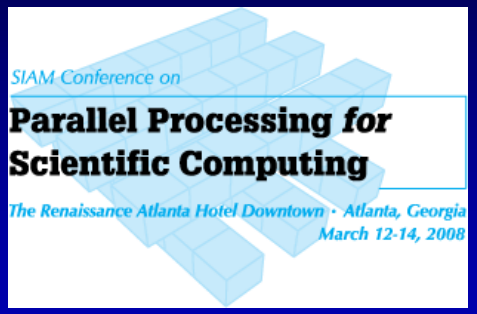


GPU-Enhanced Linear Solver Results

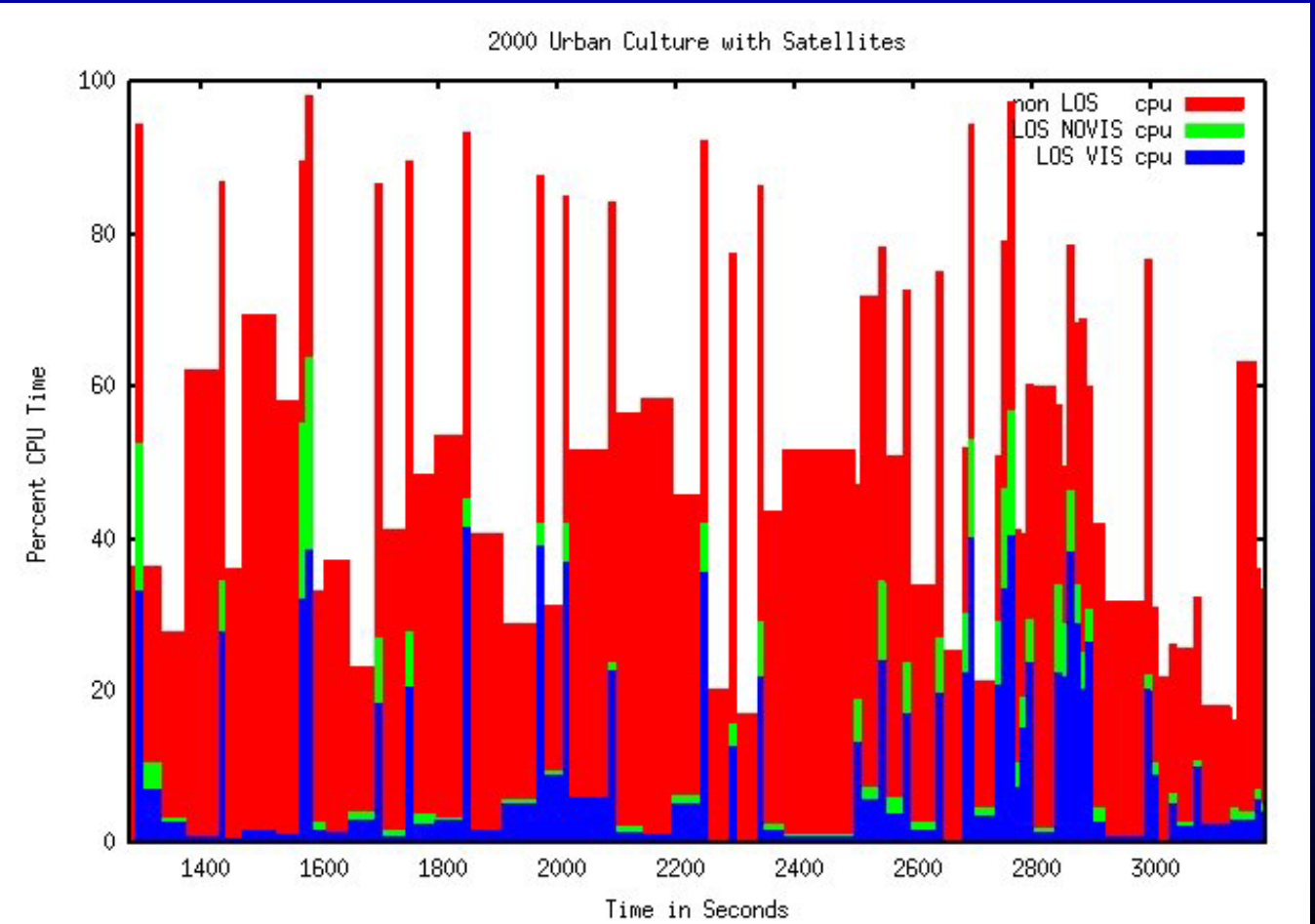
Robert F. Lucas
Information Sciences Institute
University of Southern California
rflucas @ isi.edu

