A Comparison of CUDA and OpenACC

Accelerating the Tsunami Simulation EasyWave

PASA@ARCS, Lübeck, February 26th 2014

Johannes Spazier[1], **Steffen Christgau**[1], Bettina Schnor[1], Martin Hammitzsch[2], Andrey Babeyko[2], Joachim Wächter[2]

[1] Institute for Computer Science, University of Potsdam

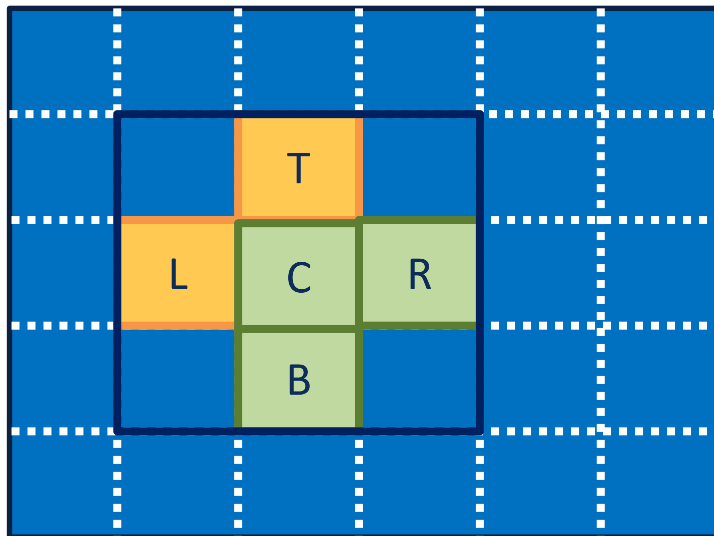
[2] GFZ German Research Center for Geosciences, Potsdam

Motivation

- TRIDEC project (critical decision support in evolving crisis)
- Tsunami (Early) Warning Center
  - collects sensor data (seismic, tide, GPS etc.)
  - **Tsunami analysis** (e.g. **simulation**)
  - emission of warnings to affected areas
- use DB-stored, pre-computed models and interpolate
- analysis is time critical, but compute intense
  - optimize computation

EasyWave

- TRIDEC's grid-based simulation component (C++)
- simplified computation (e.g. linear approximations)
- computation done in dynamic bounds/window
  - Phase 1: TLC stencil for wave heights
  - Phase 2: BRC stencil for flux update (momentum conservation)
  - Phase 3: extend window, if required

