



A GPU –Enhanced Linux Cluster for Accelerated FMS

Computational Sciences



21 June 07
Gene Wagenbreth

genew@isi.edu

(310)448-8213



Background



Computational Sciences Division of ISI works with clusters, compilers and parallel hardware and software

Gene Wagenbreth has 35 years experience working with a wide range of parallel hardware and software

Supported JFCOM use of clusters for several years



JFCOM/JSAF



JFCOM is a leading large-scale battlefield simulation provider

Interactive use by over one hundred personnel common (HITL)

Many user/operators were and are uniformed warfighters

Use a scalable version of JSAF

Scales to hundreds or thousands of nodes

LAN of workstations

WAN connects to multiple compute clusters

Simulate hundreds of thousands to millions of entities

Used clusters at MHPCC and ASC



JFCOM Proposed a Need for a DHPI



Used HPCMP assets in the past but needed dedicated machine

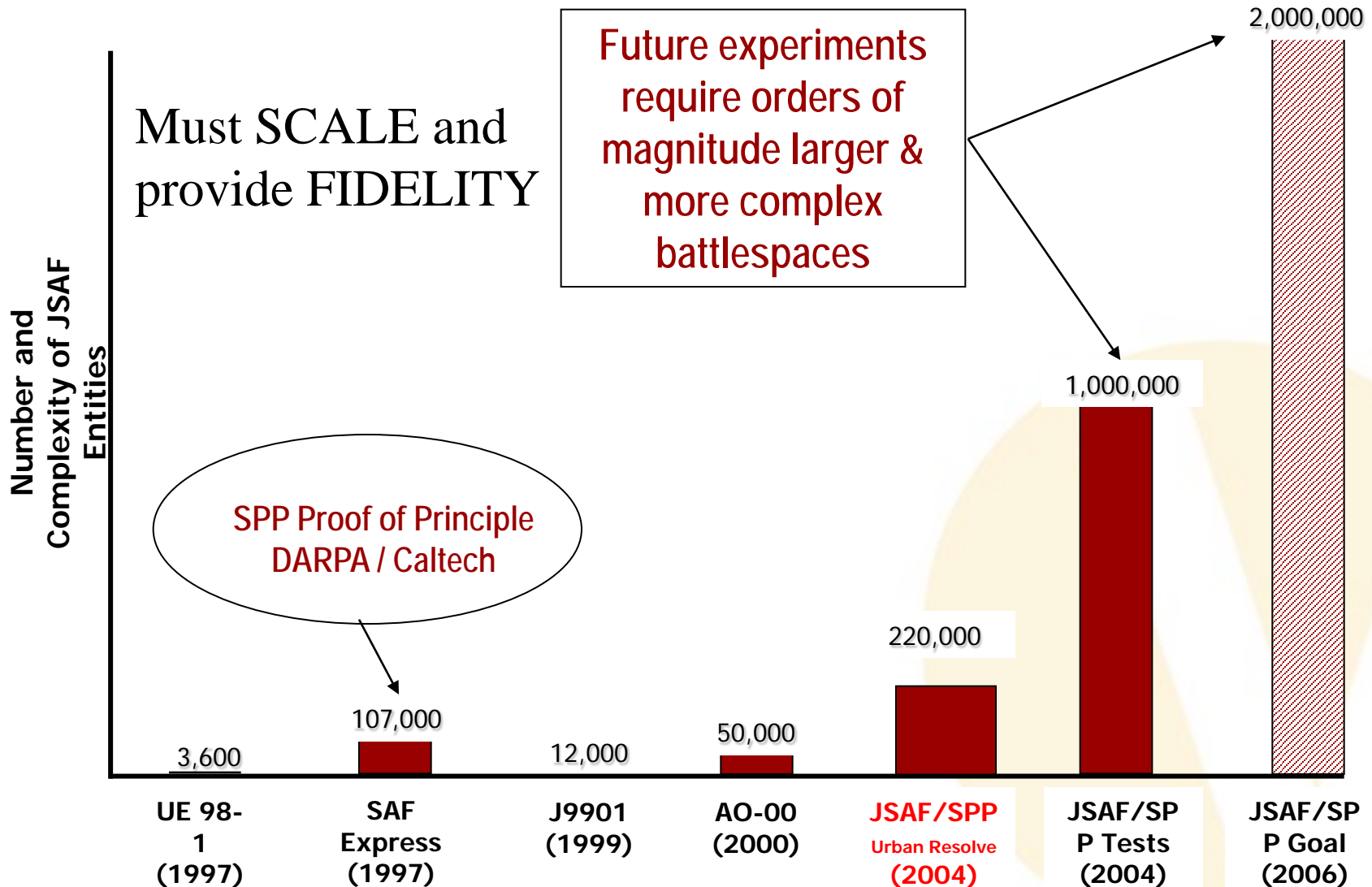
Proposal requested 1024 cores, plus 128 GPUs as accelerators

Submitted proposal on 03 April 07

Accepted by HPCMP on 30 November 06, award of LNXI cluster



Why JFCOM Needs HPC Power





Cluster Configuration as Awarded



256 Nodes

Processor (2) AMD Santa Rosa 2220 2.8 GHz dual-core processors, for a total of 4 cores per node