Why GPUs

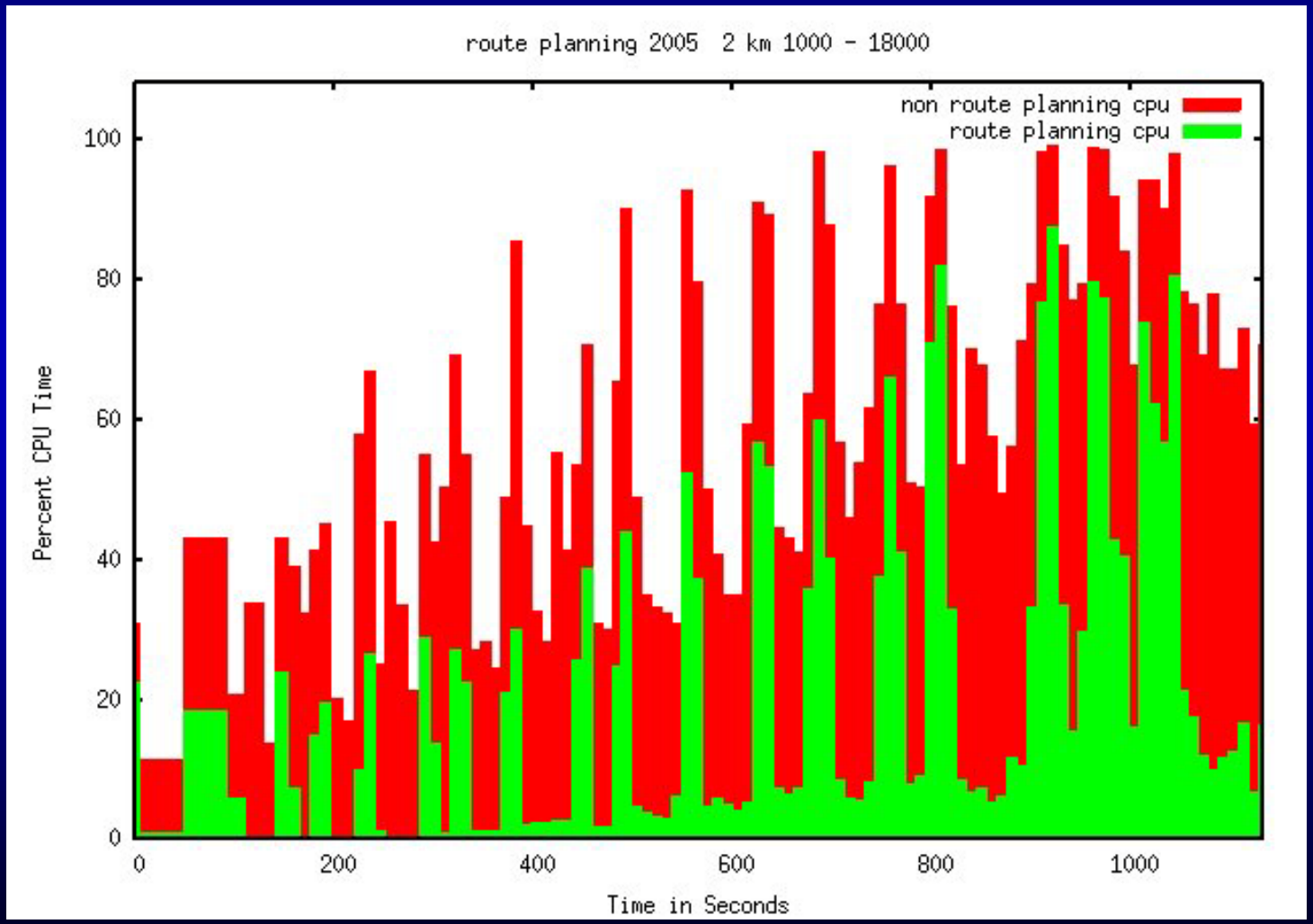
- **GPU performance can be 100X host performance. This differential is expected to grow.**
- **Several algorithms identified by Dinesh Manocha (UNC) and others as amenable to GPU acceleration**
- **ISI performed experiments to quantify CPU-intensive algorithms as candidates for conversion to GPU**
- **Measured performance of Line Of Sight (LOS) and Route Finding algorithms in agent based simulations**
- **Candidate algorithms use large amount of time in small amount of code to enable conversion**

