Performance Insights on Executing Non-Graphics Applications on CUDA on the NVIDIA GeForce 8800 GTX

Hot Chips 19

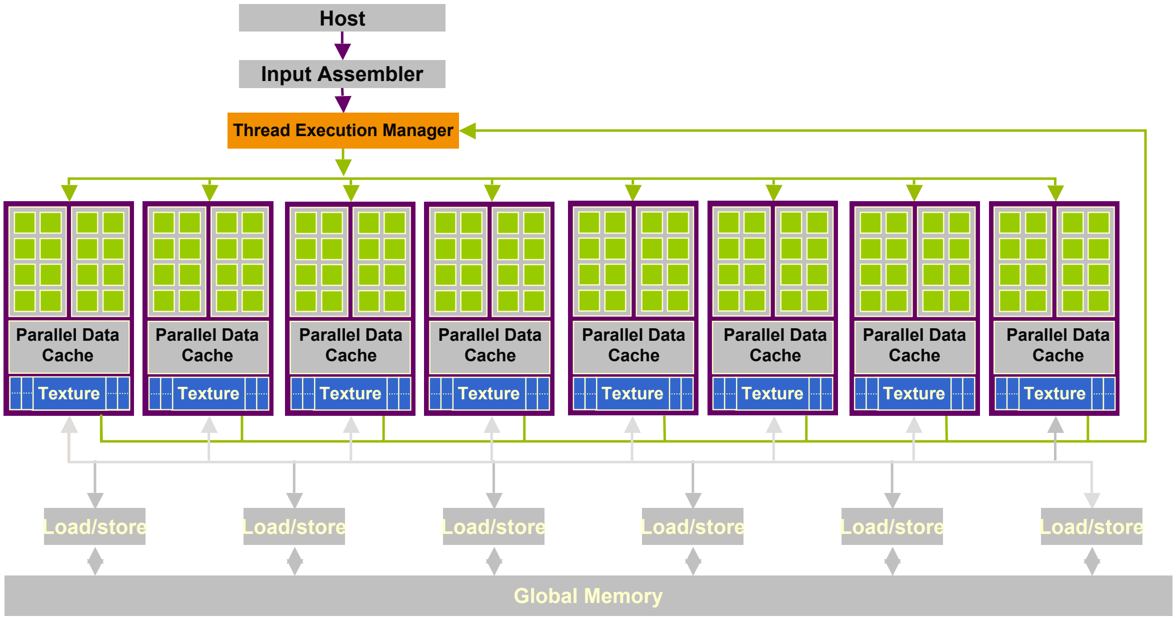
Wen-mei Hwu
with
David Kirk, Shane Ryoo, Christopher Rodrigues,
John Stratton, Kuangwei Huang

Overview

- Brief rundown of GeForce 8800 architecture
- Considerations in GPU performance optimization
- Benchmark performance
- Three case studies
  - MRI image reconstruction
  - LBM fluid dynamics simulation
  - H.264 image comparison
- Common performance limitations
- Concluding remarks

GeForce 8800 GPU Computing

Up to 65,535^2 thread blocks with up to 512 threads each
128 cores, 367 GFLOPS, 768 MB DRAM, 8GB/s total BW
Resources allocated at per-block granularity



Computation Strategy

- We make use of compute resource and hide global memory latency via:
  - Many independent threads
  - Independent instructions within a thread
  - Use of several local memories per Streaming Multiprocessor to reduce latency, avoid redundant global memory accesses and thus bandwidth saturation
- Memory latencies must be overlapped with useful work to achieve good overall performance
  - Global memory latency is at least 200 cycles (estimated)
  - Texture memory accesses and some floating-point operations also have long latencies