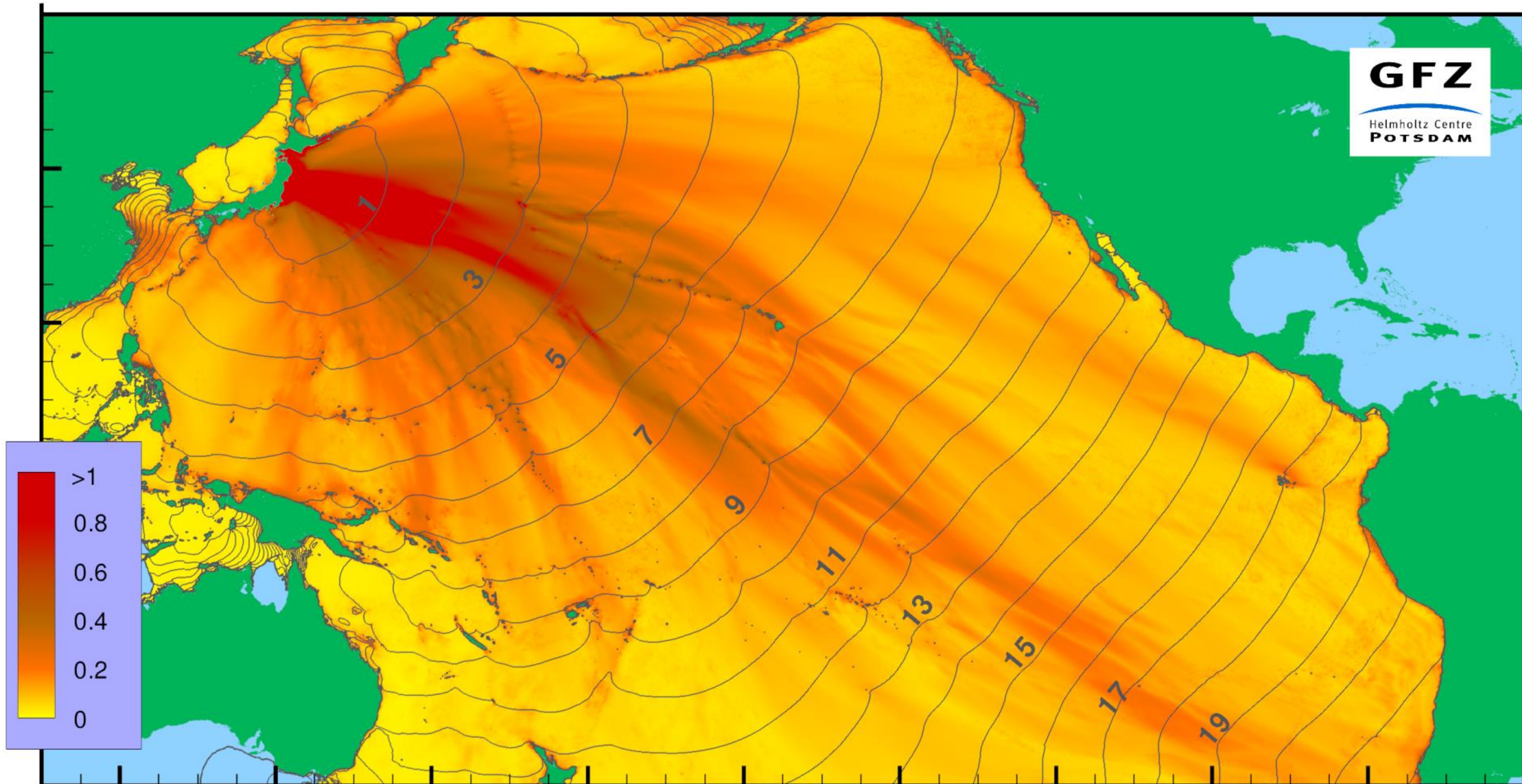


grid cell properties
- current wave height
- maximum wave height
- flux properties
- wave arrival time
- …

(single precision float arrays)

grid size: 2800 x 1800, 7200 time steps (typical) computed in approx. 5 min

EasyWave: Visualization

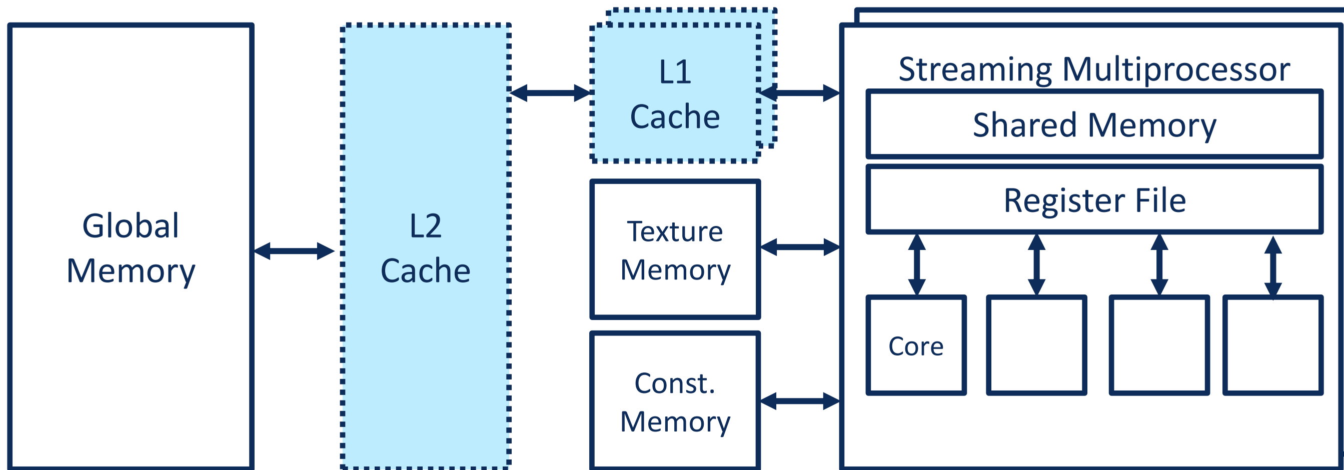


Parallelization

- massively parallel and time-critical problem
- use GPUs to calculate wave propagation
- program with different GPU APIs
  - CUDA C, low-level, manual parallelization
  - OpenACC, high level compiler-supported parallelization
  - OpenCL not considered
- compare APIs for real-world scientific application
  - Performance, i.e. required wall time for simulation
  - coding effort to achieve gained performance

Experimental Setup

- two generations of Nvidia cards in different host systems
  - C1060: 240 Cores, 4 GB RAM, **Tesla** Architecture
  - C2075: 448 Cores, 5.2 GB ECC RAM, **Fermi** Architecture