Line of Sight

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**
- **GT** 8800 Series supports CUDA**

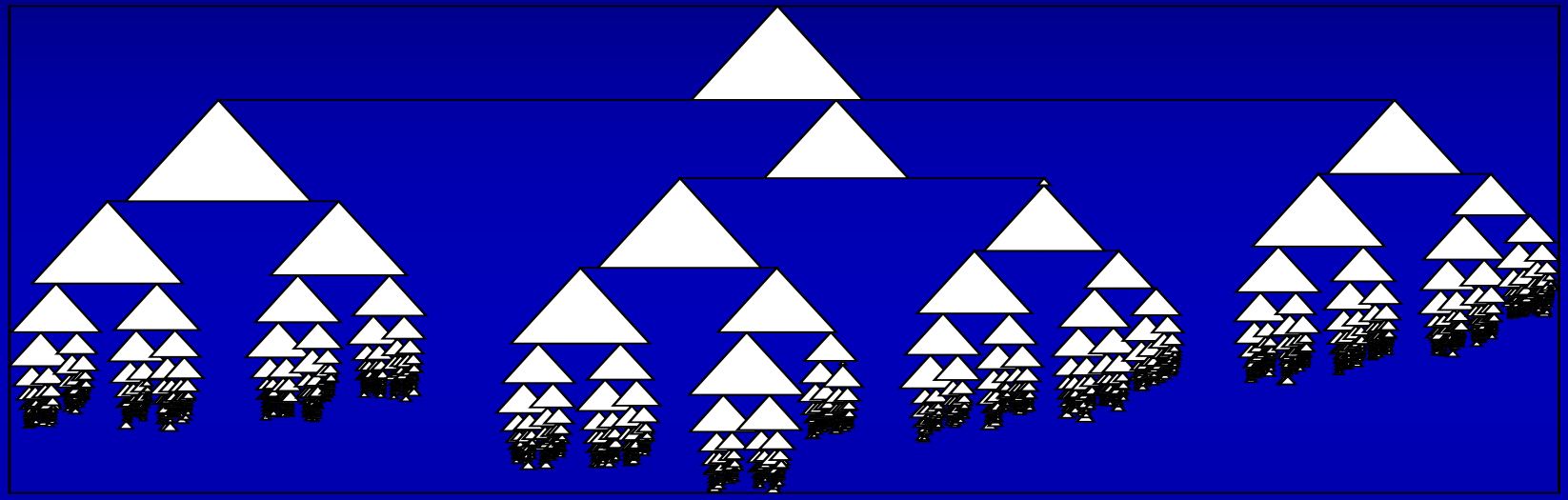
CUDA