GPU (1) NVIDIA 7950 Video Card

Node Chassis 4U chassis

Memory 16 GB DIMM DDR2 667 per node – 4 GB per processor core

Motherboard 2 Processor Sockets, NVIDIA NPF3600 + NPF3050 with PCIExpress

GigE Internode Communications

Delivered to J7 Computer Room, Suffolk Virginia



Racks of the Linux Network Cluster for JFCOM



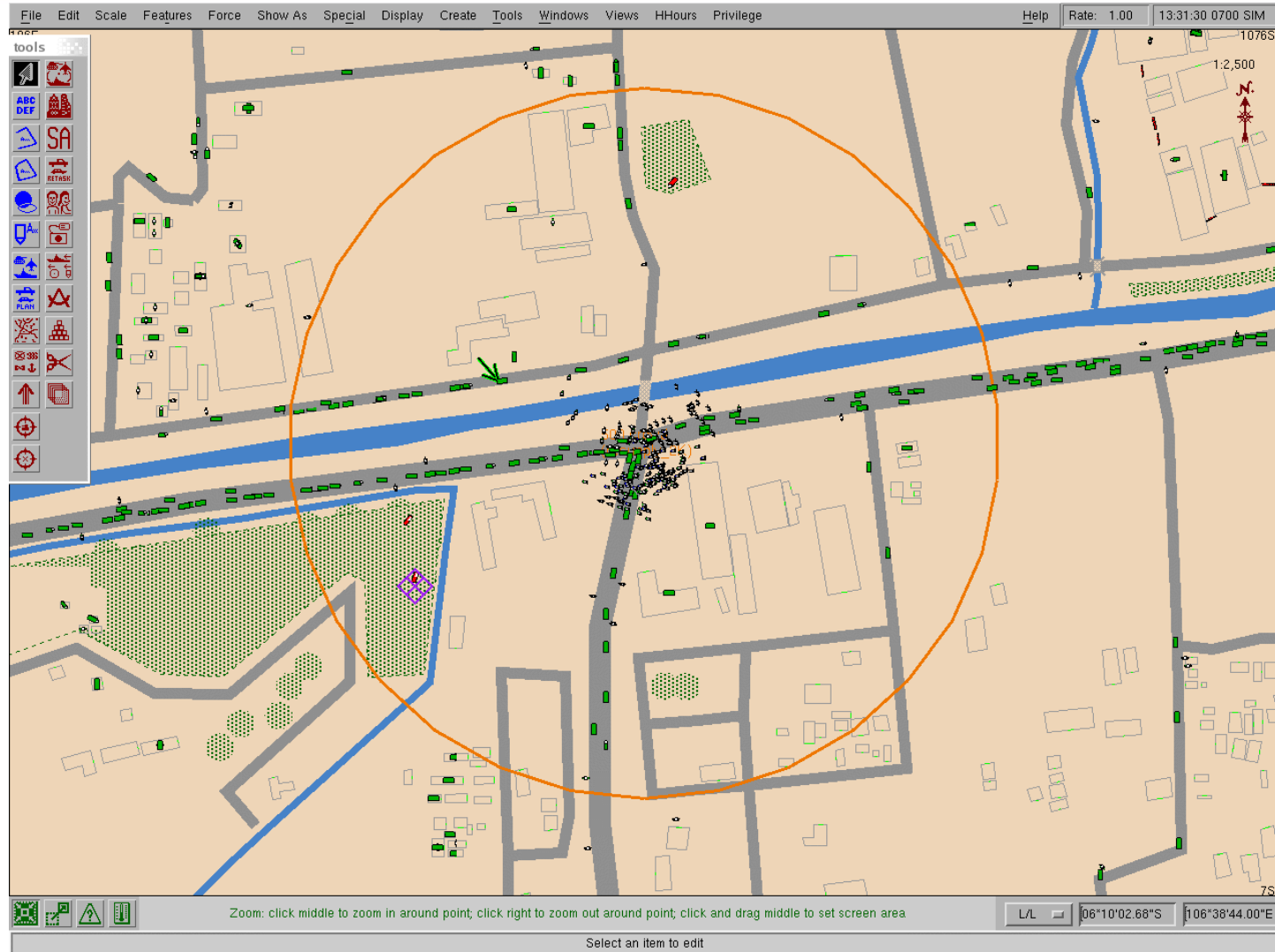
20 of 28
racks, shown
here at the
LNXI Shop in
Utah



