ACCELERATING THE PRODUCTION OF SYNTHETIC SEISMOGRAMS BY A MULTICORE PROCESSOR CLUSTER WITH MULTIPLE GPUs

Ferdinando Alessi  Annalisa Massini  Roberto Basili
Introduction

- The simulation of **wave propagation** represents a typical context demanding massive computation.

- The **COMPSYN** package is a set of applications
  + designed for producing **synthetic seismograms**
  + used in evaluating seismic hazard and risk.

- **COMPSYN** is **time consuming**.

- We explore the possibility of accelerating COMPSYN by using a cluster of multicore processors with multiple GPUs.

- P. Spudich and L. Xu *Documentation of software package COMPSYN sxv3.11: programs for earthquake ground motion calculation using complete 1-D Greens functions*
  International Handbook of Earthquake and Engineering Seismology, 2002
Motivations

The study of **seismic wave propagation** allows
- to formulate models describing the **Earth surface motion**
- to foresee the **consequences of earthquakes**

**comparison to verify the accuracy of a model**

- **synthetic seismograms** produced by a simulation
- **real seismograms** obtained from seismometers
COMPSYN

three input files

- *.old
  Earth velocity structure
- *.tfd
  - fault geometry definition
  - observer locations
- *.sld
  earthquake slip distribution

five applications

- OLSON
  calculates Green’s functions in the frequency and wavenumber domain for laterally homogeneous velocity structures
- XOLSON
- TFAULT
  computes Green’s functions evaluated for surface observers as functions of position by Bessel-transforming the OLSON output rearranged by XOLSON
- SLIP
- SEESLO
Parallelization

- **TFAULT module** has been selected for parallelization:
  - **TFAULT** is one of the most time-consuming module
  - **TFAULT** can be required to run many times

**Parallelization of TFAULT**

- Translation from Fortran to C
- Analysis of the code of TFAULT and its structure
- Parallelization on the **host side**
- Parallelization on the **device side**: GPU

**Language**

- OpenMP API to exploit the power of **multicore processors**
- MPI to allow the execution on the **cluster**
- CUDA to develop the parallelization for **GPU**
Determining the kernel

TFAULT computes the tractions on a grid of points on the fault plane, for each observer. The structure of TFAULT is highly parallel and consists of five nested loops:

1. **Loop 1**: Loop over the observers.
2. **Loop 2**: Loop over the frequencies.
3. **Loop 3**: Loop over the lines of constant depth on the fault surface.
4. **Loop 4**: Loop over the sample points on lines of constant depth on the fault surface.
5. **Loop 5**: Loop over the wavenumbers.
Determining the kernel

✔ Loop 4 is the best candidate for becoming the kernel

❌ Loop 5 presents dependencies among iterations

❌ Loop 1 and Loop 2 are too complex

❌ Loop 3 is simpler
  but has too few iterations to harness the GPUs
  and presents many memory accesses

Loop 4 has
  - a number of iterations greater than other loops
  - but not great enough to fully exploit the GPU capabilities
  - then we group more Loops 4 in the same GPU grid

grouped Loop 4 are consecutive iterations of Loop 3
Determining the kernel

Loop 3 is splitted into **two phases**

- **first phase**
  - gathers the information needed by each iteration for its computing tasks on the GPU

- **second phase**
  - consists of a sequence of grid launches

Note that the number of grid launches
- is determined at runtime by the number of iterations of Loop 3
- is much lower than the number of iterations of Loop 3
- the number of executed CUDA threads is increased
Execution configuration

Great attention is required in the **choice of the size and number of blocks**

In the choice we considered:

- the available multiprocessors
- the dimension of the warp and the size of the grid
- the shared memory required by each block
- the occupancy (active threads/number of supported threads)
- the block sizes for obtaining the best performances (64–256)

We choose the **largest block size** which determines the **least number of void CUDA threads** among the possible values of 64, 128 and 192
Memory accesses

The design of memory transfers is a very important issue. We have:
- host memory
- many memory supports on device

Data to transfer:
- Bessel lookup table
- vertical and horizontal arrays of weights for linear interpolation among values in the Bessel lookup table
- support arrays
- *.xoo file
- output array
Memory accesses

- host-to-device transfers must be minimized
- *xoo* file and data related to Bessel functions must be transfer from host to device
- to reduce the impact of these transfers we **batch** several transfers in **one large transfer**
Memory accesses