- **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.**

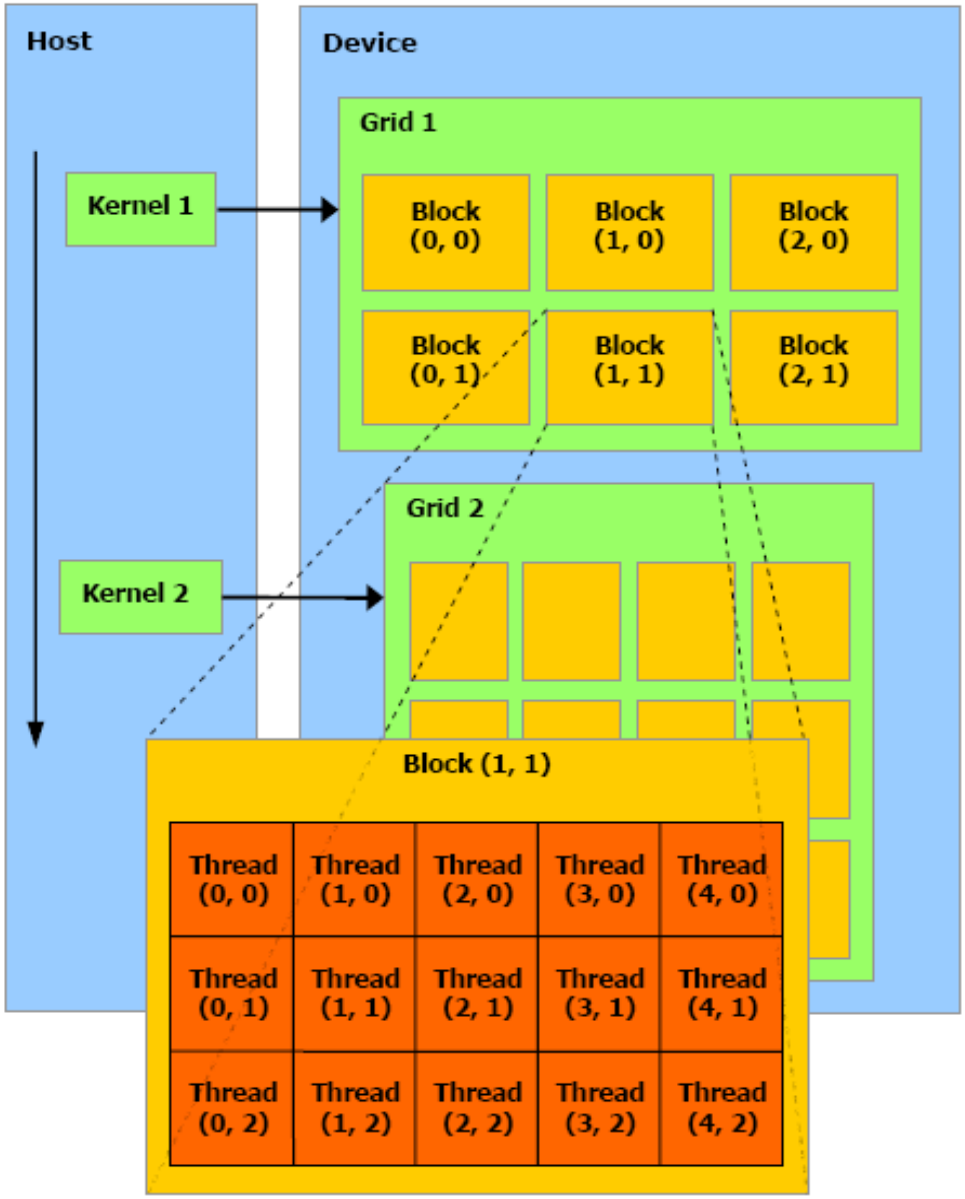
“Hood” Elimination Tree



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.