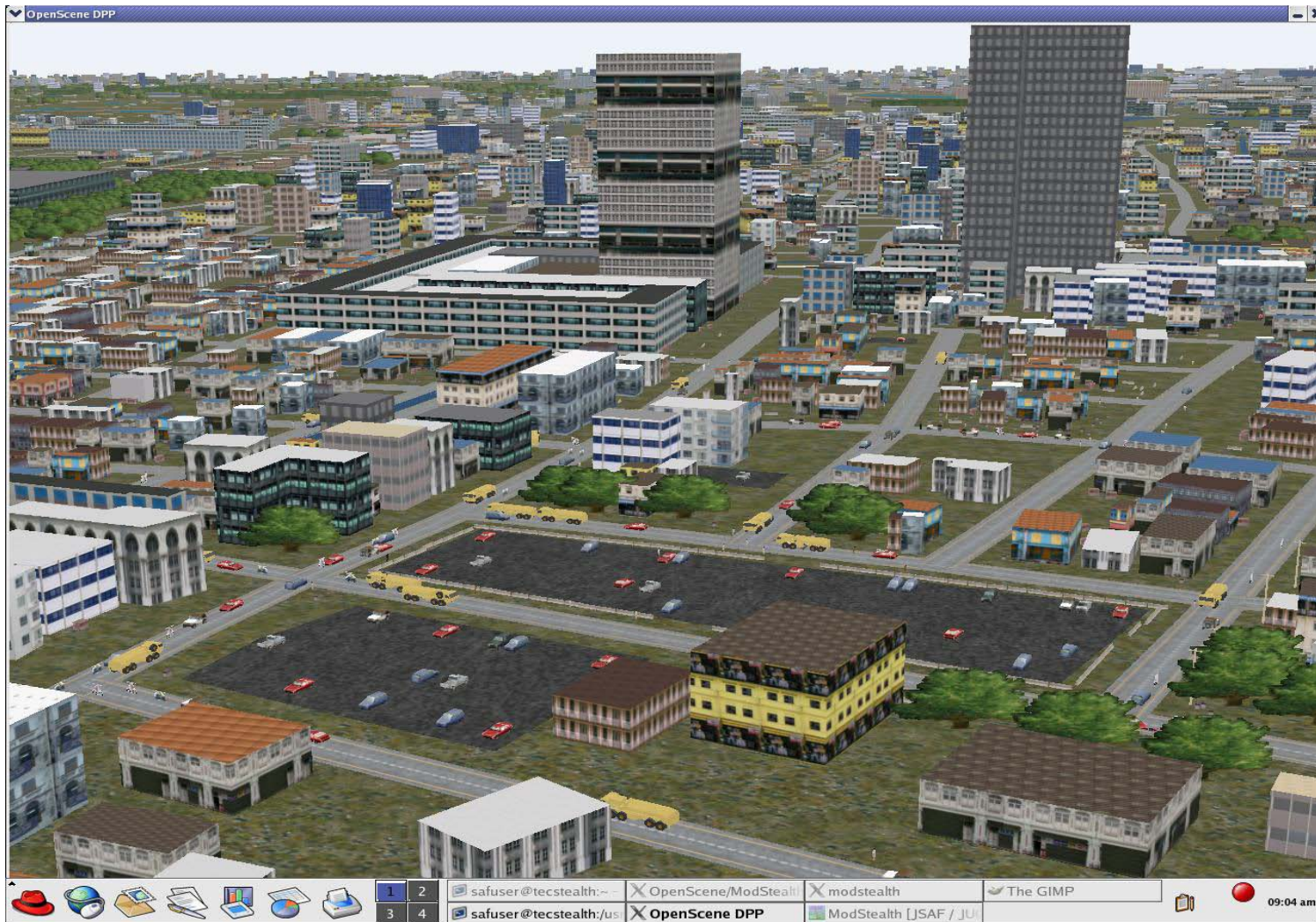
Front view of Management Rack



Plan View Display of Battlespace



3D Stealth View of Battlespace





Why GPUs



GPU performance can be 100X host performance. This differential is expected to grow.

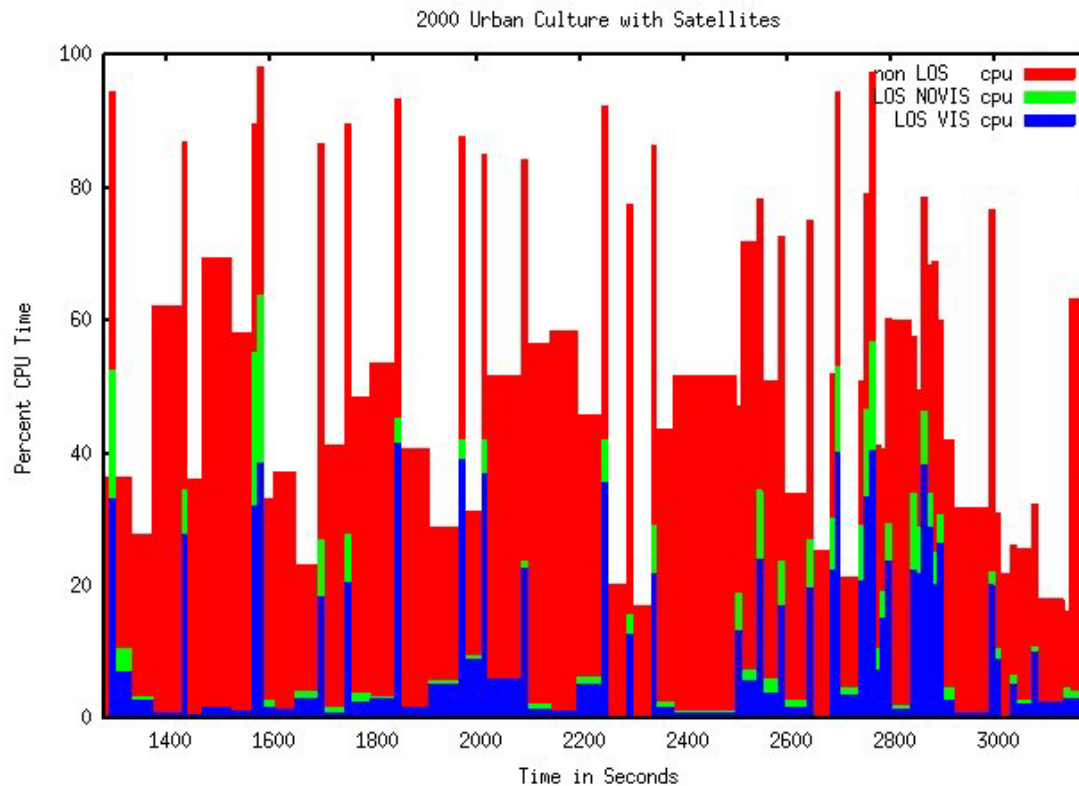
Line of Sight (LOS) and Route Finding algorithms identified by Dinesh Manocha (UNC) and others

ISI performed experiments to quantify CPU intensive algorithms in JSAF as candidates for conversion to GPU

