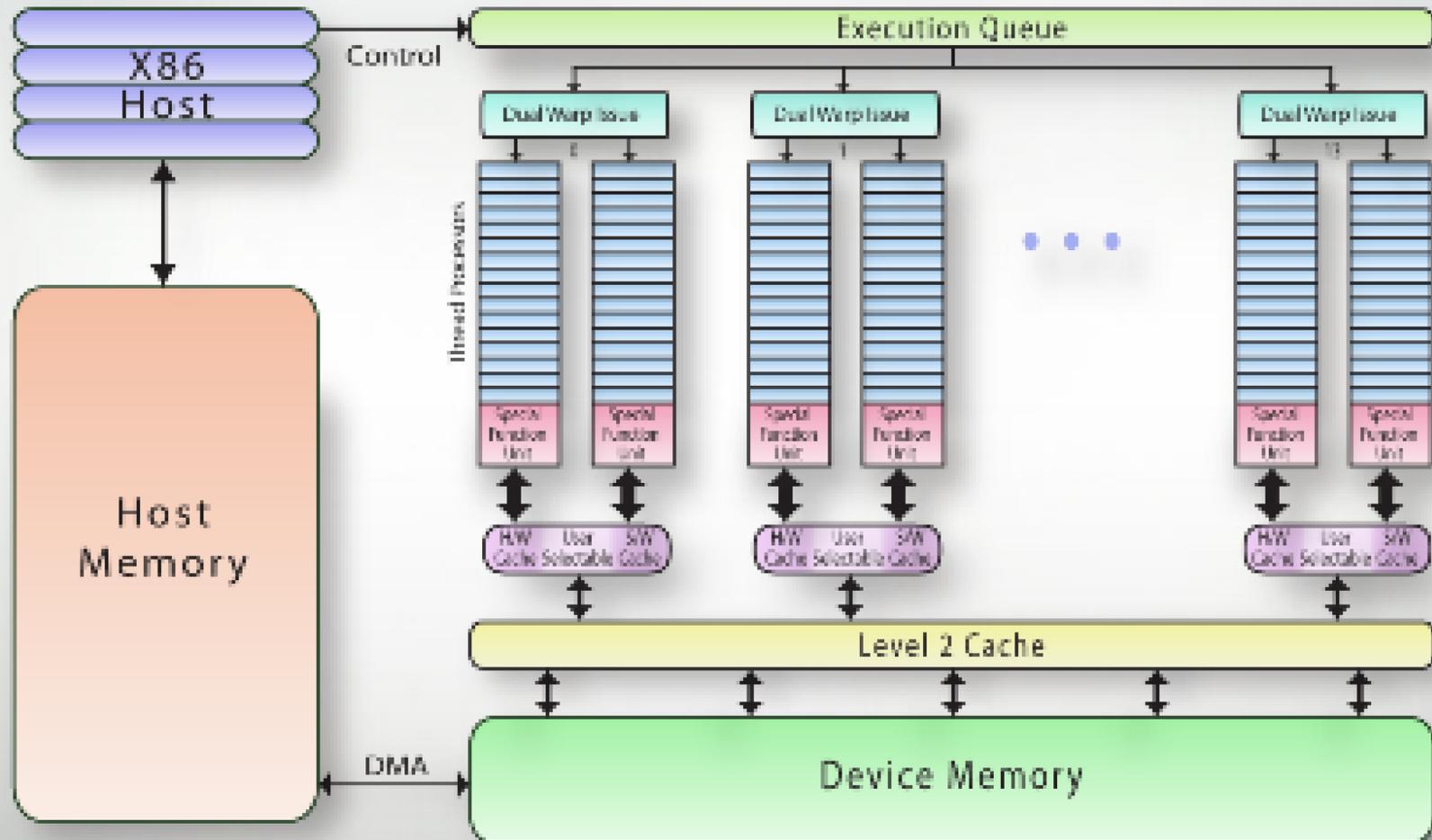


- Primary issues with programming for GPUs:
 - Learn new language/programming model
 - Maintain two code bases/lack of portability
 - Tuning for complex processor architecture (and split CPU/GPU structure)
- Need a single programming model that is **portable across machine types**, and also **forward scalable** in time
 - Portable expression of heterogeneity and multi-level parallelism
 - Programming model and optimization should not be significantly difference for “accelerated” nodes and multi-core x86 processors
 - *Allow users to maintain a single code base*
- Need to shield user from the complexity of dealing with heterogeneity
 - High level language with good complier and runtime support
 - Optimized libraries for heterogeneous multicore processors
- Directive-based approach makes sense (OpenACC)
- Getting the division of labor right:
 - User should focus on identifying parallelism (we can help with good tools)
 - Compiler and runtime can deal with mapping it onto the hardware

Intel 's Knight's Ferry



Nvidia Fermi



©2010 The Portland Group, Inc.

	Intel MIC	NVIDIA Fermi
MIMD Parallelism	32	32
SIMD Parallelism	16	16
Instruction-Level Parallelism	2	1
Thread Granularity	coarse	fine
Multithreading	4	24
Clock	1.2GHz	1.1GHz
L1 cache/processor	32KB	64KB
L2 cache/processor	256KB	24KB
programming model	posix threads/ Directives	CUDA kernels/Directives
virtual memory	yes	no
memory shared with host	no	no
hardware parallelism support	no	yes
mature tools	yes	yes

Short Term Petascale Systems – Node Architecture

	Cores on the node	Total threading	Vector Length	Programming Model
Blue Waters	16	32	8	OpenMP/MPI/Vector
Blue Gene Q	16	32	8	OpenMP/MPI/Vector
Magna-Cours	24	24	4	OpenMP/MPI/Vector
Titan	16	32 (768*)	16	Threads/Cuda/Vector
Intel MIC	32	128	8	OpenMP/MPI/Vector
Interlagos	32	64	8	OpenMP/MPI/Vector

* Nvidia allows oversubscription to SIMT units

Hybrid Multi-core Architecture

- Massively Parallel System with high powered nodes that exhibit
 - Multiple levels of parallelism
 - Shared Memory parallelism on the node
 - SIMD vector units on each core or thread
 - Potentially disparate processing units
 - Host with conventional X86 architecture
 - Accelerator with highly parallel – SIMD units
 - Potentially disparate memories
 - Host with conventional DDR memory
 - Accelerator with high bandwidth memory

Hybrid Multi-core Architecture

- All MPI may not be best approach
 - Memory per core will decrease
 - Injection bandwidth/core will decrease
 - Memory bandwidth/core will decrease
- Hybrid MPI + threading on node may be able to
 - Save Memory
 - Reduce amount of off node communication required
 - Reduce amount of memory bandwidth required

And then there is Intel's MIC processor

- Current MICs have 32 Intel processors moving to 50 processors, both of these systems have vector length of 512 bits (8 – 64 bit words of 16-32 bit words)
- While Intel is saying that codes can be compiled directly for the MIC (Including MPI), one has to be concerned about
 - The scalar performance of one of those cores
 - The amount of memory on the MIC
- If there is too much scalar code and/or too much memory required, then the MIC will necessarily be treated like the other accelerators
 - Two disparate memories
 - Two disparate computational engines
- Remember if you cannot outperform the best of class X86 node, then why go to a new architecture?