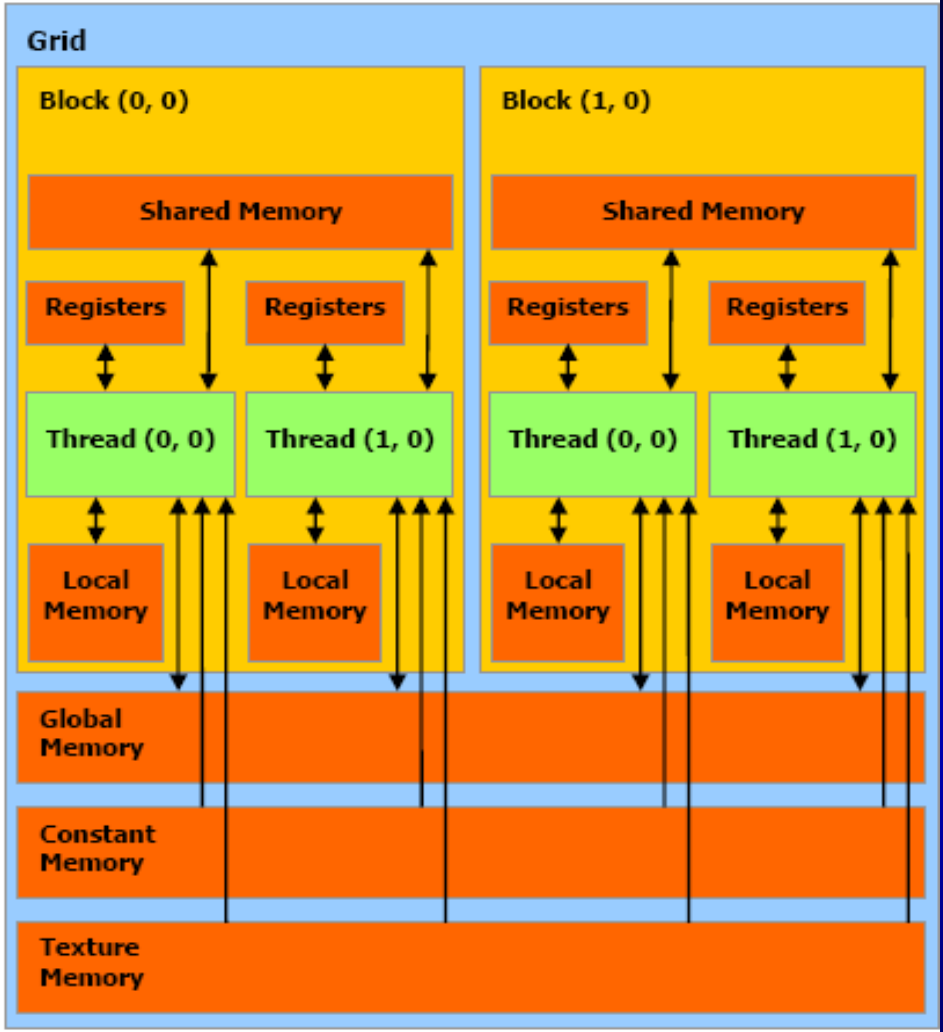
Thread Batching



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

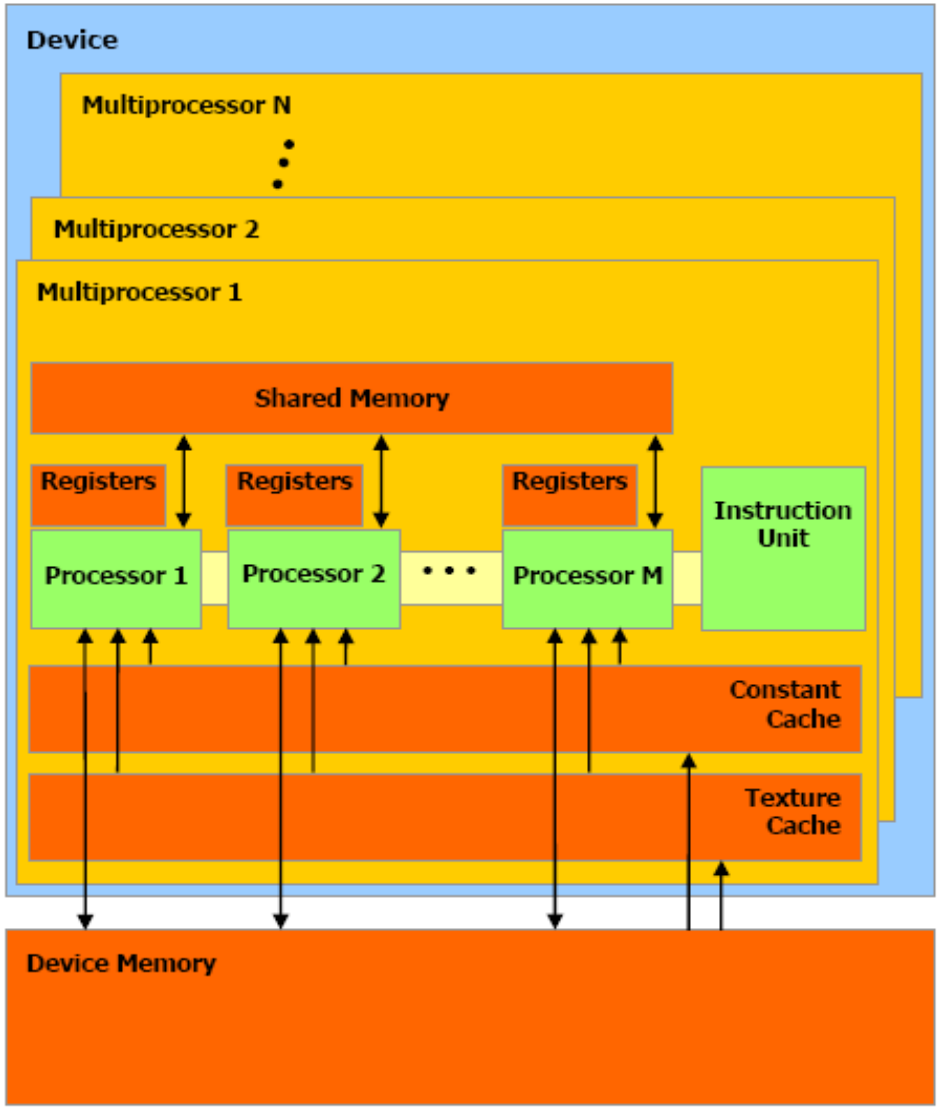
Memory Model



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

Code Conversion

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

First Attempt

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

Some CUDA Code