Additional Performance Considerations

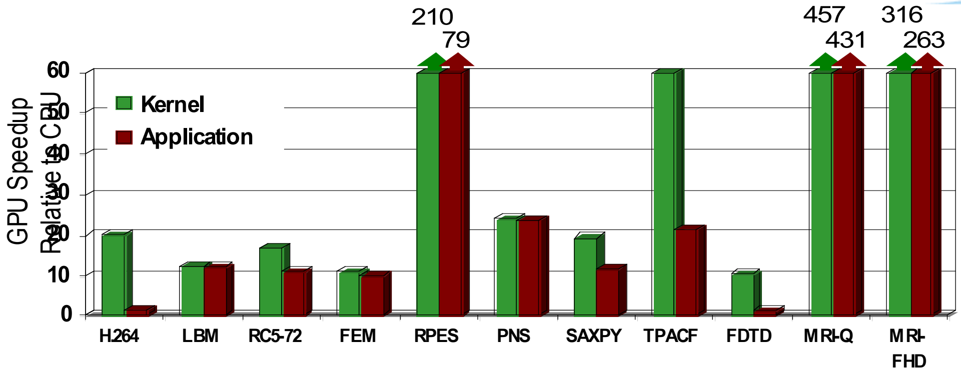
Developers need to keep additional potential limiters in mind:

- Stalls and bubbles in the pipeline
  - Port conflicts to shared/constant memory
  - Branch divergence
- Shared resource saturation
  - Global memory bandwidth can be saturated
    - Especially if hardware cannot coalesce multiple loads/stores into fewer memory accesses
  - Local memories and registers can also be filled, limiting the number of simultaneously-executing threads

Parallel Programming Experience

Application	Description	Source	Kernel	% time
H.264	SPEC '06 version, change in guess vector	34,811	194	35%
LBM	SPEC '06 version, change to single precision and print fewer reports	1,481	285	>99%
RC5-72	Distributed.net RC5-72 challenge client code	1,979	218	>99%
FEM	Finite element modeling, simulation of 3D graded materials	1,874	146	99%
RPES	Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion	1,104	281	99%
PNS	Petri Net simulation of a distributed system	322	160	>99%
SAXPY	Single-precision implementation of saxpy, used in Linpack's Gaussian elim. routine	952	31	>99%
TRACF	Two Point Angular Correlation Function	536	98	96%
FDTD	Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation	1365	93	16%
MRI-Q	Computing a matrix Q, a scanner's configuration in MRI reconstruction	490	33	>99%

Speedup of GPU-Accelerated Functions

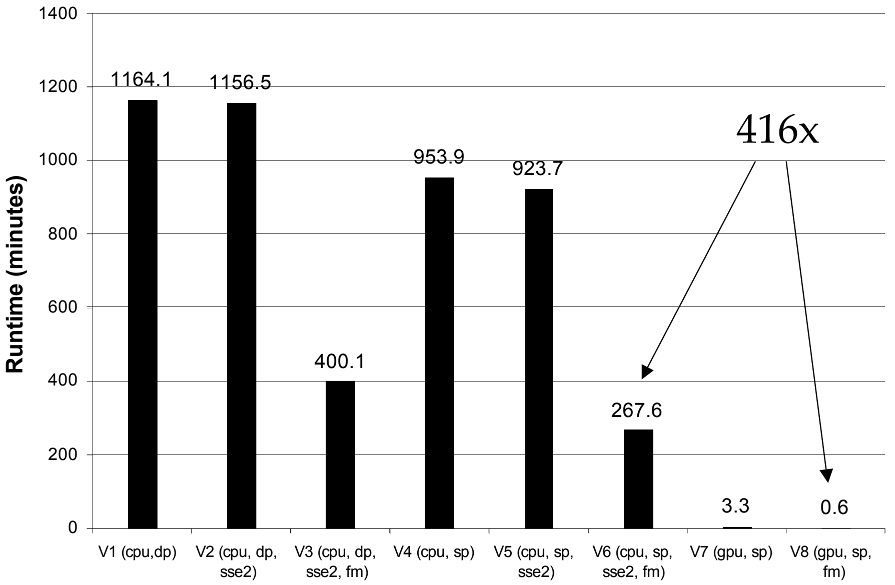


- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25× to 400× speedup if the function’s data requirements and control flow suit the GPU and the application is optimized
- Keep in mind that the speedup also reflects how suitable the CPU is for executing the kernel

