Installation of Big Red II at Indiana University (2 min)
http://www.youtube.com/watch?v=LJ2woKhQBo8
## Blue Waters System

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray System &amp; Storage cabinets:</td>
<td>&gt;300</td>
</tr>
<tr>
<td>Compute nodes:</td>
<td>&gt;25,000</td>
</tr>
<tr>
<td>Usable Storage Bandwidth:</td>
<td>&gt;1 TB/s</td>
</tr>
<tr>
<td>System Memory:</td>
<td>&gt;1.5 Petabytes</td>
</tr>
<tr>
<td>Memory per core:</td>
<td>4 GB</td>
</tr>
<tr>
<td>Gemini Interconnect Topology:</td>
<td>3D Torus</td>
</tr>
<tr>
<td>Usable Storage:</td>
<td>&gt;25 Petabytes</td>
</tr>
<tr>
<td>Peak performance:</td>
<td>&gt;11.5 Petaflops</td>
</tr>
<tr>
<td>Number of AMD processors:</td>
<td>&gt;49,000</td>
</tr>
<tr>
<td>Number of AMD x86 core:</td>
<td>&gt;380,000</td>
</tr>
<tr>
<td>Number of NVIDIA GPUs:</td>
<td>&gt;3,000</td>
</tr>
</tbody>
</table>
Interlagos was just announced this week, but we have been shipping it in XE6 product for over 1 month.

**Node Characteristics**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Compute Modules</td>
<td>8 per socket or 16 per node</td>
</tr>
<tr>
<td>MPI Ranks</td>
<td>16 or 32</td>
</tr>
<tr>
<td>Peak Performance IL-16 (2.3)</td>
<td>294 Gflops/sec</td>
</tr>
<tr>
<td>Memory Sizes</td>
<td>32 GB per node, 64 GB per node, 128GB per node</td>
</tr>
<tr>
<td>Memory Bandwidth (Peak)</td>
<td>102.4 GB/sec</td>
</tr>
</tbody>
</table>
# Accelerated Performance through Integration

<table>
<thead>
<tr>
<th><strong>XK6 Compute Node Characteristics</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host Processor</strong></td>
<td>AMD Series 6200 (Interlagos)</td>
</tr>
<tr>
<td><strong>Tesla X2090 Perf.</strong></td>
<td>665 Gflops</td>
</tr>
<tr>
<td><strong>Host Memory</strong></td>
<td>16, 32, or 64GB 1600 MHz DDR3</td>
</tr>
<tr>
<td><strong>Tesla X2090 Memory</strong></td>
<td>6GB GDDR5 170 GB/sec</td>
</tr>
<tr>
<td><strong>Gemini High Speed Interconnect</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Upgradeable to Kepler many-core processor</strong></td>
<td></td>
</tr>
</tbody>
</table>
CUDA Core count  512
Peak Performance (DP) 665GFlops
Power consumption 225W
Memory type and size 6GB of GDDR5 (ECC)
Custom heat sink for XK6
Greater than 3TF per blade
Easily Upgrade to Future Accelerators
XE6 I/O Blade
High Density Packaging

- 24 blades in a cabinet
- Air or liquid cooled
- One central blower to air-cool blades
- Integrated Hardware Supervisory System
- Warm swappable blades
ECOphlex Cooling – Cray XE6

Hot air is cooled using liquid cooling before being exhausted into the computer room.

Air heats as it passes through the compute blades.

Room Neutral Air Exhaust

Exit Evaporators

Cooling Coil (Cray ECOphlex R134 piping)

Cool air is released into the computer room.

Liquid in

Liquid/Vapor Mixture out

Hot air stream passes through evaporator, rejects heat to R134a via liquid-vapor phase change (evaporation).