TITAN



Pre Upgrade Configuration

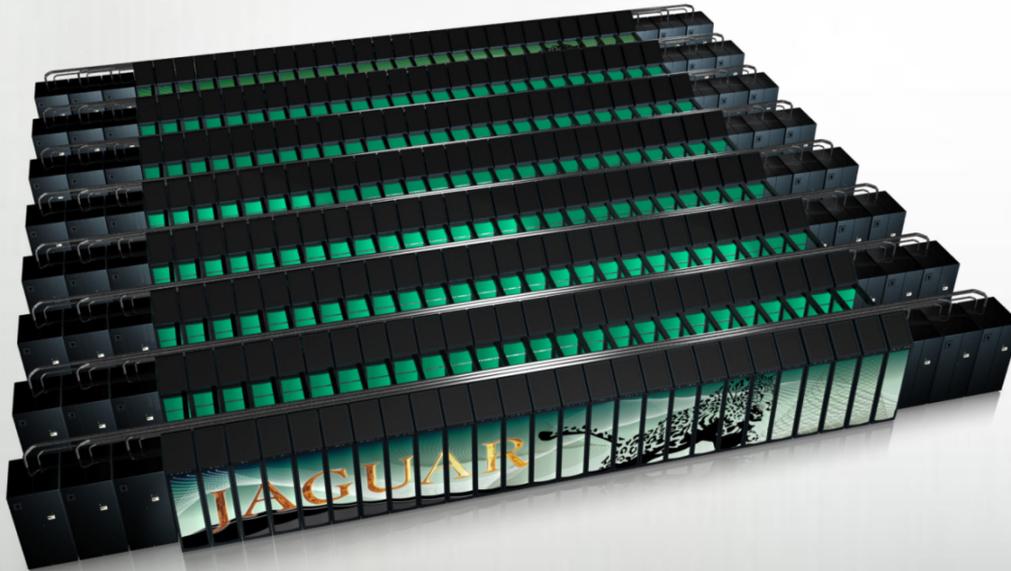
Name	Jaguar
Architecture	XT5
Processor	6-Core AMD
Cabinets	200
Nodes	18,688
Cores/node	12
Total Cores	224,256
Memory/Node	16 GB
Memory/Core	1.3 GB
Interconnect	SeaStar2+
GPUs	0





2011 Configuration

Name	Jaguar
Architecture	XK6
Processor	16-Core AMD
Cabinets	200
Nodes	18,688
Cores/node	16
Total Cores	299,008
Memory/Node	32 GB
Memory/Core	2 GB
Interconnect	Gemini
GPUs	960



And Why is it call an XK6?

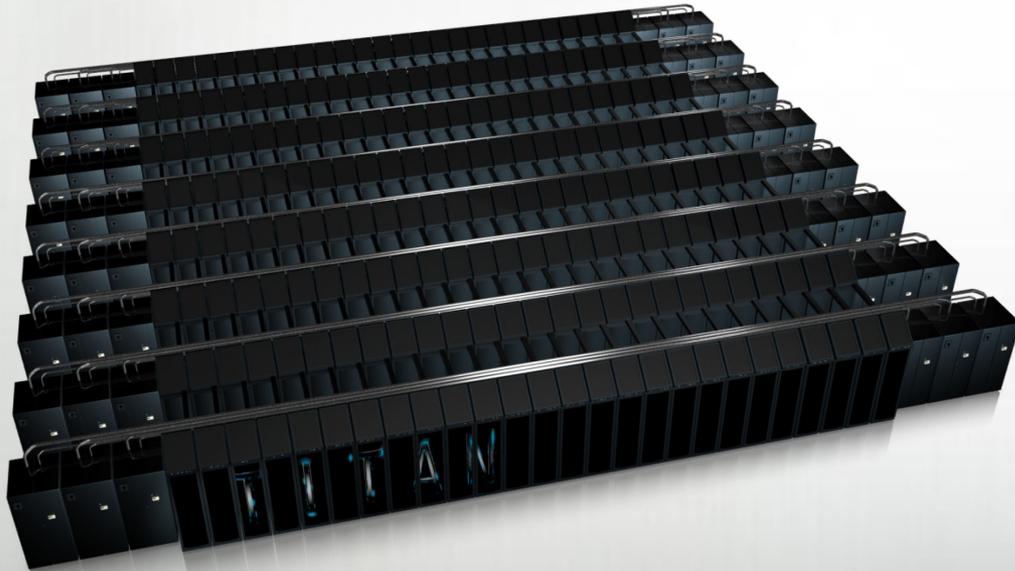


TZ's Sixth Jaguar



Final Configuration

Name	Titan
Architecture	XK6
Processor	16-Core AMD
Cabinets	200
Nodes	18,688
Cores/node	16
Total Cores	299,008
Memory/Node	32 GB
Memory/Core	2 GB
Interconnect	Gemini
GPUs	TBD



- Fact
 - For the next decade all HPC system will basically have the same architecture
 - Message passing between nodes
 - Multi-threading within the node – MPI will not do
 - Vectorization at the lower level -
- Fact
 - Current petascale applications are not structured to take advantage of these architectures
 - Current – 80-90% of application use a single level of parallelism, message passing between the cores of the MPP system
 - Looking forward, application developers are faced with a significant task in preparing their applications for the future

- Tools for identifying additional parallel structures within the application
 - Investigation of looping structures within a complex application
 - Scoping tools for investigating the parallelizability of high level looping structures
- Tools for maintaining performance portable applications
 - Supply compiler that accepts directives from OpenACC and the OpenMP sub-committee formulating extensions to target companion accelerators
 - Application developer able to develop a single code that can run efficiently on multi-core nodes with or without accelerator

Hybridization* of an All MPI Application

* Creation of an application that exhibits three levels of parallelism, MPI between nodes, OpenMP** on the node and vectorized looping structures

** Why OpenMP? To provide performance portability. OpenMP is the only threading construct that a compiler can analyze sufficiently to generate efficient threading on multi-core nodes and to generate efficient code for companion accelerators.

CAUTION!!

- Do not read “Automatic” into this presentation, the Hybridization of an application is difficult and efficient code only comes with a thorough interaction with the compiler to generate the most efficient code and
 - High level OpenMP structures
 - Low level vectorization of major computational areas
- Performance is also dependent upon the location of the data. Best case is that the major computational arrays reside on the accelerator. Otherwise computational intensity of the accelerated kernel must be significant

**Cray's Hybrid Programming Environment
supplies tools for addressing these issues**

Three levels of Parallelism required

- Developers will continue to use MPI between nodes or sockets
- Developers must address using a shared memory programming paradigm on the node
- Developers must vectorize low level looping structures
- While there is a potential acceptance of new languages for addressing all levels directly. Most developers cannot afford this approach until they are assured that the new language will be accepted and the generated code is within a reasonable performance range

Possible Programming Models for the Node

- Cuda
- OpenCL
- Existing Fortran, C and C++ with extensions
 - Pthreads, Thread Building Blocks
 - Comment line directives
 - OpenMP extensions for Accelerators

All of these programming models require the application developer to replace MPI within the node – to develop Hybrid versions of the application

comparisons between Cuda and OpenMP accelerator extensions

- Cuda
 - Widely used programming model for effectively utilizing the accelerator
 - Flexibility to obtain good performance on the accelerator
- OpenMP accelerator extensions – **things to prove**
 - **Are the directives powerful enough to allow the developer to pass information on to the compiler**
 - **Can the compiler generate code that get performance close to Cuda.**

Consider the following kernel

```

do k = 1,nz
  do j=1,ny
    SPECIES: do n=1,n_spec-1
      do i = 1,nx
        diffFlux(i,j,k,n,1) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,1) &
          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,1) )
        diffFlux(i,j,k,n,2) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,2) &
          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,2) )
        diffFlux(i,j,k,n,3) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,3) &
          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,3) )
      enddo
      do i = 1,nx
        diffFlux(i,j,k,n_spec,1) = diffFlux(i,j,k,n_spec,1) - diffFlux(i,j,k,n,1)
        diffFlux(i,j,k,n_spec,2) = diffFlux(i,j,k,n_spec,2) - diffFlux(i,j,k,n,2)
        diffFlux(i,j,k,n_spec,3) = diffFlux(i,j,k,n_spec,3) - diffFlux(i,j,k,n,3)
      enddo
    enddo SPECIES
    do i = 1,nx
      grad_T(i,j,k,1) = -lambda(i,j,k) * grad_T(i,j,k,1)
      grad_T(i,j,k,2) = -lambda(i,j,k) * grad_T(i,j,k,2)
      grad_T(i,j,k,3) = -lambda(i,j,k) * grad_T(i,j,k,3)
    enddo
    do n=1,n_spec
      do i = 1,nx
        grad_T(i,j,k,1) = grad_T(i,j,k,1) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,1)
        grad_T(i,j,k,2) = grad_T(i,j,k,2) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,2)
        grad_T(i,j,k,3) = grad_T(i,j,k,3) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,3)
      enddo
    enddo
  enddo
enddo
enddo
enddo