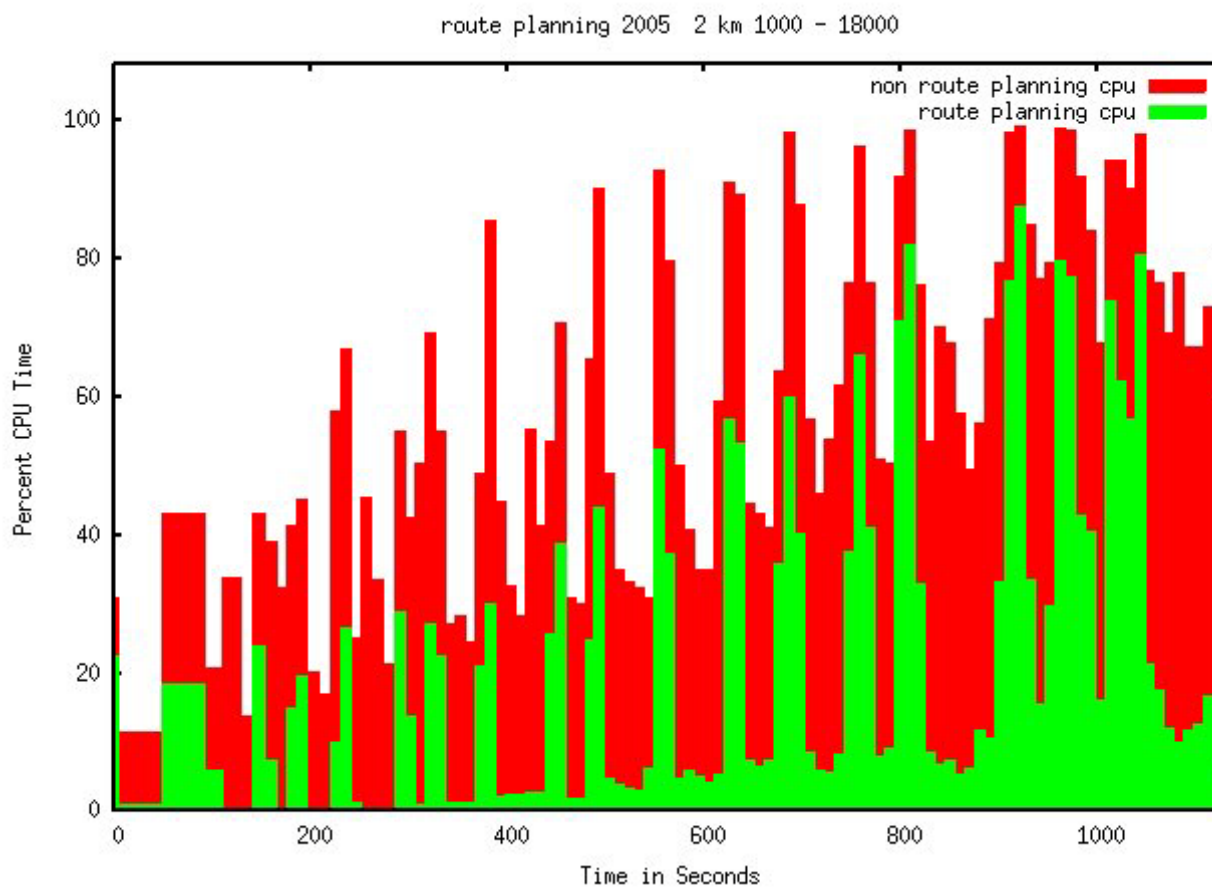
Measured performance of Line Of Sight (LOS) and Route Finding algorithms

Candidate algorithms use large amount of time in small amount of code to enable conversion