- Software
  - CUDA 5.5 Toolkit, GCC's g++
  - OpenACC: PGI Compiler

CUDA Parallelization

1. Straight-forward parallelization to SIMT
   - one thread per grid cell in phase 1 and 2
   - phases computed in different kernels (synchronization)
2. parallel window extension

program variant	time, C1060 (Tesla)	time, C2075 (Fermi)
CPU Version, sequential	348 s	305 s
SIMT port	162 s (-53 %)	28,4 s (-90%)
par. window ext.	142 s (-13 %)	15,3 s (-46 %)

numbers from typical data set, relative numbers valid for other tested scenarios

CUDA: memory alignment

- usual / well-known tuning method
- use `cudaMallocPitch/cudaMemcpy2d`
- maintain alignment in case of window extension
- negligible improvement (4% for Tesla)
  - additional computation due to window extension

CUDA: Call by Value

- arrays passed to kernels using pointer array

- serialized read access on `data` array (Tesla) + double dereferencing
- avoided by passing all arrays by value

program variant	time, C1060 (Tesla)	time, C2075 (Fermi)
CPU Version, sequential	348 s	305 s
SIMT port	162 s (-53 %)	28,4 s (-90%)
par. window ext.	142 s (-13 %)	15,3 s (-46 %)
call by value	**62 s (-59 %)**	**13,9 s (-6 %)**

CUDA: Shared Memory

- well known optimization technique
  - Shared Memory as SW managed cache
  - copy computed domain to shared memory
- performance reduction on Fermi card
  - additional overhead, cache present in hardware

program variant	time, C1060 (Tesla)	time, C2075 (Fermi)
CPU Version, sequential	348 s	305 s
SIMT port	162 s (-53 %)	28,4 s (-90 %)
par. window ext.	142 s (-13 %)	15,3 s (-46 %)
call by value	62 s (-59 %)	13,9 s (-6 %)
shared memory	**34 s (-45 %)**	**17,7 s (+27 %)**

CUDA: Summary

- good performance on Fermi by just porting to SIMT
- more tuning to HW required on Tesla
- „traditional“ optimization techniques show low performance gains (even loss) on Fermi
- large programming effort (ca. 50% more LoC)
- Kepler architecture not considered here

program variant	time, C1060 (Tesla)	time, C2075 (Fermi)
CPU Version, sequential	348 s	305 s
fastest CUDA	**34 s (10x faster)**	**13,9 s (22x faster)**

OpenACC

- OpenMP-like parallelization using compiler hints
  - pragmas for data movement and parallelization
  - small programming effort, easy integration
  - compiler generates code for accelerator HW
  - requires compiler support

```
#pragma acc data copyin(height[:w*h])
for (it = 0; it < nsteps; it++) {
  #pragma acc loop
  for (y = …)
    for (x = …) { /* compute */ }
}
```

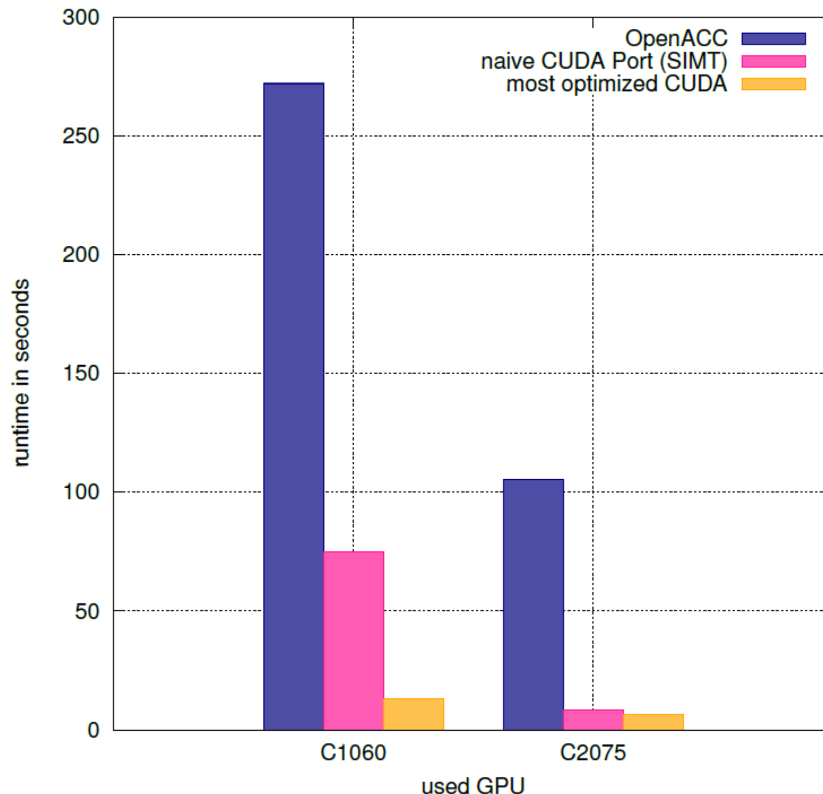
OpenACC: Parallelization

- straight forward code additions (+21 of 462 LoC)
- disappointing results
  - slow compared to untuned and most tuned CUDA
  - tested with PGI Compiler 13.6

