GPU Acceleration of CFD Codes and Optimizing for GPU Memory Hierarchies

Dr. Frank Mueller
Nishanth Balasubramanian

North Carolina State University
Infrastructure: ARC cluster at NCSU

- **Hardware**
  - 2x AMD Opteron 6128 (8 cores each), 120 nodes = ~2000 CPU cores
  - NVIDIA GTX480, GTX680, C2050, K20c, K40c
  - Mellanox

- **Software**
  - CUDA 5.5
  - PGI Compilers V13.9 w/ CUDA Fortran & OpenACC support
  - OpenMPI & MVAPICH2-1.9 w/ GPUDirect V2 capability
  - Torque/Maui job management system
Topic 1: GPU Acceleration for CFD

- CFD targeted: RDGFLO
- an MPI-based parallel discontinuous Galerkin finite element solver
- GPU technology: OpenACC
- Identify compute intensive regions in code
  - Run them on GPU
- Aided in porting code to
  - At first: single GPU, no MPI
Initial attempt

- Compute intensive regions considered of mainly 2 loops
  - where > 50% of run time was spent
- First approach: Naïve parallelization
  - add OpenACC directives around loops
  - with data copy in/out statements
- opened up set of new problems...
Challenges / Solutions

- **Compiler Auto Optimizations:**
  - Compiler matched variables inside/outside kernel
    - automatically upgraded them to live out variables
  - made code run serially
    - since last value of variables needs to be computed
  - **Solution:** variable renaming inside kernel
    - ensures that variables not matched by compiler

- **Subroutines:**
  - Subroutine calls not supported in OpenACC
  - **Manual inlining** of essential parts per subroutine
Race condition

- Data dependence check disabled
  - compiler too conservative to get good speedups
- Naïve parallelization produced incorrect output data
- Race condition:
  - Single array location w/ >1 writes from different threads
  - Solution: Update data in batches
    - indices to be updated were reordered to create batches
    - Each batch modifies only unique elements in its batch
- Overheads: extra data structures
  - to reorder and maintain batch information
Other Subroutines + Results

- Problem: data copy
  - Frequent copies CPU ↔ GPU: lots of time spent here
- Objective: ensure only 1 copy in + 1 copy out of GPU
- Effect: had to move essential parts of other subroutines → GPU
  - Majority: computation (w/o memory contention)
  - Minority: memory contention → batching strategy used again

![Execution Time](chart)
Future Work

- **Status Quo:**
  - multiple kernels called
    - each subroutine parallelized independently

- **Future:** Once subroutines supported on GPUs in PGI compiler
  - single kernel call w/ subroutine calls
    - eliminates unnecessary overhead

- Run solver for bigger grid sizes

- Enable MPI
  - run on >1 CPU each w/ a GPU + full optimizations
    - need ghost cells /halo region
      - xfer ghost/helo GPU₁ -- CPU₁ ← MPI → GPU₂ -- CPU₂
Architectures feature reconfigurable memory hierarchies
- Support hybrid scratch pad + cache

NVIDIA: scratch pad “shared” (shmem)/L1 cache per SM

Intel MIC (KNL): scratch pad “near memory”/L1 cache

Which one should be use?
- Cache: → always best?
  - Transparent, no pgm change
- Scratch pad:
  - explicit addressing
  - more control
Matrix Multiply (MM)+FFT

- Tiled 16x16, total 256x256, TB (16, 16)
  - Shmem: MM 0.16 ms → wins!
  - FFT 0.69 ms → wins!

- Why?

---

**Software-managed cache code**

```c
#define : tx,ty : threadIdx.x , threadIdx.y ; bx, by : blockIdx.x, blockIdx.y

for (each thread block)
{
    _shared__ float
    As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float
    Bs[BLOCK_SIZE][BLOCK_SIZE];
    AS(ty, tx) = A[a + WA * ty + tx];
    BS(ty, tx) = B[b + WB * ty + tx];
    __syncthreads();
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += AS(ty, k) * BS(k, tx);
        __syncthreads();
    }
}
```

**Hardware-managed cache code**

```c
for (each thread block)
{
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += A[a + WA * ty + k] * B[b + k * WB + tx];
    }
}
```