```

```
! transfer
```

```
grad_Ys_d = grad_Ys_h  
diffFlux_d = diffFlux_h  
yspecies_d = yspecies_h  
h_spec_d = h_spec_h  
ds_mixavg_d = ds_mixavg_h  
grad_t_d = grad_t_h  
grad_mixMW_d = grad_mixMW_h  
lambda_d = lambda_h
```

```
grid = dim3((nxyz-1)/512+1,1,1)  
threadBlock = dim3(512,1,1)
```

```
call calcDiffFlux<<<grid,threadBlock>>>(lambda_d, grad_T_d, grad_mixMW_d, diffFlux_d, &  
    Ds_mixavg_d, yspecies_d, grad_Ys_d, h_spec_d)  
ierr = cudaThreadSynchronize()
```

A rewrite for Cuda Fortran

And now the kernel (1)

```
i = (blockIdx%x-1)*blockDim%x + threadIdx%x
if (i <= nxyz) then
  ! move first part of grad T here
  lambda_r = lambda(i)
  grad_T_1 = -lambda_r*grad_T(i,1)
  grad_T_2 = -lambda_r*grad_T(i,2)
  grad_T_3 = -lambda_r*grad_T(i,3)
  ! now diffFlux
  diffFlux_n_spec_1 = diffFlux(i,n_spec,1)
  diffFlux_n_spec_2 = diffFlux(i,n_spec,2)
  diffFlux_n_spec_3 = diffFlux(i,n_spec,3)
  grad_mixMW_1 = grad_mixMW(i,1)
  grad_mixMW_2 = grad_mixMW(i,2)
  grad_mixMW_3 = grad_mixMW(i,3)
  do n=1, n_spec-1
    Ds_mixavg_r = Ds_mixavg(i,n)
    yspecies_r = yspecies(i,n)
    diffFlux_1 = - Ds_mixavg_r *(grad_Ys(i,n,1) + yspecies_r*grad_mixMW_1)
    diffFlux_2 = - Ds_mixavg_r *(grad_Ys(i,n,2) + yspecies_r*grad_mixMW_2)
    diffFlux_3 = - Ds_mixavg_r *(grad_Ys(i,n,3) + yspecies_r*grad_mixMW_3)
    diffFlux_n_spec_1 = diffFlux_n_spec_1 - diffFlux_1
    diffFlux_n_spec_2 = diffFlux_n_spec_2 - diffFlux_2
    diffFlux_n_spec_3 = diffFlux_n_spec_3 - diffFlux_3
    h_spec_r = h_spec(i,n)
    grad_T_1 = grad_T_1 + h_spec_r*diffFlux_1
    grad_T_2 = grad_T_2 + h_spec_r*diffFlux_2
    grad_T_3 = grad_T_3 + h_spec_r*diffFlux_3
    diffFlux(i,n,1) = diffFlux_1
    diffFlux(i,n,2) = diffFlux_2
    diffFlux(i,n,3) = diffFlux_3
  enddo
enddo
```

- 1) Unrolled all dir loops
- 2) Combined two n_spec loops
- 3) Moved around computation
- 4) Assigned most arrays to temps

```
! do n = n_spec iteration and write out final data

  h_spec_r = h_spec(i,n_spec)
  grad_T_1 = grad_T_1 + h_spec_r*diffFlux_n_spec_1
  grad_T_2 = grad_T_2 + h_spec_r*diffFlux_n_spec_2
  grad_T_3 = grad_T_3 + h_spec_r*diffFlux_n_spec_3

  grad_T(i,1) = grad_T_1
  grad_T(i,2) = grad_T_2
  grad_T(i,3) = grad_T_3

  diffFlux(i,n_spec,1) = diffFlux_n_spec_1
  diffFlux(i,n_spec,2) = diffFlux_n_spec_2
  diffFlux(i,n_spec,3) = diffFlux_n_spec_3
endif
```

Or Add directives

```

!$acc parallel private(i,j,k,n)
!$acc loop
o k = 1,nz
  do j=1,ny
    SPECIES: do n=1,n_spec-1
      do i = 1,nx
        diffFlux(i,j,k,n,1) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,1) &
          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,1) )
        diffFlux(i,j,k,n,2) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,2) &
          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,2) )
        diffFlux(i,j,k,n,3) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,3) &
          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,3) )
      enddo
      do i = 1,nx
        diffFlux(i,j,k,n_spec,1) = diffFlux(i,j,k,n_spec,1) - diffFlux(i,j,k,n,1)
        diffFlux(i,j,k,n_spec,2) = diffFlux(i,j,k,n_spec,2) - diffFlux(i,j,k,n,2)
        diffFlux(i,j,k,n_spec,3) = diffFlux(i,j,k,n_spec,3) - diffFlux(i,j,k,n,3)
      enddo
    enddo SPECIES
    do i = 1,nx
      grad_T(i,j,k,1) = -lambda(i,j,k) * grad_T(i,j,k,1)
      grad_T(i,j,k,2) = -lambda(i,j,k) * grad_T(i,j,k,2)
      grad_T(i,j,k,3) = -lambda(i,j,k) * grad_T(i,j,k,3)
    enddo
    do n=1,n_spec
      do i = 1,nx
        grad_T(i,j,k,1) = grad_T(i,j,k,1) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,1)
        grad_T(i,j,k,2) = grad_T(i,j,k,2) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,2)
        grad_T(i,j,k,3) = grad_T(i,j,k,3) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,3)
      enddo
    enddo
  enddo
enddo
!$acc end loop
!$acc end parallel

```

What the compiler Shows

```

218. 1 G-----< !$acc parallel private(i,j,k,n)
219. 1 G          !$acc loop
220. 1 G C-----<      do k = 1,nz
221. 1 G C g-----<      do j=1,ny
222. 1 G C g 5-----<          SPECIES: do n=1,n_spec-1
223. 1 G C g 5 gf--<              do i = 1,nx
224. 1 G C g 5 gf          ! driving force is just the gradient in mole fraction:
225. 1 G C g 5 gf          diffFlux(i,j,k,n,1) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,1) &
226. 1 G C g 5 gf          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,1) )
227. 1 G C g 5 gf          diffFlux(i,j,k,n,2) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,2) &
228. 1 G C g 5 gf          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,2) )
229. 1 G C g 5 gf          diffFlux(i,j,k,n,3) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,3) &
230. 1 G C g 5 gf          + yspecies(i,j,k,n) * grad_mixMW(i,j,k,3) )
231. 1 G C g 5 gf-->              enddo
232. 1 G C g 5 f---<              do i = 1,nx
233. 1 G C g 5 f          diffFlux(i,j,k,n_spec,1) = diffFlux(i,j,k,n_spec,1) - diffFlux(i,j,k,n,1)
234. 1 G C g 5 f          diffFlux(i,j,k,n_spec,2) = diffFlux(i,j,k,n_spec,2) - diffFlux(i,j,k,n,2)
235. 1 G C g 5 f          diffFlux(i,j,k,n_spec,3) = diffFlux(i,j,k,n_spec,3) - diffFlux(i,j,k,n,3)
236. 1 G C g 5 f--->              enddo
237. 1 G C g 5----->          enddo SPECIES
238. 1 G C g g-----<              do i = 1,nx
239. 1 G C g g          grad_T(i,j,k,1) = -lambda(i,j,k) * grad_T(i,j,k,1)
240. 1 G C g g          grad_T(i,j,k,2) = -lambda(i,j,k) * grad_T(i,j,k,2)
241. 1 G C g g          grad_T(i,j,k,3) = -lambda(i,j,k) * grad_T(i,j,k,3)
242. 1 G C g g----->              enddo
243. 1 G C g 5-----<          do n=1,n_spec
244. 1 G C g 5 g---<              do i = 1,nx
245. 1 G C g 5 g          grad_T(i,j,k,1) = grad_T(i,j,k,1) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,1)
246. 1 G C g 5 g          grad_T(i,j,k,2) = grad_T(i,j,k,2) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,2)
247. 1 G C g 5 g          grad_T(i,j,k,3) = grad_T(i,j,k,3) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,3)
248. 1 G C g 5 g--->              enddo
249. 1 G C g 5----->          enddo
250. 1 G C g----->          enddo
251. 1 G C----->          enddo
252. 1 G          !$acc end loop
253. 1 G-----> !$acc end parallel

