Advanced Optimization Techniques for Sparse Grids on Modern Heterogeneous Systems

Alin Murarașu

Computer Architecture Group
Technische Universität München
murarasu@in.tum.de
Outline

- Computational Steering (Scope and Motivation)
- Heterogeneous Computing (Introduction)
- The Sparse Grid Method (Problem Statement)
- Part 1: Porting to GPUs
- Part 2: Auto-tuning
- Part 3: Load Balancing
- Performance Evaluation
- Conclusion
Computational Steering

Simulation Cluster → slow → Data Repository → fast → Visualization Cluster

- **Scope:**
  - Computational steering using sparse grid method (*Virtual Arabia project*)

- **Motivation:**
  - No implementation of sparse grids for heterogeneous systems
  - Heterogeneous systems (~1 Tflops) could bring a substantial speedup

- **Goal:**
  - Develop a methodology for porting sparse grids to heterogeneous systems based on CPUs and GPUs

Regular sparse grid: 129 points
Dimensionally truncated sparse grid: 89 points

[benzina, buse, butnaru, murarasu, treib, varduhn, mac2013]
Heterogeneous Computing

- Heterogeneous system = CPU + GPU (GPU accelerated system)
  - Why? High Flops, high Flops / Watt, high Flops / €, high Flops / m^2

- CPUs = latency oriented processors
  - Complex logic for out-of-order execution and speculation
  - Quad-core peak performance: ~100 Gflops

- GPUs = throughput oriented processors
  - Large # of simple cores (16) with wide SIMD units (32)
  - Peak performance: ~1 TFlops

The path to high performance has many obstacles:
- Less memory on GPU
- Costly data transfers over PCIe
- GPU less suited for: control bound & recursion, irregular mem. accesses & parallelism
- GPU tradeoffs, e.g. concurrency vs. locality
- Computation offload programming model
The Sparse Grid Method

- **Sparse grid** = hierarchical & compressed representation of a d-dimensional function

- **Algorithms:**
  - **Hierarchize** = computes sparse grid values
  - **Interpolate** = evaluates sparse grid at some point(s)

- **Scenario:** sparse grid contains long vectors ⇒ interpolate → **SpVM**

**Problem statement:**
- Sparse grid stored as *(key, val)* pairs ⇒ keys consume more memory than values
- Algorithms are recursive & have complex control flow
- Parallelism & data access are irregular
- **Sparse grids are incompatible with GPUs(?)**
Solution and Scientific Contributions

The proposed solution has 3 parts:

1) Porting sparse grids to GPUs:
   - \( gp2idx = \text{a bijective mapping} \) (grid point \( \rightarrow \) index) that minimizes a sparse grid's memory footprint
   - Non-recursive & data parallel algorithms

2) An auto-tuning strategy for sparse grids on GPUs:
   - A pattern-based search method that explores a 3d tuning space
   - An input reduction technique that speeds-up auto-tuning

3) A load balancing strategy for sparse grids on GPUs:
   - A dynamic scheme: eager + prefetching + multiple grain sizes
   - A static scheme including fast calibration for new input parameters
Outline

- Computational Steering (Scope and Motivation)
- Heterogeneous Computing (Introduction)
- The Sparse Grid Method (Problem Statement)
- **Part 1: Porting to GPUs**
- Part 2: Auto-tuning
- Part 3: Load Balancing
- Performance Evaluation
- Conclusion
A Bijection Based Data Structure

- **Main idea:**
  - Store only values in a special order
    ⇒ 1d dense array, sg1d
  - **Sparse grid = sequence of dense**
    **d-dimensional blocks**

- **gp2idx(gridPoint) = index**

- **Example:**
  - `gp2idx((0.875, 0.125))` returns 76
  - `sg1d[76]` is the value for (0.875, 0.125)

[murarasu, buse, pflüger, weidendorfer, bode, iccs2012]
Non-recursive Data Parallel Algorithms

- **Hierarchization**: computing a value requires $2^d$ dependencies
  - **Recursive hierarchization + tree**:
    
    ```c
    function hierarchize1dr(gp, leftParVal, rightParVal, l, n)
    
    if l < n then
    
    hierarchize1dr(gp.leftChild, leftParVal, gp.val, l+1)
    
    hierarchize1dr(gp.rightChild, gp.val, rightParVal, l+1, n)
    
    gp.val = gp.val - (leftParVal + rightParVal) / 2
    ```

- **Non-recursive hierarchization + gp2idx**:
  
  ```c
  function hierarchize1di(numGridPoints, sg1d)
  
  for i = numGridPoints - 1 downto 0
  
  leftParVal = sg1d[gp2idx(leftParent(i))]
  
  rightParVal = sg1d[gp2idx(rightParent(i))]
  
  sg1d[i] = sg1d[i] - (leftParVal + rightParVal) / 2
  ```

- **Interpolation**:
  - Interpolate at $m$ points, $m \geq 10,000$
    ⇒ **Embarrassingly parallel**
  