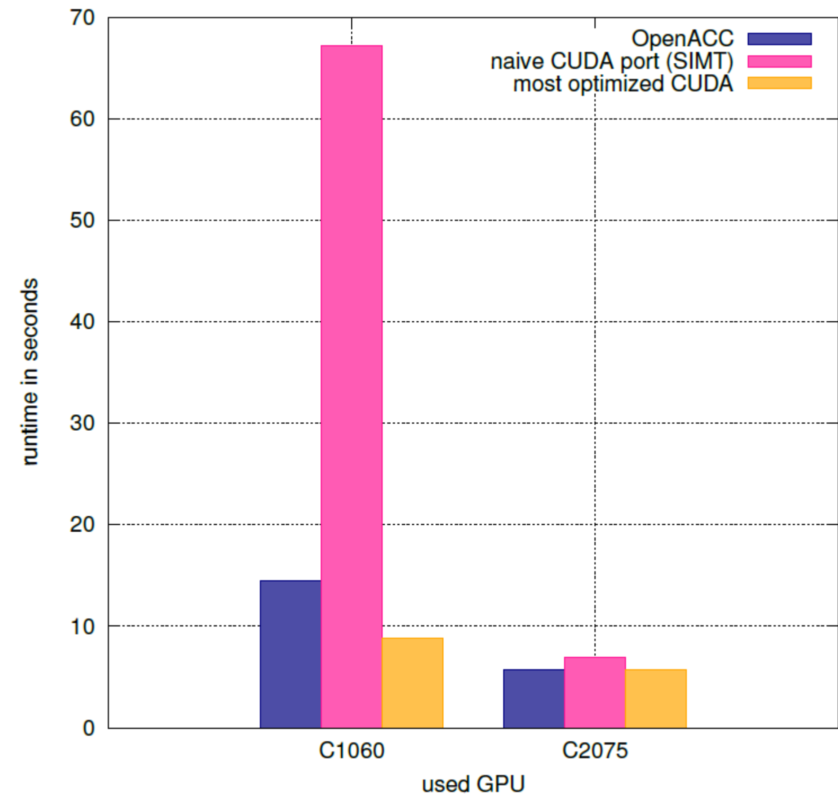
program variant	time, C1060 (Tesla)	time, C2075 (Fermi)
CPU Version, sequential	348 s	305 s
fastest CUDA	34 s (10x faster)	13,9 s (22x faster)
OpenACC	**302 s (8.8x slower)**	**130 s (9.4x slower)**

OpenACC: performance analysis

Phase 1: TLC stencil



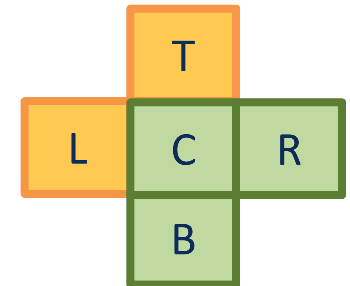
Phase 2: BRC stencil



- approx. equal performance of Phase 1 and 2 in direct CUDA/SIMT port
- OpenACC: performance loss for Phase 1, good performance for Phase 2

OpenACC: Compiler Issues

- very similar source code for phases
  - similar CPU and generated GPU code
  - phases differ mainly in stencil (memory bound)
- mapping of threads influenced by stencil
  - removing T from Phase 1 changes dimension of block grid
- bad choice for block and grid size by compiler
  - only 60% device occupancy (CUDA achieves 87%)



Workarounds

- specify mapping manually using `vector` (partially)
  - violates hardware abstraction
  - reduces gap between CUDA and OpenACC
  - still 3x slower on Fermi (compared to 10x slower)
- compiler update does not resolve issues
  - vendor contacted
  - same numbers for PGI 13.9 and 14.1 (recent: 14.2)

Summary

- CUDA
  - high performance, but high programming effort
  - detailed hardware knowledge required
- OpenACC
  - promising, easy API (see OpenMP)
  - performance (can be) comparable to tuned CUDA
  - compiler support is crucial