```

Legend to compiler Notes

Primary Loop Type

A - Pattern matched

C - Collapsed

D - Deleted

E - Cloned

G - Accelerated

I - Inlined

M - Multithreaded

V - Vectorized

Modifiers

a - atomic memory operation

b - blocked

c - conditional and/or computed

f - fused

g - partitioned

i - interchanged

m - partitioned

n - non-blocking remote transfer

p - partial

r - unrolled

s - shortloop

w - unwound

And a little restructuring

```

!$acc parallel private(t_1,t_2,t_3,I,j,k,n)
!$acc loop collapse(3)
  do k = 1,nz
    do j=1,ny
      do i = 1,nx
        grad_T(i,j,k,1) = -lambda(i,j,k) * grad_T(i,j,k,1)
        grad_T(i,j,k,2) = -lambda(i,j,k) * grad_T(i,j,k,2)
        grad_T(i,j,k,3) = -lambda(i,j,k) * grad_T(i,j,k,3)
        t_1=diffFlux(i,j,k,n_spec,1)
        t_2=diffFlux(i,j,k,n_spec,2)
        t_3=diffFlux(i,j,k,n_spec,3)
        SPECIES: do n=1,n_spec
          if(n < n_spec) then
            diffFlux(i,j,k,n,1) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,1) &
              + yspecies(i,j,k,n) * grad_mixMW(i,j,k,1) )
            diffFlux(i,j,k,n,2) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,2) &
              + yspecies(i,j,k,n) * grad_mixMW(i,j,k,2) )
            diffFlux(i,j,k,n,3) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,3) &
              + yspecies(i,j,k,n) * grad_mixMW(i,j,k,3) )

            t_1 = t_1 - diffFlux(i,j,k,n,1)
            t_2 = t_2 - diffFlux(i,j,k,n,2)
            t_3 = t_3 - diffFlux(i,j,k,n,3)
          end if
          grad_T(i,j,k,1) = grad_T(i,j,k,1) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,1)
          grad_T(i,j,k,2) = grad_T(i,j,k,2) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,2)
          grad_T(i,j,k,3) = grad_T(i,j,k,3) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,3)
        enddo SPECIES
        diffFlux(i,j,k,n_spec,1) = t_1
        diffFlux(i,j,k,n_spec,2) = t_2
        diffFlux(i,j,k,n_spec,3) = t_3
      enddo
    enddo
  enddo
!$acc end loop
!$acc end parallel

```

What the compiler Shows

```

219. 1 G-----< !$acc parallel private(t_1,t_2,t_3,I,j,k,n)
220. 1 G          !$acc loop collapse(3)
221. 1 G C-----<      do k = 1,nz
222. 1 G C C-----<      do j=1,ny
223. 1 G C C g-----<      do i = 1,nx
224. 1 G C C g          grad_T(i,j,k,1) = -lambda(i,j,k) * grad_T(i,j,k,1)
225. 1 G C C g          grad_T(i,j,k,2) = -lambda(i,j,k) * grad_T(i,j,k,2)
226. 1 G C C g          grad_T(i,j,k,3) = -lambda(i,j,k) * grad_T(i,j,k,3)
228. 1 G C C g          t_1=diffFlux(i,j,k,n_spec,1)
229. 1 G C C g          t_2=diffFlux(i,j,k,n_spec,2)
230. 1 G C C g          t_3=diffFlux(i,j,k,n_spec,3)
232. 1 G C C g 6--<      SPECIES: do n=1,n_spec
234. 1 G C C g 6          if(n < n_spec) then
235. 1 G C C g 6              diffFlux(i,j,k,n,1) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,1) &
236. 1 G C C g 6                  + yspecies(i,j,k,n) * grad_mixMW(i,j,k,1) )
237. 1 G C C g 6              diffFlux(i,j,k,n,2) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,2) &
238. 1 G C C g 6                  + yspecies(i,j,k,n) * grad_mixMW(i,j,k,2) )
239. 1 G C C g 6              diffFlux(i,j,k,n,3) = - Ds_mixavg(i,j,k,n) * ( grad_Ys(i,j,k,n,3) &
240. 1 G C C g 6                  + yspecies(i,j,k,n) * grad_mixMW(i,j,k,3) )
242. 1 G C C g 6              t_1 = t_1 - diffFlux(i,j,k,n,1)
243. 1 G C C g 6              t_2 = t_2 - diffFlux(i,j,k,n,2)
244. 1 G C C g 6              t_3 = t_3 - diffFlux(i,j,k,n,3)
246. 1 G C C g 6          end if
248. 1 G C C g 6          grad_T(i,j,k,1) = grad_T(i,j,k,1) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,1)
249. 1 G C C g 6          grad_T(i,j,k,2) = grad_T(i,j,k,2) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,2)
250. 1 G C C g 6          grad_T(i,j,k,3) = grad_T(i,j,k,3) + h_spec(i,j,k,n)*diffFlux(i,j,k,n,3)
251. 1 G C C g 6
252. 1 G C C g 6-->      enddo SPECIES
254. 1 G C C g          diffFlux(i,j,k,n_spec,1) = t_1
255. 1 G C C g          diffFlux(i,j,k,n_spec,2) = t_2
256. 1 G C C g          diffFlux(i,j,k,n_spec,3) = t_3
258. 1 G C C g----->      enddo
259. 1 G C C----->      enddo
260. 1 G C----->      enddo
261. 1 G          !$acc end loop
262. 1 G-----> !$acc end parallel

```

And the timings Ignoring Data Transfer*

	Original OpenMP Across entire node	Cuda Fortran	Directive Approach	Directive Approach Restructured
Kernel Only	.0417 Seconds	.0061 Seconds	.0113 Seconds	.0067 Seconds

* In S3D all of the arrays used in this computation will reside on the Accelerator prior to the invocation of the kernel.