- The *.xoo file is moved to global memory
- We use shared memory and cooperation among threads in the same block to obtain coalesced accesses
- The two arrays of weights for linear interpolation for Bessel functions (and other arrays) are moved in constant memory that is cached
- These (small) arrays are accessed in read-only mode at the same address by all threads
Memory accesses

- The output array is accessed in write-only mode and is stored in the global memory.
- The output array has a different size at each GPU execution.
- Each grid has a different number of threads, depending on Loops 3 grouped, and produces a different amount of output.
- To eliminate the overhead of multiple re-allocations we pre-allocate a large enough array.
Host side parallelization

**multicore processors**

- the several cores generate a set of flows of execution at CPU level
- sequential execution of CUDA grids
- grids related to **different observers** working with the **same frequency value**

**creation of several CUDA grids at the same time**

**competition for the GPU among the CPU threads (+ other drawbacks)**

merged

**single grid**

**Loop 1 is efficiently multicore parallelized**

This level of parallelization is implemented by **OpenMP**
Host side parallelization

- Observers are partitioned among the nodes of the **cluster** (multicore CPU + GPU)

- This level of parallelization is implemented by **MPI**
  - The inter-process communications are limited
  - The resulting overhead is not significant
  - The use of the cluster shows a **good scalability**

- A **cluster** corresponds to an architecture with **distributed memory** instead of shared memory

- Input scenarios for real cases with a high number of observers (order of $10^2$)
Experimental results

Our cluster consists of 4 nodes

Each node consists of

- 2 quad-core Intel Xeon E5520 CPUs operating at 2.27 GHz
- 2 Tesla T10 GPUs, each with
  - 240 cores
  - 4 GB memory
  - four T10 GPUs forms a **Tesla S1070 GPU Computing System**

- Each node runs Linux kernel 2.6.27
- Nodes are connected by a PCI Express-2 bus
Experimental results

- Data related to the earthquake of Colfiorito, Italy, 1997
- The experimental test considers
  + 256 observers
  + 287 different frequencies for each observer
  + a grid of 120000 points associated with the fault plane

We evaluate the performance of the two parallelized versions:
- TFaultMPI the version running on the multicore CPU cluster
- TFaultCUDA the version realized for the multicore CPU cluster with two Tesla associated to each node.

We measure the execution time required by the two versions by using 1 up to 4 nodes of the cluster
### Performance

<table>
<thead>
<tr>
<th></th>
<th>1 node</th>
<th>2 nodes</th>
<th>3 nodes</th>
<th>4 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TFaultMPI</strong></td>
<td>61126 s (16 h 59 m)</td>
<td>32084 s (8 h 55 m)</td>
<td>20728 s (5 h 45 m)</td>
<td>15532 s (4 h 19 m)</td>
</tr>
<tr>
<td><strong>TFaultCUDA</strong></td>
<td>6428 s (1 h 47 m)</td>
<td>3242 s (54 m)</td>
<td>2024 s (33.7 m)</td>
<td>1511 s (25.2 m)</td>
</tr>
<tr>
<td><strong>Speedup</strong></td>
<td>9.5x</td>
<td>9.9x</td>
<td>10.2x</td>
<td>10.3x</td>
</tr>
</tbody>
</table>

#### Graphs
- **Bar chart**: Comparison of time (s) for TFaultMPI and TFaultCUDA across different node counts.
- **Line chart**: Graph of speed-up with increasing node counts.
Experimental results

We perform a comparison between

- the original version of COMPSYN
- the two parallel versions, TFaultMPI and TFaultCUDA

Due to the long time required to perform the simulation over the whole set of 256 observers, we made the simulation for a smaller subset and extrapolated the execution time value.

The computed estimate of the speedup for 256 observers is:

- $15x$ for TFaultMPI
- $140x$ for TFaultCUDA,

by using a single node in both cases.
Conclusions and future work

This parallelization study has given promising results

We have two parallelized versions

- TFaultMPI, running on a multicore CPU cluster (host side parallelization)
- TFaultCUDA, parallelized both host side and device side

Advantages

- the use of COMPSYN is effective even in the most complex scenarios
- the parallel version is naturally scalable thanks to the balancing of the workload obtained by distributing the computations related to the different observers on the available computing nodes
Conclusions and future work

Meanwhile work

- We have improved the parallelization design by
  - implementing the register spilling
  - improving the memory coalesced accesses
  - using the page-locked memory and the texture memory
  obtaining a better performance

Future work

- We are planning to realize a browser based interface
- We are evaluating the parallelization of other modules of COMPSYN in a problem-oriented fashion