Magnetic Resonance Imaging

- 3D MRI image reconstruction from non-Cartesian scan data is very accurate, but compute-intensive
- 416× speedup in MRI-Q (267.6 minutes on the CPU, 36 seconds on the GPU)
  - CPU – Athlon 64 2800+ with fast math library
- MRI code runs efficiently on the GeForce 8800
  - High-floating point operation throughput, including trigonometric functions
  - Fast memory subsystems
    - Larger register file
    - Threads simultaneously load same value from constant memory
    - Access coalescing to produce < 1 memory access per thread, per loop iteration

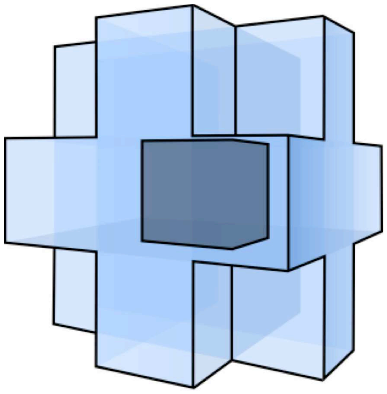
Computing Q: Performance



CPU (V6): 230 MFLOPS GPU (V8): 96 GFLOPS

LBM Fluid Simulation (from SPEC)

- Simulation of fluid flow in a volume divided into a grid
  - It's a stencil computation: A cell's state at time *t+1* is computed from the cell and its neighbors at time *t*
- Synchronization is required after each timestep – achieved by running the kernel once per timestep
- Local memories on SMs are emptied after each kernel invocation
  - Entire data set moves in and out of SMs for every time step
  - High demand on bandwidth
- Reduce bandwidth usage with software-managed caching
  - Memory limits 200 grid cells/threads per SM
  - Not enough threads to completely cover global memory latency



Flow through a cell (dark blue) is updated based on its flow and the flow in 18 neighboring cells (light blue).

H.264 Video Encoding (from SPEC)

- GPU kernel implements sum-of-absolute difference computation
  - Compute-intensive part of motion estimation
  - Compares many pairs of small images to estimate how closely they match
  - An optimized CPU version is 35% of execution time
  - GPU version limited by data movement to/from GPU, not compute
- Loop optimizations remove instruction overhead and redundant loads
- …and increase register pressure, reducing the number of threads that can run concurrently, exposing texture cache latency