Ease of Use

Kernel	Lines of Code	Cuda lines of Code Added or changed	Lines of Code added or changed for directives	Directive lines of Code added
comp_heat	65	45	0	4
comp_heat (opt)	65	45	9	4
Getrates	5869	6362	0	4
Divergent_sphere	89	44	0	4
Gradient_sphere	45	42	0	4



Task 1 – Identification of potential accelerator kernels

- Identify high level computational structures that account for a significant amount of time (95-99%)
 - To do this, one must obtain global runtime statistics of the application
 - High level call tree with subroutines and DO loops showing inclusive/exclusive time, min, max, average iteration counts.
- **Tools that will be needed**
 - **Advanced instrumentation to measure**
 - DO loop statistics, iteration counts, inclusive time
 - Routine level sampling and profiling

Gathering High Level looping statistics

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function PE=HIDE
100.0%	15.310111	--	--	6301416.2	Total

88.4%	13.535874	--	--	6300424.0	USER

25.3%	3.875405	0.028876	0.8%	2362500.0	parabola_
15.3%	2.338122	0.014399	0.7%	262500.0	remap_
10.0%	1.531963	0.778900	36.0%	262500.0	riemann_
8.5%	1.298146	0.009031	0.7%	262500.0	ppmlr_
6.3%	0.972150	0.008129	0.9%	262500.0	evolve_
3.9%	0.595926	0.002998	0.5%	525000.0	paraset_
3.6%	0.545528	0.003813	0.7%	787500.0	volume_
3.2%	0.485573	0.006530	1.4%	262500.0	states_
2.3%	0.357749	0.341922	52.1%	1.0	vhone_
2.3%	0.351647	0.008985	2.7%	262500.0	flatten_
2.0%	0.309701	0.009954	3.3%	787500.0	forces_
1.7%	0.265953	0.002577	1.0%	140.0	sweepy_
1.7%	0.259623	0.001554	0.6%	70.0	sweepz_
=====					
6.8%	1.037510	--	--	501.2	MPI

5.0%	0.771153	0.754015	52.7%	2.0	mpi_comm_split_
1.7%	0.263760	0.034015	12.2%	420.0	mpi_alltoall_
=====					
4.8%	0.736727	--	--	491.0	MPI_SYNC

3.4%	0.523624	0.514661	98.3%	420.0	mpi_alltoall_(sync)
1.4%	0.213103	0.211751	99.4%	71.0	mpi_allreduce_(sync)
=====					
0.0%	0.000000	--	--	0.0	ETC
=====					

Gathering High Level looping statistics

Table 1: Profile by Function and Callers

Time%	Time	Calls	Group	Function	Caller	PE=HIDE
100.0%	305.000390	23760615.0	Total			

100.0%	304.997547	23760613.0	USER			

28.1%	85.696499	8910000.0	parabola_			

3	18.3%	55.915878	5940000.0	remap_		
4				ppmlr_		

5	6.1%	18.737583	1710000.0	sweepy_.LOOP.2.li.34		
6				sweepy_.LOOP.1.li.33		
7				sweepy_.LOOPS		
8				sweepy_		
9				vhone_		
5	6.0%	18.156965	855000.0	sweepz_.LOOP.06.li.50		
6				sweepz_.LOOP.05.li.49		
7				sweepz_.LOOPS		
8				sweepz_		
9				vhone_		
5	3.2%	9.830210	1687500.0	sweepx1_.LOOP.2.li.30		
6				sweepx1_.LOOP.1.li.29		
7				sweepx1_.LOOPS		
8				sweepx1_		
9				vhone_		
5	3.0%	9.191120	1687500.0	sweepx2_.LOOP.2.li.30		
6				sweepx2_.LOOP.1.li.29		
7				sweepx2_.LOOPS		
8				sweepx2_		
9				vhone_		
				vhone_		

Converting the MPI application to a Hybrid OpenMP/MPI application

Task 2 Parallel Analysis, Scoping and Vectorization

- Investigate parallelizability of high level looping structures
 - Often times one level of loop is not enough, must have several parallel loops
 - User must understand what high level DO loops are in fact independent.
 - Without tools, variable scoping of high level loops is very difficult
 - Loops must be more than independent, their variable usage must adhere to private data local to a thread or global shared across all the threads
- Investigate vectorizability of lower level Do loops
 - Cray compiler has been vectorizing complex codes for over 30 years

Task 2 Parallel Analysis and Scoping

- **Tools that will be needed**
 - **Whole program analysis scoping tool with User interaction**
 - The compiler performs an initial parallelization analysis to identify obvious inhibitors to parallelization
 - The User instructs the compiler to ignore various inhibitors if possible
 - The compiler performs an initial scoping analysis and presents the User with concerns with array usage
 - The User works with the environment to trace variables through the high level looping structure, works with the compiler to scope the variables in question.
 - **Vectorization Feedback from the compiler**
 - Tremendous experience from years of vector architectures

Lets scope out the potential OpenMP loops

```

-- Loop starting at line 221
auto shared(cell,local_lx,local_ly,lz,rho,uxyz)
-- Loop starting at line 262
auto shared(b,cell,local_lx,local_ly,lz,r,rho,uxyz)
-- Loop starting at line 438
auto shared(b,cell,local_lx,local_ly,lz,r,rho,uxyz)
-- Loop starting at line 558
auto shared(cell,grad,local_lx,local_ly,lz,rho,wet)
-- Loop starting at line 572
auto shared(cell,grad,local_lx,local_ly,lz,wet)
-- Loop starting at line 591
auto
shared(b,cell,ci1,ci10,ci11,ci12,ci13,ci14,ci2,ci3,ci4,ci5,ci6,ci7,ci8,ci9,grad,
      local_lx,local_ly,lz)
-- Loop starting at line 712
auto firstprivate(crit,icrit)
auto shared(b,cell,cix,ciy,ciz,local_lx,local_ly,lz,r,rho,uxyz)

```

msg-obj: fi FAIL -- Value/Shared Scope Conflict.

```

-- Loop starting at line 784
auto shared(b,index,index_max,r)
-- Loop starting at line 812
auto shared(b,cell,local_lx,local_ly,lz,r)
-- Loop starting at line 965
auto shared(b,cell,local_lx,local_ly,lz,r,rho,uxyz)
-- Loop starting at line 1125
auto shared(b,bbar,blue,cell,local_lx,local_ly,lz,r,rbar,red,rho,surf)
auto reduction(+:bbar,+:rbar)

```

Using directives to give the compiler information

- Developing efficient OpenMP regions is not an easy task; however, the performance will definitely be worth the effort
- compilation of OpenMP regions to accelerator by the compiler is approaching the performance of hand-coded CUDA or OpenCL with the advantage that it results in portable code. And it will only get better.
 - With OpenMP extensions targeting accelerators, data transfers between multi-core socket and the accelerator can be optimized. Utilization of registers and shared memory can also be optimized
 - With OpenMP extensions targeting accelerators, user can control the utilization of the accelerator memory and functional units.

Task 3 Correctness Debugging

- Run transformed application on the accelerator and investigate the correctness and performance
 - Run as OpenMP application on multi-core socket
 - Use multi-core socket Debugger - DDT
 - Run as Hybrid multi-core application across multi-core socket and accelerator
- **Tools That will be needed**
 - **Information that was supplied by the directives/user's interaction with the compiler**

Task 4 Fine tuning of accelerated program

- Understand current performance bottlenecks
 - Is data transfer between multi-core socket and accelerator a bottleneck?
 - Is shared memory and registers on the accelerator being used effectively?
 - Is the accelerator code utilizing the MIMD parallel units?
 - Is the shared memory parallelization load balanced?
 - Is the low level accelerator code vectorized?
 - Are the memory accesses effectively utilizing the memory bandwidth?