Room ambient air inlet
Custom Interconnects
Cray Gemini Network ASIC

- **Strong MPI Support**
  - Low Latency \(-1.2 \mu s\)
  - >10M independent messages/sec/NIC
  - Two methods to move data (FMA and BTE)

- **Advanced Synchronization and Communication Features**
  - Efficient support for UPC, CAF, One-sided MPI and Global Arrays
  - Atomic memory operations
  - Pipelined global loads and stores
    - \(~25M\) random Puts/sec/NIC
    - \(~65M\) indexed Puts/sec/NIC

- **Embedded high-performance router**
  - 160GB/sec capacity per chip
  - Scales to over 1 million cores
What’s changed since 2009?
1. DDR has moved to QDR
2. SeaStar has moved to Gemini
3. Cores per node have grown by a factor of 4 to 8

How has this picture changed?
A Comparison of the Performance Characteristics of Capability and Capacity Class HPC Systems
By Douglas Doerfler, Mahesh Rajan, Marcus Epperson, Courtenay Vaughan, Kevin Pedretti, Richard Barrett, Brian Barrett, Sandia National Laboratories
A Comparison of Two Codes with Very Different Communication Patterns

Figure 16. AMG Scaling Performance (Lower is better)

Figure 17. AMG inter-processor communication CrayPat plot

Figure 9. Charon inter-processor communication CrayPat plot

Figure 8. Charon scaling performance (Lower is better)

Irregular Pattern

Nearest Neighbor
Memory footprint (all 2 all)

- With 16k processes you lose more than 75% of your memory for communication buffers and connection handling! (JuRoPA 24GB/node ~ 3GB/process)

- Cray XE6 memory overhead is between 2% and 3% of the node memory at 60K MPI ranks

95% of Memory @ 9K MPI ranks Qsize=8

75% of Memory @ 16K MPI ranks Qsize=3

Q_SIZE MB / connection
- 30.141
- 40.172
- 80.305

(Measured on JuRoPA by N. Eicker)
Gemini Advanced Features

- Globally addressable memory provides efficient support for UPC, Co-array FORTRAN, Shmem and Global Arrays
  - Cray Programming Environment will target this capability directly

- Pipelined global loads and stores
  - Allows for fast irregular communication patterns

- Atomic memory operations
  - Provides fast synchronization needed for one-sided communication models
### Cray Programming Environment Distribution

**Focus on Differentiation and Productivity**

<table>
<thead>
<tr>
<th>Programming Languages</th>
<th>Compilers</th>
<th>Programming models</th>
<th>I/O Libraries</th>
<th>Tools</th>
<th>Optimized Scientific Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>Cray Compiling Environment (CCE)</td>
<td>Distributed Memory (Cray MPT)</td>
<td>NetCDF</td>
<td>Environment setup</td>
<td>LAPACK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MPI</td>
<td></td>
<td>Modules</td>
<td>ScaLAPCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SHMEM</td>
<td></td>
<td>Debuggers</td>
<td>BLAS (libgoto)</td>
</tr>
<tr>
<td>C</td>
<td>PGI</td>
<td>Shared Memory</td>
<td>HDF5</td>
<td>Allinea DDT</td>
<td>Iterative Refinement Toolkit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• OpenMP 3.0</td>
<td></td>
<td>Igdb</td>
<td>Cray Adaptive FFTs (CRAFFT)</td>
</tr>
<tr>
<td></td>
<td>PathScale</td>
<td>(CCE &amp; PGI)</td>
<td></td>
<td>Debugging Support Tools</td>
<td>FFTW</td>
</tr>
<tr>
<td></td>
<td>GNU</td>
<td>PGAS &amp; Global View</td>
<td></td>
<td>• Fast Track Debugger (CCE with DDT)</td>
<td>Cray PETSc (with CASK)</td>
</tr>
<tr>
<td></td>
<td>intel</td>
<td>• UPC (CCE)</td>
<td></td>
<td>• Abnormal Termination Processing</td>
<td>Cray Trilinos (with CASK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CAF (CCE)</td>
<td></td>
<td>STAT</td>
<td>Performance Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Chapel</td>
<td></td>
<td>Cray Comparative Debugger</td>
<td>Cray Performance Monitoring and Analysis Tool</td>
</tr>
</tbody>
</table>