Prevalent Performance Limits

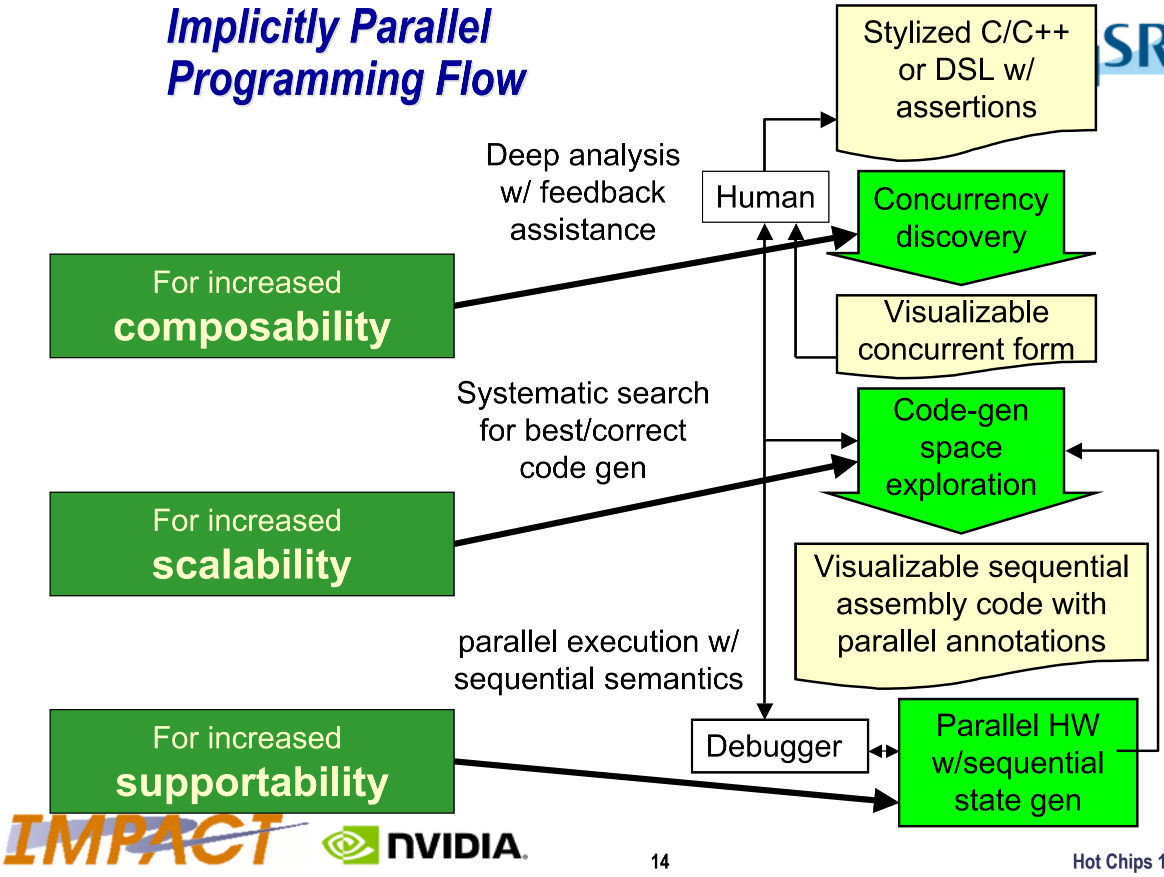
Some microarchitectural limits appear repeatedly across the benchmark suite:

- Global memory bandwidth saturation
  - Tasks with intrinsically low data reuse, e.g. vector-scalar addition or vector dot product
  - Computation with frequent global synchronization
    - Converted to short-lived kernels with low data reuse
    - Common in simulation programs
- Thread-level optimization vs. latency tolerance
  - Since hardware resources are divided among threads, low per-thread resource use is necessary to furnish enough simultaneously-active threads to tolerate long-latency operations
  - Making individual threads faster generally increases register and/or shared memory requirements
  - Optimizations trade off single-thread speed for exposed latency

Lessons Learned

- Parallelism extraction requires global understanding
  - Most programmers only understand parts of an application
- Algorithms need to be re-designed
  - Programmers benefit from clear view of the algorithmic effect on parallelism
- Real but rare dependencies often need to be ignored
  - Error checking code, etc., parallel code is often not equivalent to sequential code
- Getting more than a small speedup over sequential code is very tricky
  - ~20 versions typically experimented for each application to move away from architecture bottlenecks

Implicitly Parallel Programming Flow



Stylized C/C++ or DSL w/ assertions

Human

Deep analysis w/ feedback assistance

Concurrency discovery

For increased **composability**

Visualizable concurrent form

Systematic search for best/correct code gen

Code-gen space exploration

For increased **scalability**

Visualizable sequential assembly code with parallel annotations

parallel execution w/ sequential semantics

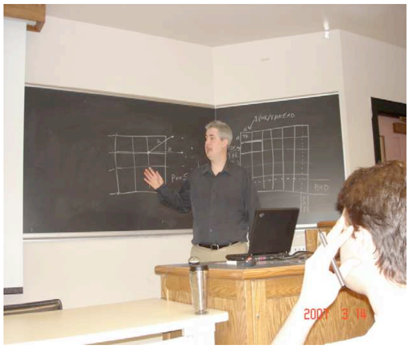
Debugger

Parallel HW w/sequential state gen

For increased **supportability**

To Learn More

- UIUC ECE498AL – Programming Massively Parallel Processors (http://courses.ece.uiuc.edu/ece498/al/)
  - David Kirk (NVIDIA) and Wen-mei Hwu (UIUC) co-instructors
  - CUDA programming, GPU computing, lab exercises, and projects
  - Lecture slides and voice recordings



Thank you! Any Questions?