Task 4 Fine tuning of accelerated program

- Tools that will be needed:
 - compiler feedback on parallelization and vectorization of input application
 - Hardware counter information from the accelerator to identify bottlenecks in the execution of the application.
 - Information on memory utilization
 - Information on performance of SIMT units

Several other vendors are supplying similar performance gathering tools

Gathering High Level looping statistics

Table 1: Profile by Function and Callers

Time%	Time	Calls	Group
			Function
			Caller
			PE=HIDE
			Thread=HIDE
100.0%	124.290353	14413778.2	Total

86.3%	107.238820	14405003.0	USER

16.6%	20.596195	5400000.0	parabola_

3	10.9%	13.531857	3600000.0 remap_
4			ppmlr_

5	3.5%	4.373766	600000.0 sweepz_.LOOP@li.53
6			sweepz_.REGION@li.51
7			sweepz_
8			vhone_
5	3.5%	4.348439	600000.0 sweepy_.LOOP@li.35
6			sweepy_.REGION@li.33
7			sweepy_
8			vhone_
5	1.9%	2.407785	1200000.0 sweepx1_.LOOP@li.43
6			sweepx1_.REGION@li.40
7			sweepx1_
8			vhone_
5	1.9%	2.401867	1200000.0 sweepx2_.LOOP@li.31
6			sweepx2_.REGION@li.29
7			sweepx2_
8			vhone_
=====			

Gathering High Level looping statistics

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function PE=HIDE
100.0%	94.287239	--	--	22122.2	Total
88.3%	83.254169	--	--	18559.0	USER
24.7%	23.320669	0.040706	0.2%	500.0	sweepy_.ACC_SYNC_WAIT@li.113
23.9%	22.580065	0.068378	0.3%	250.0	sweepz_.ACC_SYNC_WAIT@li.143
11.7%	11.034601	0.036352	0.4%	250.0	sweepx1_.ACC_SYNC_WAIT@li.98
11.6%	10.975658	0.041705	0.4%	250.0	sweepx2_.ACC_SYNC_WAIT@li.95
2.8%	2.628305	0.006066	0.2%	500.0	sweepy_.ACC_COPY@li.113
1.4%	1.316031	0.002970	0.2%	250.0	sweepx2_.ACC_COPY@li.95
1.4%	1.313542	0.002868	0.2%	250.0	sweepx1_.ACC_COPY@li.98
1.4%	1.280935	0.003241	0.3%	250.0	sweepz_.ACC_COPY@li.143
1.3%	1.243179	0.001617	0.1%	500.0	sweepy_.ACC_COPY@li.120
1.0%	0.983736	0.002562	0.3%	500.0	sweepy_.ACC_COPY@li.37
7.7%	7.214336	--	--	1776.0	MPI_SYNC
6.6%	6.197997	5.631113	90.9%	1500.0	mpi_alltoall_(sync)
4.1%	3.818692	--	--	1786.2	MPI
3.2%	2.977898	0.460985	14.3%	1500.0	mpi_alltoall_
0.0%	0.000042	0.000005	11.4%	1.0	OMP
0.0%	0.000000	--	--	0.0	ETC

Compiler listing from GPU code

```

42.  1 G-----< !$acc parallel loop private(i,j,n,radius,xa,dx,xa0,dx0,e,sweep, &
43.  1 G          !$acc   flat,para,dvol,q,localsvel, ngeom,nleft,nright,nmin,nmax) &
44.  1 G          !$acc   reduction(max:svel) copyout(r,u,v,w,f,p)copyin(zro,zpr,zux,zuy,zuz,zfl, &
45.  1 G          !$acc   zxa,zdx,zxa,zdx) copy(svel)
51.  1 G g-----< do j = 1, js
52.  1 G g          sweep = 1
53.  1 G g          ngeom = ngeomx
54.  1 G g          nleft = nleftx
55.  1 G g          nright = nrightx
56.  1 G g          nmin = 7
57.  1 G g          nmax = imax + 6
58.  1 G g          localsvel =-1e9
59.  1 G g gf-----<> flat = 0.0
60.  1 G g          radius = 0.0
61.  1 G g g-----< do i = 1,imax
62.  1 G g g          n = i + 6
63.  1 G g g          r (n,j) = zro(i,j,k)
64.  1 G g g          p (n,j) = zpr(i,j,k)
65.  1 G g g          u (n,j) = zux(i,j,k)
66.  1 G g g          v (n,j) = zuy(i,j,k)
67.  1 G g g          w (n,j) = zuz(i,j,k)
68.  1 G g g          f (n,j) = zfl(i,j,k)
69.  1 G g g
70.  1 G g g          xa0(n) = zxa(i)
71.  1 G g g          dx0(n) = zdx(i)
72.  1 G g g          xa (n) = zxa(i)
73.  1 G g g          dx (n) = zdx(i)
74.  1 G g g          p (n,j) = max(smallp,p(n,j))
75.  1 G g g          e (n) = p(n,j)/(r(n,j)*gamm)+0.5*(u(n,j)**2+v(n,j)**2+w(n,j)**2)
76.  1 G g g-----> enddo
79.  1 G g gfr4 I--> call ppmlr (localsvel,sweep,nmin, nmax, ngeom, nleft, nright,r(1,j), p(1,j), &
80.  1 G g          e, q, u(1,j), v(1,j), w(1,j), xa, xa0, dx, &
81.  1 G g          dx0, dvol,f(1,j), flat,para ,radius)
82.  1 G g          svel = max(localsvel,svel)
83.  1 G g-----> enddo
85.  1 G-----> !$acc end parallel loop

```

Compiler listing from GPU code

```

ftn-6405 ftn: ACCEL File = sweepxl.f90, Line = 42
  A region starting at line 42 and ending at line 85 was placed on the accelerator.

ftn-6417 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory and copy whole array "zro" to accelerator, free at line 85 (acc_copyin).

ftn-6417 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory and copy whole array "zpr" to accelerator, free at line 85 (acc_copyin).

ftn-6417 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory and copy whole array "zux" to accelerator, free at line 85 (acc_copyin).

ftn-6417 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory and copy whole array "zuy" to accelerator, free at line 85 (acc_copyin).

ftn-6419 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory for whole array "v" on accelerator, copy back at line 85 (acc_copyout).

ftn-6419 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory for whole array "w" on accelerator, copy back at line 85 (acc_copyout).

ftn-6419 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory for whole array "f" on accelerator, copy back at line 85 (acc_copyout).

ftn-6419 ftn: ACCEL File = sweepxl.f90, Line = 42
  Allocate memory for whole array "p" on accelerator, copy back at line 85 (acc_copyout).

ftn-6423 ftn: ACCEL File = sweepxl.f90, Line = 42
  Private array "xa" was allocated to global memory.

ftn-6423 ftn: ACCEL File = sweepxl.f90, Line = 42
  Private array "dx" was allocated to global memory.

ftn-6423 ftn: ACCEL File = sweepxl.f90, Line = 42
  Private array "xa0" was allocated to global memory.

```

Results before next step of tuning

All Done during the four day workshop

		Chester OpenMP	Chester OpenACC	Chester AllMPI
totaltime	=	55.4403	39.93815	43.37071
beforetimesteploop	=	6.45E-02	8.26E-02	0.137425
aftertimesteploop	=	0.215131	0.246678	1.453568
timesteploop	=	55.1607	39.60889	41.77971
sweepx1time	=	7.69359	5.896201	6.213687
sweepx2time	=	7.710696	5.756214	6.140237
sweepytime	=	19.70721	13.30642	14.30337
sweepztime	=	19.38663	14.14406	15.00088
timesteploopb4swps	=	1.08E-04	7.69E-05	1.30E-04
timesteploopdtcon	=	0.661224	0.406726	0.119561
timesteploopend	=	3.34E-04	9.90E-02	5.27E-04
timesteploopend1	=	7.58E-05	2.25E-05	1.25E-04
timesteploopend2	=	5.82E-05	2.26E-05	9.45E-05
timesteploopend3	=	4.28E-05	9.89E-02	6.95E-05