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[IDXS(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[IDXS(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[IDXS(j,j)];
        s[IDXS(j,i)] = x;
        tls_max_coef[tid] = fmaxf(tls_max_coef[tid], fabsf(x));
   }
   for (j = jl; j <= jr; j++) {
       s[IDXS(i,j)] = gpuls(j-jl,ltid);
   }
  }
```

Second Attempt

Use SGEMM library routine on GPU for most of the work

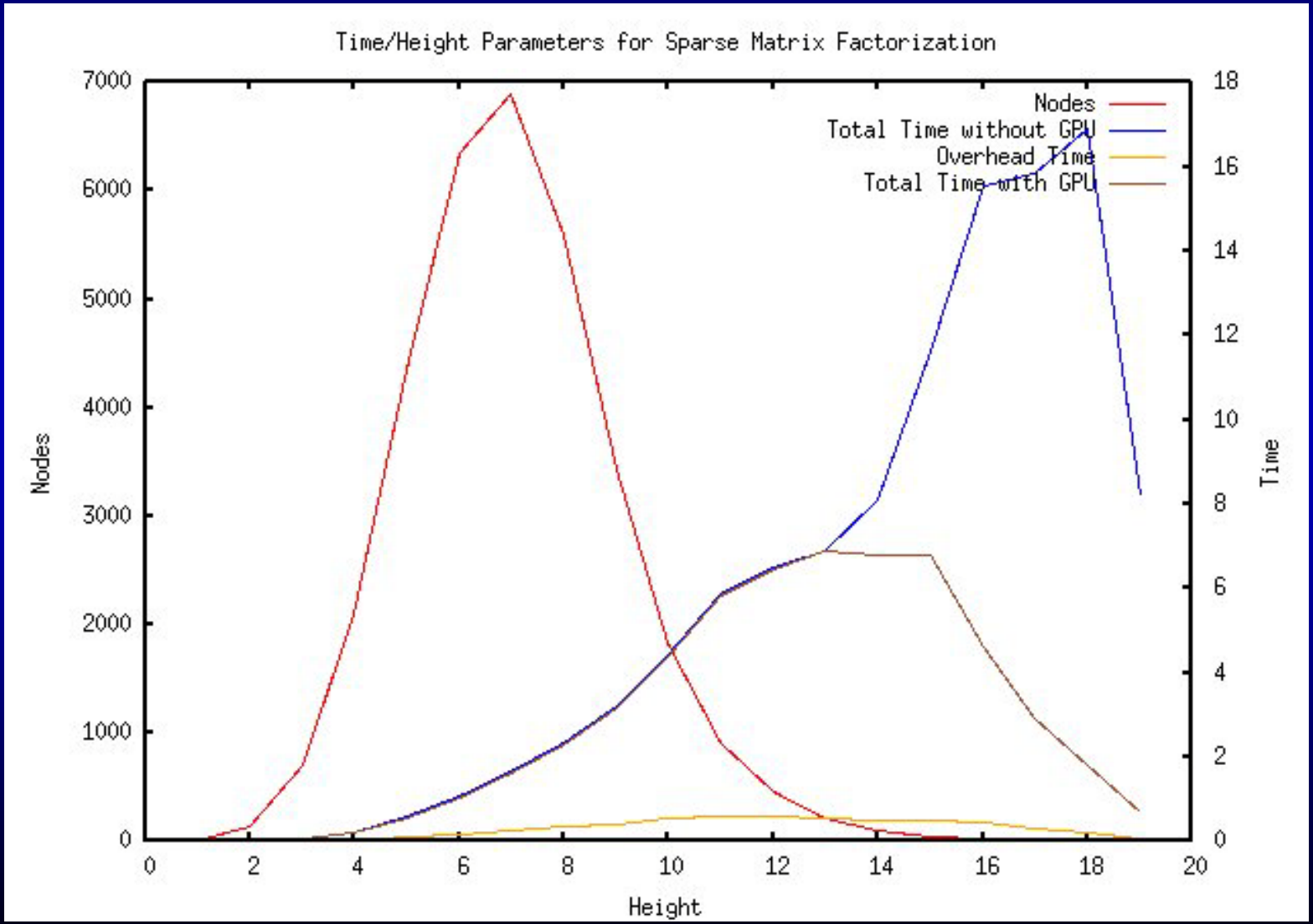