Instrumented JSAF to Measure time spent in LOS



Route Planning





Experiment Conclusions



Overall LOS and Route Planning do not use significant time

In bursts either can use 50% or more of CPU

Entity count could double if algorithms accelerated





NVIDIA GPU



Similar to CELL processor and other GPU's

Hundreds of GIGAFLOPS single precision performance. Up to 100X speedup over host

Performance differential expected to continue to grow

Efficient libraries for linear algebra, FFT

Supports CUDA





CUDA



High level language supported by NVIDIA for current and future architectures

No need to hand code low level language and rewrite every few years

C language with GPU specific extensions

Don't use OpenGL





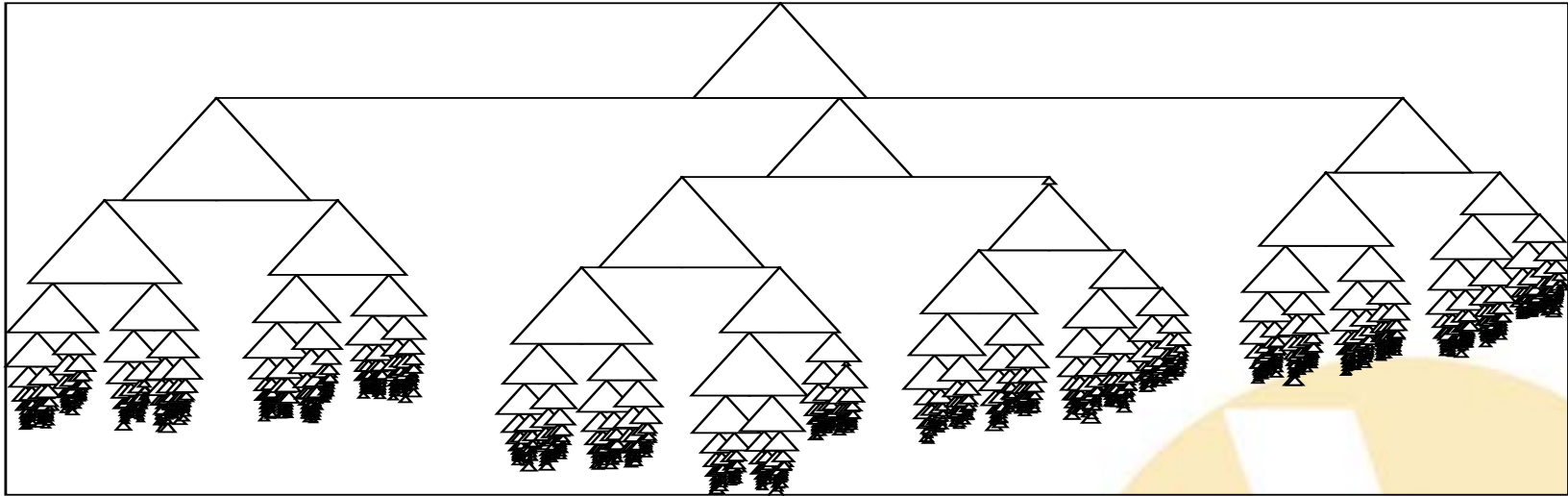
Sparse Matrix Solver



We coded a multifrontal sparse matrix solver for the NVIDIA GPU as a real world experiment in converting code.

Solves a large sparse matrix by processing a tree of dense matrices.

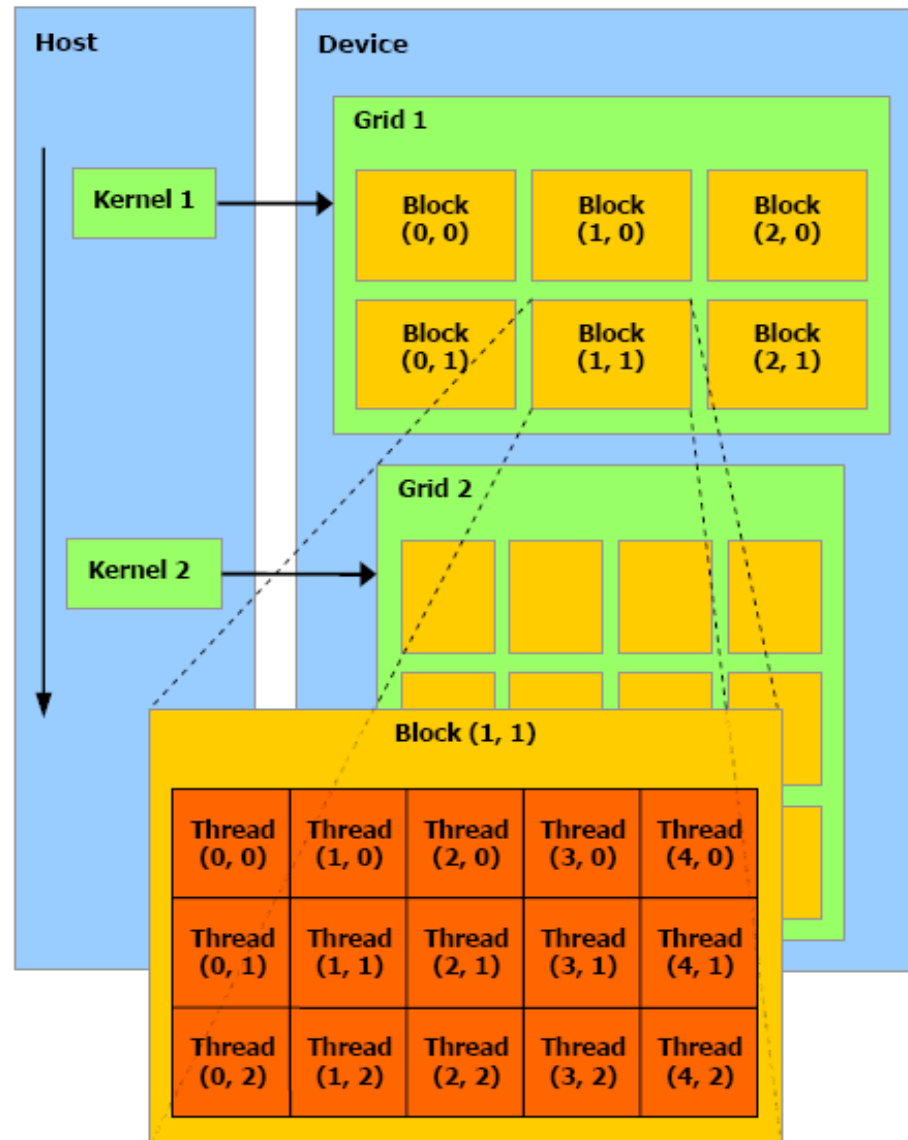




Each frontal matrix's triangle scaled by operations required to factor it.

Small matrices inefficient on GPU. Solved on host. Large matrices solved on GPU.