- **Cray developed**
  - #: Under development
  - Licensed ISV SW
  - 3rd party packaging
- **Cray added value to 3rd party**
Example 1: reduction

- Sum elements of an array
- Original Fortran code

```
a=0.0

do i = 1,n
   a = a + b(i)
end do
```
The reduction code in simple CUDA

```c
__global__ void reduce0(int *g_idata, int *g_odata)
{
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if ((tid % (2*s)) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}

extern "C" void reduce0_cuda_(int *n, int *a, int *b)
{
    int *b_d,red;
    const int b_size = *n;

cudaMalloc((void **) &b_d , sizeof(int)*b_size);
cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

*red = red;
}
```

```c
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(2048, 1, 1);
dim3 small_dimGrid(16, 1, 1);

int smemSize = 128 * sizeof(int);
int *buffer_d, *red_d;
int *small_buffer_d;

cudaMalloc((void **) &buffer_d ,
    sizeof(int)*2048);
cudaMalloc((void **) &small_buffer_d ,
    sizeof(int)*16);
cudaMalloc((void **) &red_d , sizeof(int));

reduce0<<< dimGrid, dimBlock, smemSize >>>(b_d, buffer_d);
reduce0<<< small_dimGrid, dimBlock, smemSize >>>(buffer_d, small_buffer_d);
reduce0<<< 1, 16, smemSize >>>(small_buffer_d, red_d);
icudaMemcpy(red, red_d, sizeof(int), cudaMemcpyDeviceToHost);
*a = red;
cudaFree(buffer_d);
cudaFree(small_buffer_d);
cudaFree(b_d);
```
The reduction code in optimized CUDA

template<class T>
struct SharedMemory
{
  __device__ inline operator T*()
  {
    extern __shared__ int __smem[];
    return (T*)__smem;
  }

  __device__ inline operator const T*() const
  {
    extern __shared__ int __smem[];
    return (T*)__smem;
  }
};

template <class T, unsigned int blockSize, bool nlsPow2>
__global__ void
reduce6(T* __g_idata, T* __g_odata, unsigned int n)
{
  T *sdata = SharedMemory<T>();

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockSize + threadIdx.x;
  unsigned int gridSize = blockSize*gridDim.x;

  T mySum = 0;
  while (i < n)
    {
      mySum += __g_idata[i];
      if (nlsPow2 || i + blockSize < n)
        mySum += __g_idata[i + blockSize];
      i += gridSize;
    }
  sdata[tid] = mySum;
  __syncthreads();

  if (blockSize >= 512) {
    if (tid < 256) {
      sdata[tid] = mySum = mySum + sdata[tid + 256];
    }
    __syncthreads();
  }
  if (blockSize >= 256) {
    if (tid < 128) {
      sdata[tid] = mySum = mySum + sdata[tid + 128];
    }
    __syncthreads();
  }
  if (blockSize >= 128) {
    if (tid < 64) {
      sdata[tid] = mySum = mySum + sdata[tid + 64];
    }
    __syncthreads();
  }
}

if (tid < 32)
{
  volatile T* smem = sdata;
  if (blockSize >= 64) {
    smem[tid] = mySum = mySum + smem[tid + 32];
  }
  if (blockSize >= 32) {
    smem[tid] = mySum = mySum + smem[tid + 16];
  }
  if (blockSize >= 16) {
    smem[tid] = mySum = mySum + smem[tid + 8];
  }
  if (blockSize >= 8) {
    smem[tid] = mySum = mySum + smem[tid + 4];
  }
  if (blockSize >= 4) {
    smem[tid] = mySum = mySum + smem[tid + 2];
  }
  if (blockSize >= 2) {
    smem[tid] = mySum = mySum + smem[tid + 1];
  }
}

if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];