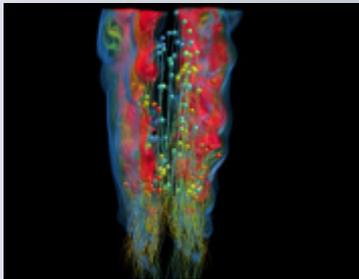
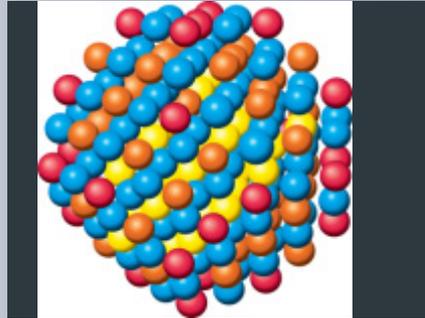
Cray GPU Programming Environment

- Objective: Enhance productivity related to porting applications to hybrid multi-core systems
- Four core components
 - Cray Statistics Gathering Facility on host and GPU
 - Cray Optimization Explorer – Scoping Tools (COE)
 - Cray Compilation Environment (CCE)
 - Cray GPU Libraries

Titan: Early Science Applications

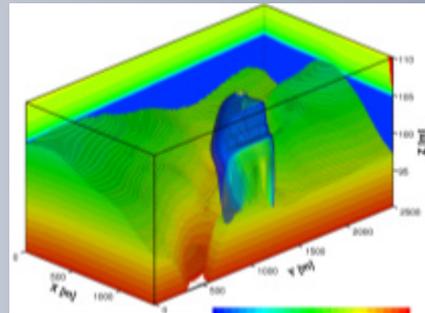
WL-LSMS

Role of material disorder, statistics, and fluctuations in nanoscale materials and systems.



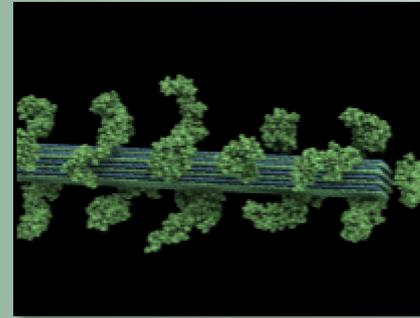
S3D

How are going to efficiently burn next generation diesel/bio fuels?



PFLOTRAN

Stability and viability of large scale CO₂ sequestration; predictive containment groundwater transport

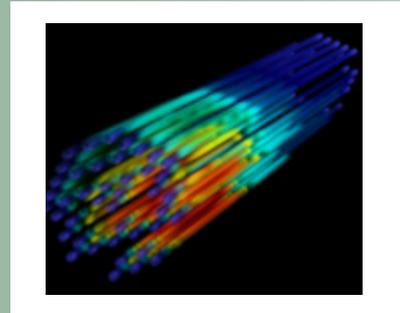
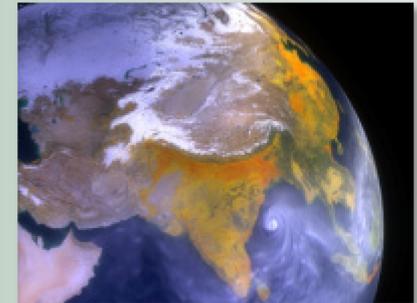


LAMMPS

Biofuels: An atomistic model of cellulose (blue) surrounded by lignin molecules comprising a total of 3.3 million atoms. Water not shown.

CAM / HOMME

Answer questions about specific climate change adaptation and mitigation scenarios; realistically represent features like precipitation patterns/statistics and tropical storms

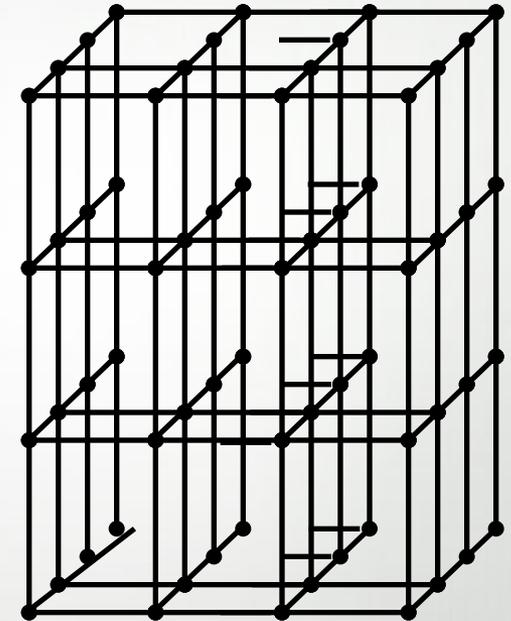


Denovo

Unprecedented high-fidelity radiation transport calculations that can be used in a variety of nuclear energy and technology applications.

S3D – A DNS solver

- Structured Cartesian mesh flow solver
- Solves compressible reacting Navier-Stokes, energy and species conservation equations.
 - 8th order explicit finite difference method
 - 4th order Runge-Kutta integrator with error estimator
- Detailed gas-phase thermodynamic, chemistry and molecular transport property evaluations
- Lagrangian particle tracking
- MPI-1 based spatial decomposition and parallelism
- Fortran code. Does not need linear algebra, FFT or solver libraries.



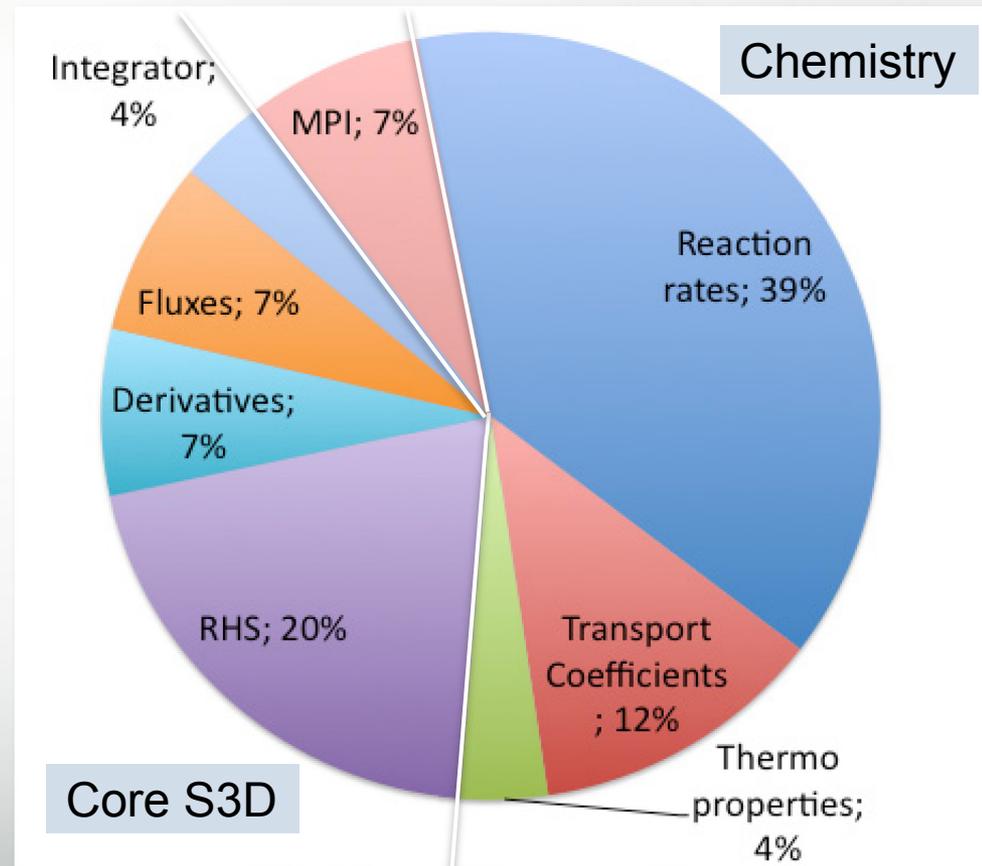
Developed and maintained at CRF, Sandia (Livermore) with BES and ASCR sponsorship. PI – Jacqueline H. Chen (jhchen@sandia.gov)

Benchmark Problem and Profile

- A benchmark problem was defined to closely resemble the target simulation
 - 52 species n-heptane chemistry and 48^3 grid points per node

- $48^3 * 18,500$ nodes = 2 billion grid points
- Target problem would take two months on today's Jaguar

- Code was benchmarked and profiled on dual-hex core XT5
- Several kernels identified and extracted into stand-alone driver programs



Acceleration Strategy

Team:

Ramanan Sankaran ORNL

Ray Grout

NREL

John Levesque

Cray

Goals:

- Convert S3D to a hybrid multi-core application suited for a multi-core node with or without an accelerator.
- Be able to perform the computation entirely on the accelerator.
 - Arrays and data able to reside entirely on the accelerator.
 - Data sent from accelerator to host CPU for halo communication, I/O and monitoring only.