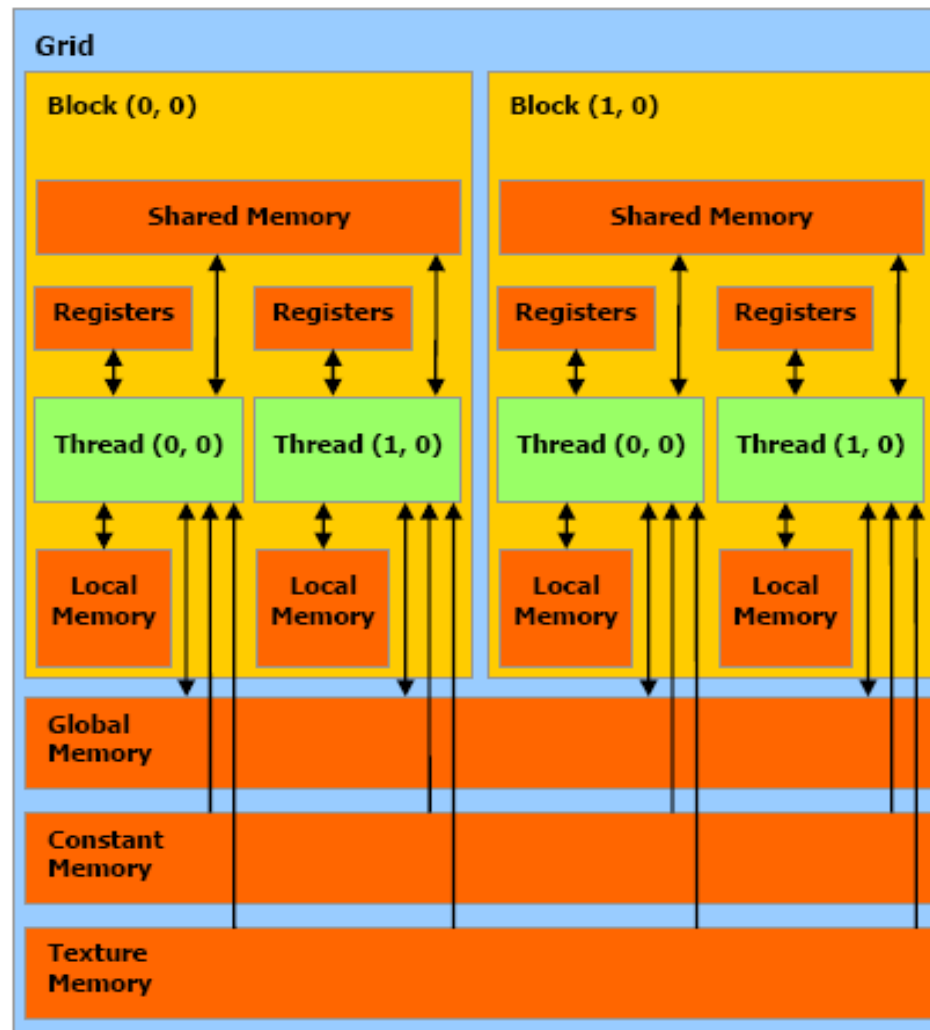
Thread Batching



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Figure 2-1. Thread Batching

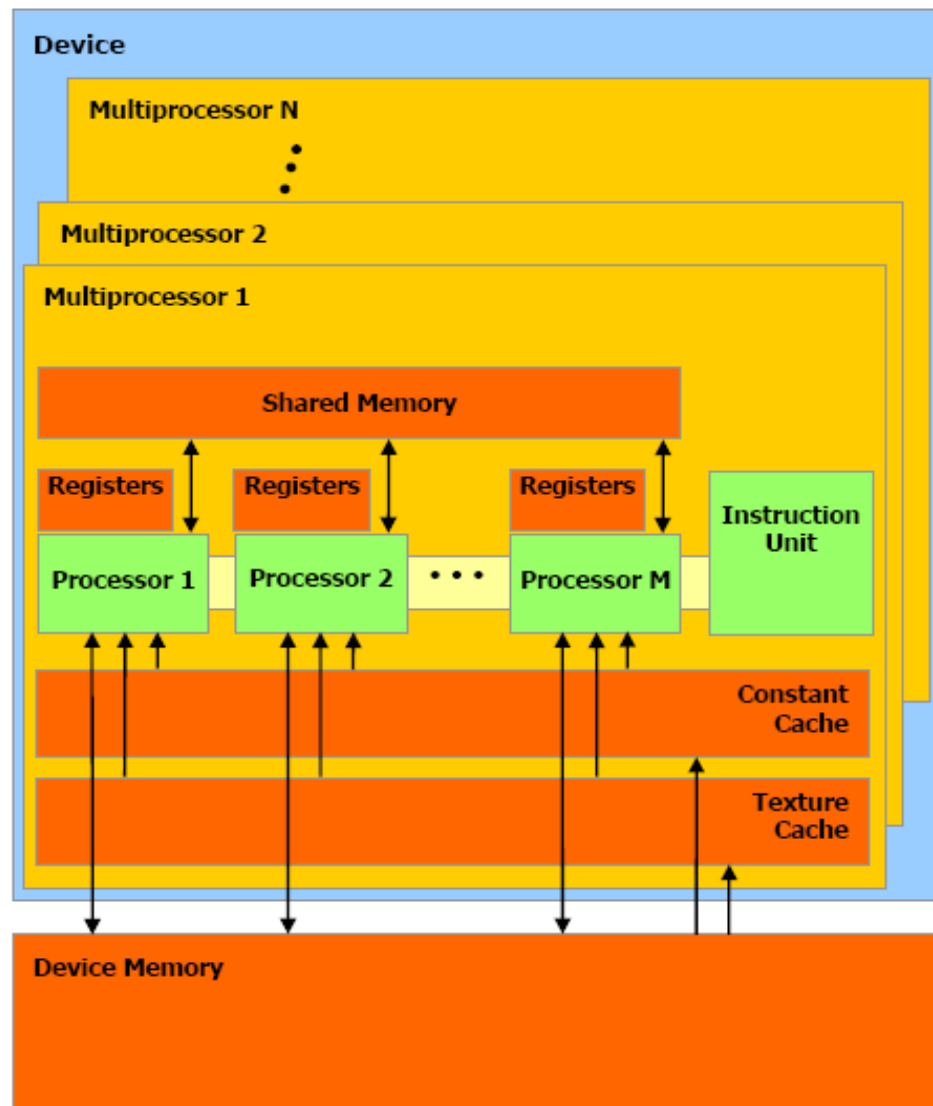
Memory Model



A thread has access to the device's DRAM and on-chip memory through a set of memory spaces of various scopes.