extern "C" void reduce6_cuda(int n, int *a, int *b)
{
  int *b_d;
  const int b_size = *n;

  cudaMalloc((void**)&b_d, sizeof(int)*b_size);
  cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

  dim3 dimBlock(128, 1, 1);
  dim3 dimGrid(128, 1, 1);
  int smemSize = 128 * sizeof(int);
  int *buffer_d;
  int small_buffer[4], *small_buffer_d;

  cudaMalloc((void**)&buffer_d, sizeof(int)*128);
  cudaMemcpy(buffer_d, small_buffer, sizeof(int), cudaMemcpyHostToDevice);

  cudaMemcpy(small_buffer_d, small_buffer, sizeof(int), cudaMemcpyDeviceToHost);

  cudaMemcpy(small_buffer_d, b_d, sizeof(int)*b_size, cudaMemcpyDeviceToHost);
  cudaMemcpy(small_buffer_d, buffer_d, sizeof(int)*128, cudaMemcpyHostToDevice);

  *a = small_buffer;
### Compiler does the work:
- Identifies parallel loops within the region
- Determines the kernels needed
- Splits the code into accelerator and host portions
- Workshares loops running on accelerator
  - Make use of MIMD and SIMD style parallelism
- Data movement
  - allocates/frees GPU memory at start/end of region
  - moves of data to/from GPU

```c
!$acc data present(a,b)
!$acc parallel

a = 0.0

!$acc loop reduction(+:a)

do i = 1,n
  a = a + b(i)
end do

!$acc end parallel
!$acc end data
```
CLE - An Adaptive Linux OS designed specifically for HPC

**ESM – Extreme Scalability Mode**
- No compromise scalability
- Low-Noise Kernel for scalability
- Native Comm. & Optimized MPI
- Application-specific performance tuning and scaling

**CCM – Cluster Compatibility Mode**
- No compromise compatibility
- Fully standard x86/Linux
- Standardized Communication Layer
- Out-of-the-box ISV Installation
- ISV applications simply install and run

**CLE run mode is set by the user on a job-by-job basis to provide full flexibility**
Exa’s PowerFLOW Supports Cray XE6 Supercomputers with Excellent Scalability to Thousands of Cores

Exa® Corporation, a global innovator of fluids simulation solutions for product engineering, today announced the release of PowerFLOW® 4.3c with support for the Cray XE6 line of supercomputers. This newest release of PowerFLOW allows customers to minimize their simulation turnaround time by scaling to a larger number of cores than any other platform.

“With PowerFLOW on a Cray XE6 or a Cray XE6m supercomputer, our customers can run simulations on thousands of cores to radically reduce turnaround times and answer key design questions quickly without sacrificing accuracy,” remarked Chuck Alexander, Exa’s Director of Product Management for Simulation Products.

Burlington, MA (PRWEB) October 18, 2011
As part of the performance testing with a native version of Fluent has shown scalability to 3000 cores even with this modest size model. Increasingly ISV codes are demonstrating “extreme scalability”.

Fluent external aerodynamics simulation: truck_111m

![Graph showing Fluent rating vs. NCores for 13.0 Native (Beta) and 14.0 Native (Beta)]
## Cray status of key ISV applications