Strategy:

- To program using both hand-written and generated code.
 - Hand-written and tuned CUDA*.
 - Automated Fortran and CUDA generation for chemistry kernels
 - Automated code generation through compiler directives
- S3D is now a part of Cray's compiler development test cases

Original S3D

		S3D		
Time Step		Solve_Drive		
Time Step	Runge K	Integrate		
Time Step	Runge K	RHS		
Time Step	Runge K		get mass fraction	<i>l,j,k,n_spec loops</i>
Time Step	Runge K		get_velocity	<i>l,j,k,n_spec loops</i>
Time Step	Runge K		calc_inv_avg	<i>l,j,k,n_spec loops</i>
Time Step	Runge K		calc_temp Compute	<i>l,j,k,n_spec loops</i>
Time Step	Runge K		Grads	<i>l,j,k,n_spec loops</i>
Time Step	Runge K		Diffusive Flux	<i>l,j,k,n_spec loops</i>
Time Step	Runge K		Derivatives	<i>l,j,k,n_spec loops</i>
Time Step	Runge K		reaction rates	<i>l,j,k,n_spec loops</i>

Profile from Original S3D

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function PE=HIDE Thread=HIDE
100.0%	284.732812	--	--	156348682.1	Total
92.1%	262.380782	--	--	155578796.1	USER
12.4%	35.256420	0.237873	0.7%	391200.0	ratt_i_.LOOPS
9.6%	27.354247	0.186752	0.7%	391200.0	ratx_i_.LOOPS
7.7%	21.911069	1.037701	4.5%	1562500.0	mcedif_.LOOPS
5.4%	15.247551	2.389440	13.6%	35937500.0	mceval4_
5.2%	14.908749	4.123319	21.7%	600.0	rhsf_.LOOPS
4.7%	13.495568	1.229034	8.4%	35937500.0	mceval4_.LOOPS
4.6%	12.985353	0.620839	4.6%	701.0	calc_temp\$thermchem_m_.LOOPS
4.3%	12.274200	0.167054	1.3%	1562500.0	mcavis_new\$transport_m_.LOOPS
4.0%	11.363281	0.606625	5.1%	600.0	computespeciesdiffflux\$transport_m_.LOOPS
2.9%	8.257434	0.743004	8.3%	21921875.0	mixcp\$thermchem_m_
2.9%	8.150646	0.205423	2.5%	100.0	integrate_.LOOPS
2.4%	6.942384	0.078555	1.1%	391200.0	qssa_i_.LOOPS
2.3%	6.430820	0.481475	7.0%	21921875.0	mixcp\$thermchem_m_.LOOPS
2.0%	5.588500	0.343099	5.8%	600.0	computeheatflux\$transport_m_.LOOPS
1.8%	5.252285	0.062576	1.2%	391200.0	rdwdot_i_.LOOPS
1.7%	4.801062	0.723213	13.1%	31800.0	derivative_x_calc_.LOOPS
1.6%	4.461274	1.310813	22.7%	31800.0	derivative_y_calc_.LOOPS
1.5%	4.327627	1.290121	23.0%	31800.0	derivative_z_calc_.LOOPS
1.4%	3.963951	0.138844	3.4%	701.0	get_mass_frac\$variables_m_.LOOPS

S3D

Time Step		Solve_Drive	
Time Step	Runge K	Integrate	
Time Step	Runge K	RHS	
Time Step	Runge K	grid loop -omp	get mass fraction
Time Step	Runge K	grid loop-omp	get_velocity
Time Step	Runge K	grid loop-omp	calc_inv_avg
Time Step	Runge K	grid loop-omp	calc_temp
Time Step	Runge K	grid loop-omp	Compute Grads
Time Step	Runge K	grid loop-omp	Diffusive Flux
Time Step	Runge K	grid loop-omp	Derivatives
Time Step	Runge K	grid loop-omp	reaction rates

Statistics from running S3D

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group	Function
85.3%	539.077983	--	--	144908.0	USER	
21.7%	136.950871	0.583731	0.5%	600.0	rhsf_	
14.7%	93.237279	0.132829	0.2%	600.0	rhsf_.LOOP@li.1084	
8.7%	55.047054	0.309278	0.6%	600.0	rhsf_.LOOP@li.1098	
6.3%	40.129463	0.265153	0.8%	100.0	integrate_	
5.8%	36.647080	0.237180	0.7%	600.0	rhsf_.LOOP@li.1211	
5.6%	35.264114	0.091537	0.3%	600.0	rhsf_.LOOP@li.194	
3.7%	23.624271	0.054666	0.3%	600.0	rhsf_.LOOP@li.320	
2.7%	17.211435	0.095793	0.6%	600.0	rhsf_.LOOP@li.540	
2.4%	15.471160	0.358690	2.6%	14400.0	derivative_y_calc_buff_r_.LOOP@li.1784	
2.4%	15.113374	1.020242	7.2%	14400.0	derivative_z_calc_buff_r_.LOOP@li.1822	
2.3%	14.335142	0.144579	1.1%	14400.0	derivative_x_calc_buff_r_.LOOP@li.1794	
1.9%	11.794965	0.073742	0.7%	600.0	integrate_.LOOP@li.96	
1.7%	10.747430	0.063508	0.7%	600.0	computespeciesdiffflux2\$transport_m_.LOOP	
1.5%	9.733830	0.096476	1.1%	600.0	rhsf_.LOOP@li.247	
1.2%	7.649953	0.043920	0.7%	600.0	rhsf_.LOOP@li.274	
0.8%	5.116578	0.008031	0.2%	600.0	rhsf_.LOOP@li.398	
0.6%	3.966540	0.089513	2.5%	1.0	s3d_	
0.3%	2.027255	0.017375	1.0%	100.0	integrate_.LOOP@li.73	
0.2%	1.318550	0.001374	0.1%	600.0	rhsf_.LOOP@li.376	
0.2%	0.986124	0.017854	2.0%	600.0	rhsf_.REGION@li.1096	
0.1%	0.700156	0.027669	4.3%	1.0	exit	

Advantage of raising loops

- Create good granularity OpenMP Loop
- Improves cache re-use
- Reduces Memory usage significantly
- Creates a good potential kernel for an accelerator

S3D

Time Step – acc_data		Solve_Drive	
Time Step– acc_data	Runge K	Integrate	
Time Step– acc_data	Runge K	RHS	
Time Step– acc_data	Runge K	grid loop -ACC	get mass fraction
Time Step– acc_data	Runge K	grid loop-ACC	get_velocity
Time Step– acc_data	Runge K	grid loop-ACC	calc_inv_avg
Time Step– acc_data	Runge K	grid loop-ACC	calc_temp
Time Step– acc_data	Runge K	grid loop-ACC	Compute Grads
Time Step– acc_data	Runge K	grid loop-ACC	Diffusive Flux
Time Step– acc_data	Runge K	grid loop-ACC	Derivatives
Time Step– acc_data	Runge K	grid loop-ACC	reaction rates

Thank you. Questions?