- L1 cache:
  - MM L1 cache: 0.23 ms
  - FFT L1 cache: 2.36 ms

- **GTX 480**
Matrix Multiply+FFT: GPGPUSim Analysis

- Shmem: 0.16 ms \(\rightarrow\) wins - why?
  - 5 thread blocks (TBs) \(\rightarrow\) regs limit
  - Latency: 44us, +12% instr. (copies)
  - 1 mem block access / warp
    - Same bank+row (32 banks)
    - Due to user-based mapping

- Matmult: shmem good for
  - High memory-level parallelism:
    Accesses from threads overlapped
  - Classical cache misses
    \(\rightarrow\) do not matter much
    - Confirmed w/ fully assoc L1
  - Memory coalescing almost for free!

- L1 cache version
  - 5 TBs, no conflict misses (sim.)
  - Latency 80us (TLP hides this)
  - 2 cache block accesses / warp
    - Block size 128B
    - Due to phys. addr. map \(\rightarrow\) L1

- FFT: L1 write misses high
  - Allocate-on write L1 policy bad!
    - Verified in simulation
  - Associativity/capacity don’t matter here
# Marching Codes (MC) + Path Finder (PF)

- **Shmem:**
  - MC 0.139 ms
  - PF 0.108 ms
- **MC:** Max. 5 active thread blocks / SM $\rightarrow$ only 5 warps active
- **PF:** need syncthread() for shmem

<table>
<thead>
<tr>
<th>Software-managed cache</th>
<th>Hardware-managed cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC extracts a geometric isosurface from a volume dataset.</td>
<td></td>
</tr>
<tr>
<td>$&lt;$Generate triangles$&gt;$ kernel is a key step; it looks up the fields values and generates the triangle voxel data. Each TB 32 threads: NTHREADS, each grid 1024 TB: NBLOCKS, generate 32768 voxels</td>
<td></td>
</tr>
</tbody>
</table>

```c
Cacl_vertex_pos();
Lookup_field();
_shared float vertlist[12*NTHREADS];
_shared float normlist[12*NTHREADS];
//each tb in shared memory
//i: 0~11
Compute_vertlist( [tidx+i*NTHREADS] );
Compute_normlist( [tidx+i*NTHREADS] );
//each tb
Write_global();
```

```c
Cacl_vertex_pos();
Lookup_field();
float vertlist[12];
float normlist[12];
//each thread in local memory
//i: 0~11
Compute_vertlist( [i] );
Compute_normlist( [i] );
//each thread
Write_global();
```

- **L1 cache:**
  - MC 0.115 ms $\rightarrow$ wins! Why?
  - PF 0.096 ms $\rightarrow$ wins! Why?
- **Max. 8 active thread blocks / SM $\rightarrow$ no artificial shmem limit
- **PF:** fewer address calculations

---

**MC: Max. 5 active thread blocks / SM $\rightarrow$ only 5 warps active**
- MC 0.139 ms
- PF 0.108 ms

**PF:** need syncthread() for shmem
- L1 cache:
  - MC 0.115 ms $\rightarrow$ wins! Why?
  - PF 0.096 ms $\rightarrow$ wins! Why?
Marching Codes + PathFinder

- Thread-level parallelism (TLP) study

- Control # thread blocks (TBs)
  - Via fake shmem array (limits TBs)

- 4 benchmarks: best performance NOT at max. # TBs!
  - Problem: L1 capacity vs. TB pressure for L1 space
GTX 480

- `shmemp wins, except for PF, MC`
- `On average (geom. Mean): 55.7% performance win`
GTX 680

- shmem wins, except for PF, MC
- more ALUs, less latency compared to GTX480 → higher wins!
Scratch Pad vs. Cache Conclusion

- In-depth study
  - reveals interesting, unexpected tradeoffs
- TLP can significantly hide performance impact of L1 cache misses
- more subtle factors for performance & energy:
  - Key reasons for differences:
- shmem:  +MLP and coalescing
- D-cache:  +Improved TLP and store data into registers

- Most benchmarks favor shmern
  \(\rightarrow\) Justifies software complexity to manage them
Topic 3: Memory Tracing, Cache Analysis

- **ScalaMemTrace:**
  - Tool built at NCSU
  - uses PIN to instrument loads/stores of a program
  - creates compressed memory traces as RSDs/PRSDs