<table>
<thead>
<tr>
<th>ISV Application</th>
<th>Cray target segment</th>
<th>CLE status</th>
<th>Demonstrated scalability</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUENT</td>
<td>Commercial CFD</td>
<td>CCM</td>
<td>~1000 cores</td>
<td>GE pushing vendor for better Cray performance, native version scaling much higher and likely will be offered</td>
</tr>
<tr>
<td>LS-DYNA</td>
<td>Impact/crash analysis</td>
<td>Native</td>
<td>512 cores</td>
<td>Widely used across all industries. Key at DoD sites.</td>
</tr>
<tr>
<td>CFD++</td>
<td>Aero CFD</td>
<td>Native</td>
<td>&gt;1000 cores</td>
<td>Working with Swift and ISV to demonstrate scalability.</td>
</tr>
<tr>
<td>STAR-CCM+</td>
<td>Commercial CFD</td>
<td>Native</td>
<td>512 cores</td>
<td>Widely used code</td>
</tr>
<tr>
<td>PowerFLOW</td>
<td>External CFD</td>
<td>CCM</td>
<td>&gt;3000 cores</td>
<td>HLRS/Porsche push for good performance</td>
</tr>
<tr>
<td>RADIOSS</td>
<td>Impact/Crash analysis</td>
<td>CCM</td>
<td>&gt;512 cores</td>
<td>ISV is promoting scalability</td>
</tr>
<tr>
<td>PAM-CRASH</td>
<td>Impact/crash analysis</td>
<td>CCM</td>
<td>&gt;256 cores</td>
<td>HLRS is pushing for improved Cray performance.</td>
</tr>
<tr>
<td>ABAQUS</td>
<td>Structural analysis</td>
<td>CCM</td>
<td>&gt;64 cores</td>
<td>ORNL working with ISV to improve performance.</td>
</tr>
</tbody>
</table>
Cray Software Ecosystem

- **CrayPETSc**
- **CASK**
- **CrayPAT** (Cray Apprentice)
- **Iterative Refinement Toolkit**
- **Cray PETSc, CASK**

**COMPILERS**
- PGI
- PathScale
- **CRAY** (Cray The Supercomputer Company)

**APPLICATIONS**
- LSTC
-.accelrys
- CD-adapco
- The MathWorks
- DS SIMULIA
- **CEI**
- **Altair**

**DEBUGGING**
- allinea
- TotalView Technologies

**PERFORMANCE TOOLS**
- ParaTools

**JOB MANAGEMENT**
- Platform Computing
- Adaptive Computing

**IO & LIBRARIES**
- nag
NodeKARE goal: Improve percentage of jobs that run successfully by ensuring that all nodes are healthy at job launch time

- If a job fails, NodeKARE runs diagnostics on the partition looking for sources of error
- Performs file system checks, memory usage, application termination, site-specific check
- Configurable: when to run, what to do on errors, callout to site-specific script
- Future release will dump and restart downed nodes
Storage
Interlagos
Interlagos Processor Architecture

- Composed of a # of “core modules” (CM)
  - A core module has shared and dedicated components

- There are two independent integer schedulers and a shared 2x128-bit FP resource
  - A single “thread” can use the entire FP resource

- New AVX instruction set
  - AVX = Advanced Vector eXtensions
  - Both 128 and 256 bit “vector length” instr.

- Boost technology

- This architecture is very flexible, and can be applied effectively to a variety of workloads and problems
Building an Interlagos Processor

- G34 Socket is compatible with XE6 board and contains
  - 8 core modules
  - 16 MB L3 Cache
  - 4 DDR3 1600 memory channels
One thread is per core module

- MPI rank has *exclusive* access to the 2x128-bit FP unit and is capable of 8 FP results per clock cycle
- Maximize memory/core and memory/rank
- Larger L2/L3 cache per MPI rank
- The peak of the chip is not reduced
- Better with more *vectorized* and makes use of AVX instructions

% aprun -n xxx -d 2 a.out
MPI rank is pinned to each integer unit
- Each unit has exclusive access to an integer scheduler, integer pipelines and L1 Dcache
- The 2x128-bit FP unit and the L2 Cache is shared between units
- AVX instructions are dynamically executed as two 128-bit instructions utilizing either or both FP unit
- Best for highly parallel, highly scalar applications

% aprun -n xxx -d 1 a.out
NPB – Class D MPI

• Modes give similar aggregate performance for most NPB apps at this scale
  • FT has difficulty scaling from 1024 to 2048 ranks and may have some destructive interference when using 2 ranks/CM and performs much better in 1 rank/CM mode
  • EP scales very will and performs better in 2 rank / CM mode

Higher is better