Figure 2-2. Memory Model

Hardware Model



A set of SIMD multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model

Original coded in FORTRAN. Converted to C on host.

Host C converted to GPU CUDA without parallelization to insure correctness.

CUDA code parallelized and tuned.

Efficiency requirements:

- many threads with identical code

- many operations on data transferred from host

- efficient use of GPU “shared memory”



Efficient Use of GPU Memory



host memory

device memory

shared memory – small (4K words per processor) user
managed cache

Transfer inputs from host to GPU device memory

Using 100s or 1000s of SIMD THREADS:

Loop

- Transfer small block from device memory to shared memory
- Compute from shared memory
- Transfer small block back to device memory

Transfer outputs from GPU device memory to host memory

Only use SGEMM library routine on GPU

SGEMM is very efficient

Other processing performed on host

Very little CUDA code required

Conversion was relatively easy

Overall code very inefficient due to memory transfers between host and GPU



Original Fortran => CUDA

```

do j = jl, jr
do i = jr + 1, ld
  x = 0.0
  do k = jl, j - 1
    x = x + s(i, k) * s(k, j)
  end do
  s(i, j) = s(i, j) - x
end do
do i = jr + 1, ld
  x = s(i, j) * s(j, j)
  s(j, i) = x
  max_coef = max(max_coef, abs(x))
end do
end do

```

```

ip=0;
for (j = jl; j <= jr; j++) {
  if(ltid <= (j-1)-jl){
    gpulskj(ip+ltid) = s[IDX(jl+ltid,j)];
  }
  ip=ip+(j-1)-jl+1;
}
__syncthreads();
for (i = jr + 1 + tid; i <= ld; i += LSTC_GPUL_THREAD_COUNT) {
  for (j = jl; j <= jr; j++) {
    gpuls(j-jl,ltid) = s[IDX(i,j)];
  }
  ip=0;
  for (j = jl; j <= jr; j++) {
    x = 0.0f;
    for (k = jl; k <= (j-1); k++) {
      x = x + gpuls(k-jl,ltid) * gpulskj(ip);
      ip=ip+1;
    }
    gpuls(j-jl,ltid) -= x;
  }
  for (j = jl; j <= jr; j++) {
    x = gpuls(j-jl,ltid) * s[IDX(j,j)];
    s[IDX(j,i)] = x;
    tls_max_coef[tid] = fmaxf(tls_max_coef[tid], fabsf(x));
  }
  for (j = jl; j <= jr; j++) {
    s[IDX(i,j)] = gpuls(j-jl,ltid);
  }
}

```

Use SGEMM library routine on GPU for most of the work

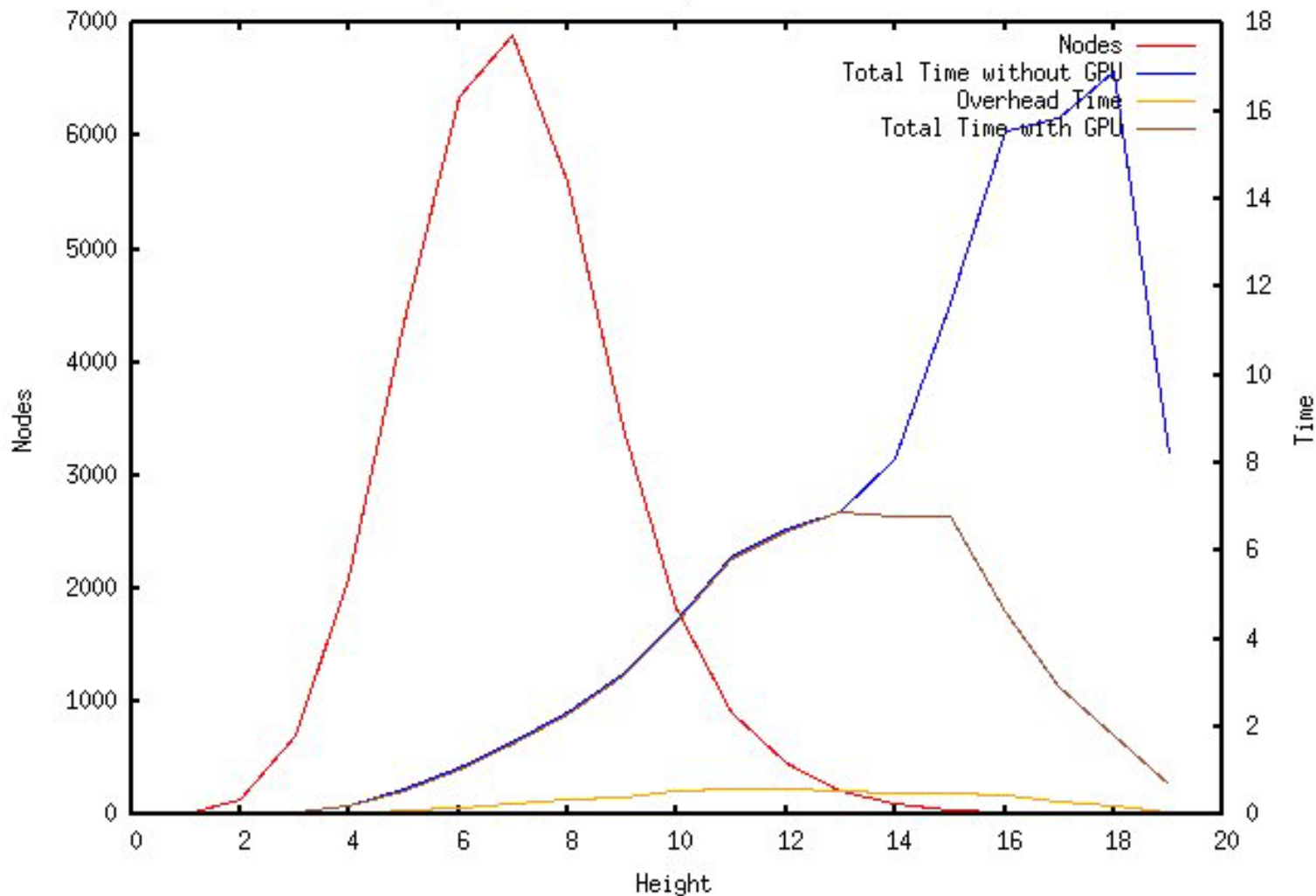
Convert a few hundred lines of non SGEMM C code to
CUDA

Parallelization of minor routines important. If done poorly
they run much slower than the host and kill performance.