Converted a few hundred lines of non-SGEMM C code to CUDA

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

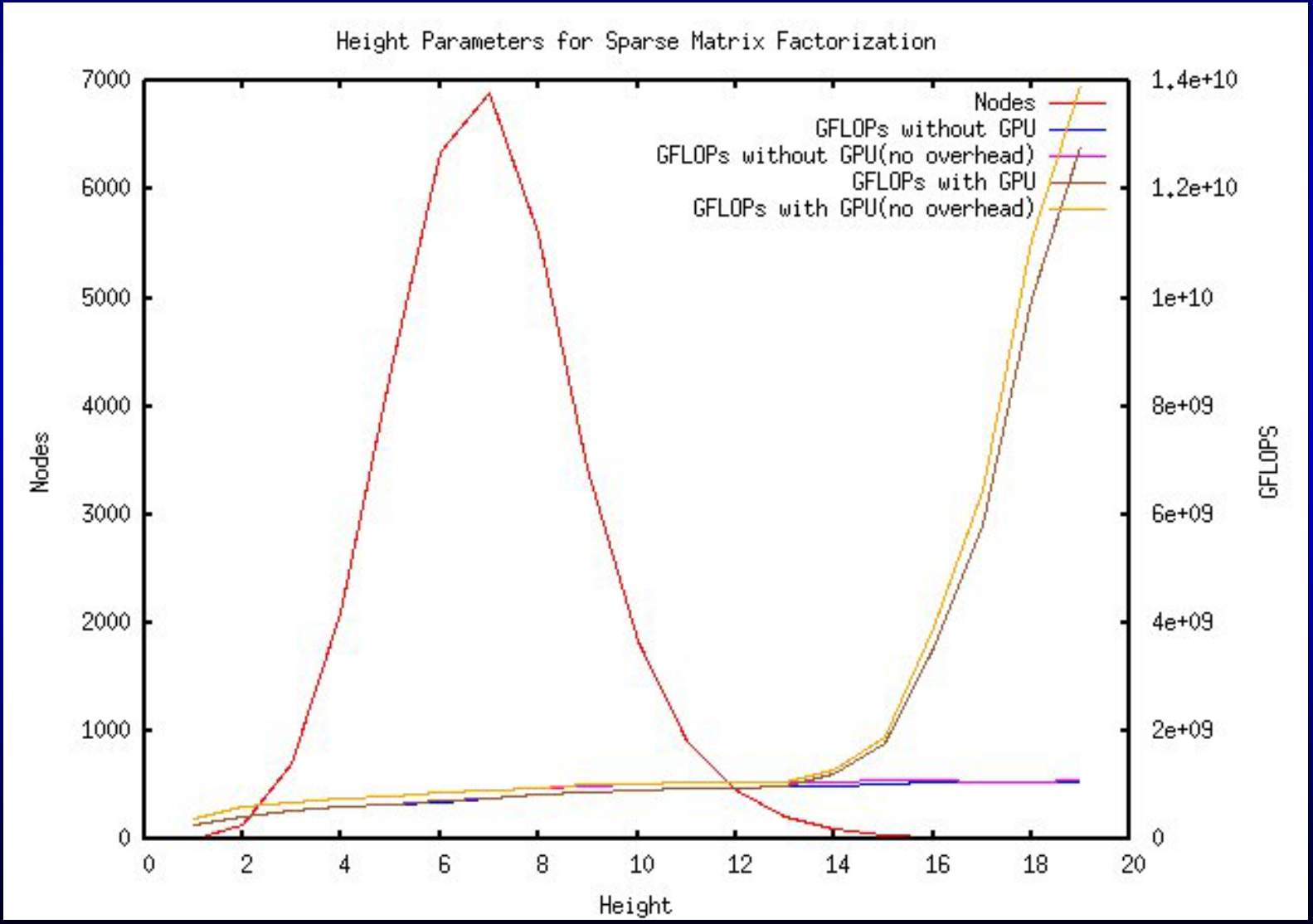
Threads, shared memory – sometimes usee fewer threads so there is space in shared memory for data.

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

Time Spent with GPU



Performance with GPU



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