- **Reuse Distance:** \( \# \) of distinct accesses b/w 2 memory accesses

- **Example:**
  - time: 1 2 3 4 5 6 7 8 9 10 11 12
  - access: d a c b c c g e f a f a f b
  - distance: \( \leftarrow 5 \) distinct accesses \( \rightarrow \)

- used to predict hit/misses in a cache given its configuration
Application of Memory Tracing

- to predict cache performance
  - Assumes regular array accesses → GPU kernels
- # hits/misses calculated @ every loop level
  → provide better understanding of cache/memory performance
  - approximate: fast prototyping (not exact)
- Target CFD codes
  - contain continuous loops w/ regular stride memory accesses
- Example from CFD:
  ```
do ifa = njfac+1 ....
  ...loop over internal faces...
do ig = 1, ngaus
  ...update flux into face array (rhsfa)...
  enddo
endo
do
endo```
Overview of Memory Tracer

Application

Memory Instrumentation tool (PIN)

Trace Compressor to generate RSDs

Predict cache hits/misses based on cache config
Memory Trace Represented as a Tree

Indicates loop

Nested loop
Context-based Reuse Distance Calc.

- Node in tree: loop head / strided data access
- Create “left+right context” per loop head
- Left context (LC): contains first set of accesses in the loop
  - Up to cache size (capacity limit)
- Right context (RC): contains last set of accesses in loop
  - in order of LRU to MRU (again, cache size capacity limit)
- Algorithm:
  - for each loop level: LC/RC in tree + memory access
    - predict hits/misses locally
  - @ next upper loop level: compose LC(child)+RC(parent)
    - adjust hits/misses of child due to earlier accesses
- Context size bounded: # arrays fitting in cache
Assumptions

- For following example:
  - Fixed context size
  - All arrays of size N
  - Size per element is fixed to 4 bytes (sizeof(int)).

- In general:
  - All cold misses counted as capacity misses
  - No partial replacement of arrays in cache
Example

- **Left context (LC):** contains first set of accesses in the loop
  - Up to cache size (capacity limit)
- **Right context (RC):** contains last set of accesses in loop
  - in order of LRU to MRU (again, cache size capacity limit)
Example

- Algorithm:
  - for each loop level: LC/RC in tree + memory access → predict hits/misses locally
  - @ next upper loop level: compose LC(child)+RC(parent) → adjust hits/misses of child due to earlier accesses
Example

- Algorithm:
  - for each loop level: LC/RC in tree + memory access
    → predict hits/misses locally
  - @ next upper loop level: compose LC(child)+RC(parent)
    → adjust hits/misses of child due to earlier accesses
Example

- Algorithm:
  - for each loop level: LC/RC in tree + memory access
    → predict hits/misses locally
  - @ next upper loop level: compose LC(child)+RC(parent)
    → adjust hits/misses of child due to earlier accesses
Partial Array Replacement

- Approach: find overlapping region first
- Hits+misses assigned to conflicting arrays
  - Depends on overlap region
- Part of an array may be left in cache
  - Keeping partial info → increases algorithm complexity
- Instead: if only part of array left in cache
  - Consider it not present in cache → removed from context
- Option (later): use % overlap to remove array from context based
Testing

- DineroIV: trace-driven simulator from University of Wisconsin
  - as reference: provides hits/misses for uncompressed trace
- compare the total misses (Dinero vs. our ScalaMemTrace)
- Results from compressed traces match DineroIV for
  - different cache sizes
  - associativity levels
    - under given assumptions
    - for a cache configuration
Current Progress

- **Initial implementation**
  - Single loops: works, validated
  - partial array replacement case
  - run test cases to check difference Dinero/ScalaMemTrace
    - for different cache configs

- **Ongoing**
  - test nested loops

**Overall Objective:** Provide a quick answer to:
- Which loops should become GPU kernels?
Future Work

- Identify where most misses occur
  - Based on cache performance data
  - Provide suggestions to increase cache hits
  - Extrapolate to GPU memory usage
- Build multi-processor cache/memory model
  - Runs multiple instances of uni-processor cache simulator
- Cache simulator’s output → DRAM latency simulator
  - Predict time taken for memory accesses

Extrapolate GPU behavior (1000s threads)
from CPU behavior (memory trace of 10s of threads)
Future Work

Multicore system

HyperThreaded system

GPU