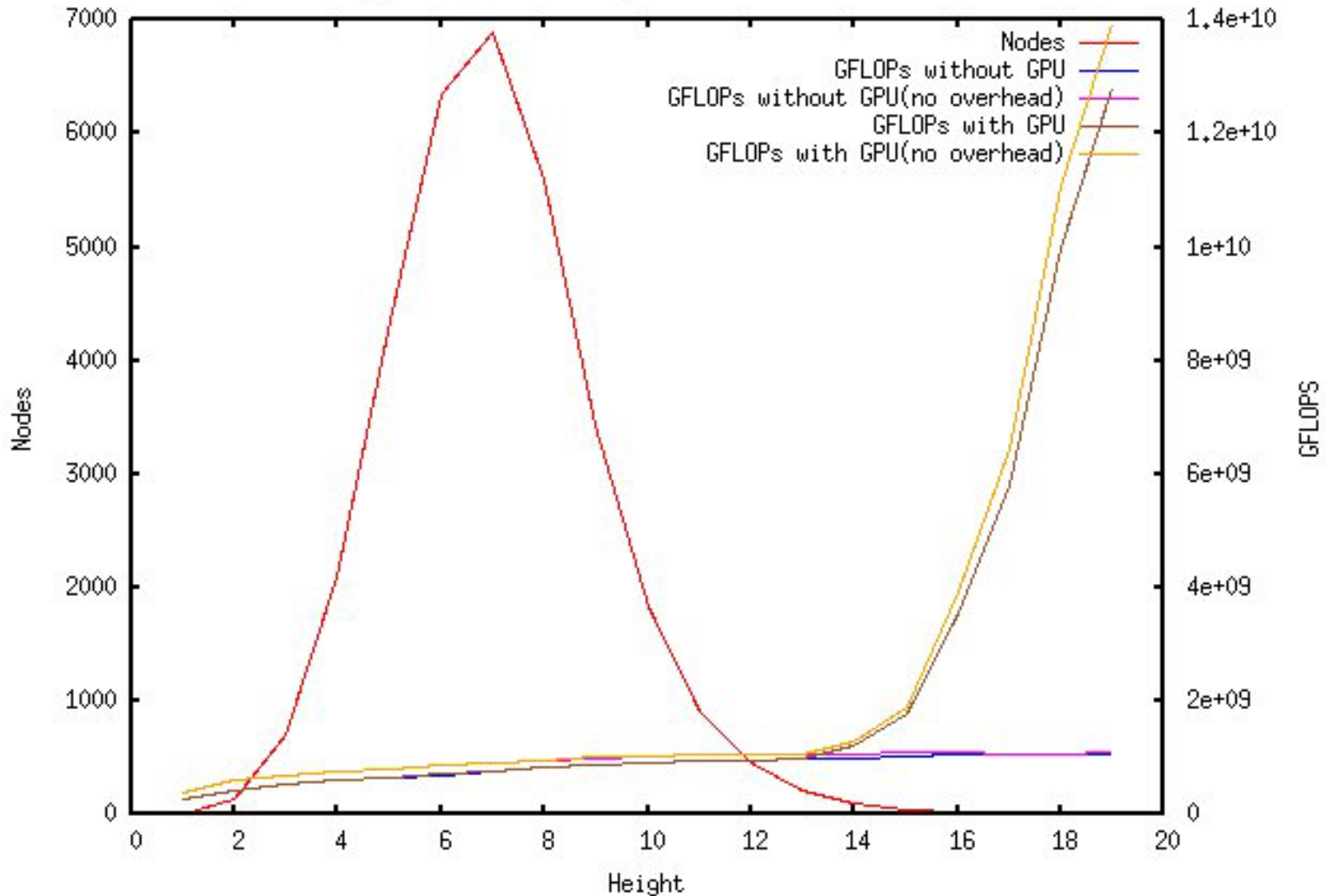
threads, shared memory – sometimes we use fewer threads
so there is space in shared memory for data.

For large matrices we ran at more than half the peak speed of
the GPU. Performance was 50 times performance on the host.

Time/Height Parameters for Sparse Matrix Factorization



Height Parameters for Sparse Matrix Factorization





Sparse Matrix Performance Summary



A small number of large matrices were solved on the GPU

GPU reduced total time from 120 seconds to 60 seconds

GPU was busy for 5 seconds out of 60, but reduced execution time by a factor of 2

GPU performed half of the computational work



Sparse Matrix Future Improvements



Elimination of system and user overhead would allow smaller matrices to be solved on the GPU, increasing speedup

Better use of device memory to allow larger matrices on GPU

Parallelize small matrices by doing more than one at a time

Use multiple GPUs

Integrate into commercial software





**“And now,
a message from our sponsor.”**



This material is based on research sponsored by the Air Force Research Laboratory under agreement numbers F30602-02-C-0213 and FA8750-05-2-0204. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation appearing thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.