  - Interpolation sums up contributions of all blocks
    ⇒ **Reduction (+)**

---

[murarasu, weidendorfer, buse, butnaru, pflüger, ppop2011]
Outline

- Computational Steering (Scope and Motivation)
- Heterogeneous Computing (Introduction)
- The Sparse Grid Method (Problem Statement)
- Part 1: Porting to GPUs
- **Part 2: Auto-tuning**
- Part 3: Load Balancing
- Performance Evaluation
- Conclusion
Tunable Parameters

- 3d search space:
  - Thread block size, $bs$: $bs$ affects TLP & ILP
  - Control over thread coarsening, $pt$:
    $pt \uparrow \Rightarrow$ TLP $\downarrow$, ILP $\uparrow$
  - Control over thread refining, $tr$:
    $tr \uparrow \Rightarrow$ TLP $\uparrow$, Sync $\uparrow$
- Use $tr$ when not enough coarse-grained TLP

- There is a pattern for $pt$ and $tr$ $\Rightarrow$ Pattern based search algorithm

Interpolation example:
- $pt = 3$: 3 interpolation points per thread
- $tr = 4$: 4 threads per interpolation point

Register reuse $\uparrow$ & ILP $\uparrow$

Register spilling $\uparrow$ & TLP $\downarrow$

[murarasu, weidendorfer, submitted]
Input reduction technique for GPU:
- Goal: reduce size of input data without altering auto-tuning results

Assumption: homogeneous threads

Wave = threads that start & finish synchronously

Main idea: reduce input so that only 1 wave is launched

Time \approx \# \text{ of waves} \times \text{time per wave}
Outline

- Computational Steering (Scope and Motivation)
- Heterogeneous Computing (Introduction)
- The Sparse Grid Method (Problem Statement)
- Part 1: Porting to GPUs
- Part 2: Auto-tuning
- **Part 3: Load Balancing**
- Performance Evaluation
- Conclusion
Load Balancing

- **Dynamic task based scheme:**
  - **Our approach: derived from eager scheduling**
    - Overlapping of PCIe communication & computation on GPU (\( \text{max. } 3x \))
    - Different grains sizes for CPU & GPU worker

- **Static scheme:**
  - Work divided in 2 chunks ~ Flops: for CPU & GPU
  - But Flops depends on input parameters, e.g. \( d \)
    \( \Rightarrow \) recalibrate chunks
  - Fast calibration is possible using input reduction

[ murarasu, weidendorfer, bode, uchpc2012 ]
Hierarchization's Performance

- Hardware configuration:
  - CPU: 2 x Intel Nehalem Quad-core
  - GPU: 1 x Nvidia Quadro 6000
- Reference: recursive
- \(d = 10, m = 10^6\) (interpolation points)

**Behavior:**
- Integer bound
- Memory bound
Interpolation's Performance

Auto-tuning results:
- \((bs, pt, tr) = (128, 4, 1)\)
- Input reduction: \(~10x\) faster auto-tuning / \(~5\%\) Gflops loss

Behavior:
- Compute bound
SpVM's Performance

### Auto-tuning results:
- \((bs, pt, tr) = (128, 10, 1)\)
- Input reduction: ~2x faster auto-tuning / ~5% Gflops loss

### Behavior:
- Memory bound
Conclusion

- The sparse grid method was successfully ported to Nvidia GPUs although it seemed incompatible at the beginning
  - The proposed optimizations are applicable to AMD GPUs and Intel Xeon Phi
- General message: mapping irregular parallelism to GPUs is difficult, e.g. redesigned data structures and algorithms
- Summary of achievements:
  - Porting to GPUs: minimal memory footprint,
    - GPU 2x – 8x faster than 8-core CPU
  - Auto-tuning: 1.3x faster GPU,
    - 2x - 10x faster auto-tuning using input reduction
  - Load balancing: CPU+GPU 1.1x – 1.4x faster than GPU-only

- Software:
  - fastsg: C++ sparse grid library (OpenMP and CUDA)
  - t-helper (ongoing): #pragma based auto-tuning framework
Thank you!

- **Lehrstuhl X**: Prof. Bode, Josef, Hans, Carsten, Thomas, Minh, David, Til, Georg, Prof. Gerndt, Andreas, Marcel, Abu Zein, Yury, Ventsislav, Fisnik, Robert, Isaias

- **Lehrstuhl V**: Prof. Bungartz, Gerrit, Dirk, Daniel, Michael, Tobias

- **Virtual Arabia**: Amal, Marc, Vasco

- **Family and friends!**

Questions?
4-core Intel CPU:
- Low latency oriented
- Short vector units (128-bit / 256-bit)
- Out-of-order execution
- Speculative techniques
- Large caches

16-core Nvidia Fermi GPU:
- High throughput oriented
- Wider vector units (1024-bit)
- In-order execution
- Extreme multi-threading
- Higher peak Flops: ~10x
- Higher memory bandwidth: ~10x