Some Hand-coded Results

App.	Archit. Bottleneck	Simult. T	Kernel X	App X
H.264	Registers, global memory latency	3,936	20.2	1.5
LBM	Shared memory capacity	3,200	12.5	12.3
RC5-72	Registers	3,072	17.1	11.0
FEM	Global memory bandwidth	4,096	11.0	10.1
RPES	Instruction issue rate	4,096	210.0	79.4
PNS	Global memory capacity	2,048	24.0	23.7
LINPACK	Global memory bandwidth, CPU-GPU data transfer	12,288	19.4	11.8
TRACF	Shared memory capacity	4,096	60.2	21.6
FDTD	Global memory bandwidth	1,365	10.5	1.2
MRI-Q	Instruction issue rate	8,192	457.0	431.0

[HKR HotChips-2007]

Magnetic Resonance Imaging

- MRI code makes effective use of fast memory subsystems
  - Larger register file allows voxel data to be stored in registers
  - Threads load the same values from constant memory in the same cycle
  - 5 load instructions per iteration, but with access coalescing, this produces < 1 memory access per thread, per loop iteration

CPU Code

```
for(i=0 to max_K) {
  for (j = 0 to max_X) {
    w = 2PI * dot(k[i], x[j]);
    cw = cos(w); sw = sin(w);

    FHD_r[j] += RP_r[i] * cw
              - RP_i[i] * sw;
    FHD_i[j] += RP_i[i] * cw
              + RP_r[i] * sw;
  }
}
```

GPU Code

```
local_x = x[threadIdx.x];
local_r = FHD_r[threadIdx.x];
local_i = FHD_i[threadIdx.x];

for(i=0 to max_K) {
  w = 2PI * dot(k[i], local_x);
  cw = cos(w); sw = sin(w);

  local_r += RP_r[i] * cw
           - RP_i[i] * sw;
  local_i += RP_i[i] * cw
           + RP_r[i] * sw;
}

FHD_r[threadIdx.x] = local_r;
FHD_i[threadIdx.x] = local_i;
```

The Compiler/Tools Challenge

"Compilers and tools must extend the human's ability to manage parallelism by doing the heavy lifting."

- To meet this challenge, the compiler must
  - Allow simple, effective control by programmers
  - Discover and verify parallelism
  - Eliminate tedious efforts in performance tuning
  - Reduce testing and support cost of parallel programs

Brief Overview of Architectural Features

- Threads are associated into 32-thread *warps*, which issue concurrently
- Threads are grouped into *blocks* of up to 512 threads which share a block of shared memory
- Hardware resources (thread contexts, registers, shared memory) allocated at per-block granularity
- Several memories

Key Performance Considerations

- Architecture provides hardware contexts for many more threads than execution resources
  - Execution throughput is the bottom line
- Categories of performance detractors
  - Stalls and bubbles in the pipeline
    - Port conflicts to shared/constant memory
    - Branch divergence
  - Long-latency operations
    - Need to run enough independent threads on the hardware to cover a thread’s latency with work from other threads
  - Shared resource saturation
    - Global memory bandwidth can be saturated
    - Especially if hardware cannot coalesce multiple loads/stores into fewer memory accesses

Machine Utilization Rules of Thumb

- Global memory load takes at least 200 cycles (estimated)
- Issuing an instruction for one warp takes 4 cycles (32 threads / 8-wide execution units)
- Need to issue at least 50 times (200 cycles / 4 cycles) to cover the latency
  - Issue independent instructions following the load
  - Issue instructions from other warps that are at a different PC
- To furnish enough threads for 24 independent warps, the kernel must be limited to
  - ≤10 registers per thread
  - ≤21 bytes of shared memory per thread
  - Most kernels we worked with required more resources than this
  - Completely hiding long latency operations is still tricky