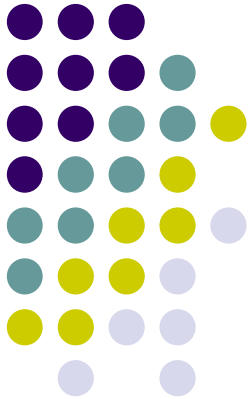
ME964
High Performance Computing for Engineering Applications

CUDA Optimization Tips (Hammad Mazhar)

Schedule related issues

March 22, 2011



"Adding manpower to a late software project makes it later".
Fred Brooks

Overview

- General Guidelines
  - What to do and What not to do
  - Debugging Tips
  - Compiler
  - Assembly
- Texture usage
- Using the profiler

What to do and what not to do

GENERAL GUIDELINES

What to do

- Use fast math operations when possible
- Waste a register rather than divide the same value multiple times
- When multiplying/dividing by powers of two use bitshifting
- Unroll loops that have a known size
- Inline simple (1/2 line) functions

What to do

- Max # of registers set to 32 by default
  - Properties for cuda wizard or build rule

  –maxrregcount=N
  - Forces compiler to use less or more registers
  - Extra registers spill to local memory
  - **Good**: use 32 registers rather than 33
    - More occupancy, usually faster
  - **Bad**: use 32 registers rather than 60
    - Too much local memory usage

What not to do

- Avoid double precision math where single precision is satisfactory
- Avoid division / modulo operators if possible
- Avoid static array declarations, compiler will (almost) always use lmem
  - Used shared memory if possible

What not to do

- Avoid Inlining large pieces of code, will cause local memory to be used unnecessarily.

- Avoid complex kernels that need many registers
  - Keep kernels simple
  - Split complex kernels to reduce register pressure

Tips For debugging

- If card is compute 2.0 use printf on device
  - cuPrintf might be useful for cards <2.0
    - look in SDK for code and example

- "Invalidate" code by putting:

  If(threadIdx.x==-1){ …code here…**}**
  - Prevents compiler from optimizing away code
  - Move statement until problem found

Tips For debugging

- Checking for execution errors:
- **CUDA_SAFE_CALL(…);**
  - Will terminate code with reference to line of code
  - Means that something before this call went wrong
- **CUT_CHECK_ERROR("ERROR MESSAGE");**
  - Prints out user specified string if something went wrong.

Compiler Info

- Compiler is smart about optimizing code
  - Takes care of register reuse
  - Combining math operations
    - Fused multiply add (MAD)
  - Delay global memory access until variable is actually used
  - Remove unused code
    - If a variable is computed but never used it gets removed at compile time

Compiler Info

- Compiler is not perfect
  - Reorganizing complex code manually can help
- Use **--ptxas-options=-v** for extra info
  - Shows info at compile time:

```
Compiling entry function '_Z8kernel_exPi' for 'sm_13'
Used 16 registers, 4 bytes lmem, 4+16 bytes smem, 4 bytes cmem[1]
```

  - Useful when optimizing register usage
    - don’t need to run code to see changes

11

Cuda Disassembler

- Look at what the compiler actually does
  - Assembly code is a bit tricky but can be followed
- **cuobjdump.exe –dump-sass prog.exe >out.txt**
  - Write assembly to out.txt
- Useful for making sure that memory reads and writes are optimized, fast math functions are used etc.

Example kernel

- Load 4 integers in single 128 bit (16 byte) load
- Do some math in a loop
- Store 4 integers in single 128 bit write

```
__global__ void kernel (int4* A, int reps){
      uint index=blockIdx.x*blockDim.x+threadIdx.x;
      for(int i=0; i<reps; i++){
            int4 temp=A[index];
            temp.x=temp.y*temp.z*temp.w;
            A[index]=temp;
      }
}
```


13

Example Assembly (1.0)

```
        Function : _Z8kernelP4int4i
/*0000*/    ISET.S32.C0 o [0x7f], g [0x5], R124,
LE;
/*0008*/    RET C0.NE;
/*0010*/    MOV.U16 R0H, g [0x1].U16;
/*0018*/    I2I.U32.U16 R1, R0L;
/*0020*/    IMAD.U16 R0, g [0x6].U16, R0H, R1;
/*0028*/    SHL R0, R0, 0x4;
/*0030*/    IADD R5, g [0x4], R0;
/*0038*/    IADD32I R0, R5, 0xc;
/*0040*/    GLD.U32 R4, global14 [R0];
/*0048*/    MOV R6, R124;
/*0050*/    GLD.S128 R0, global14 [R5];
/*0058*/    IMUL32.U16.U16 R3, R0L, R1H;
/*005c*/    IMUL32.U16.U16 R7, R4L, R2H;
/*0060*/    IMAD.U16 R3, R0H, R1L, R3;
/*0068*/    IMAD.U16 R7, R4H, R2L, R7;
/*0070*/    SHL R3, R3, 0x10;
/*0078*/    SHL R7, R7, 0x10;
/*0080*/    IMAD.U16 R0, R0L, R1L, R3;
/*0088*/    IMAD.U16 R3, R4L, R2L, R7;
/*0090*/    IMUL.U16.U16 R7, R0L, R3H;
/*0098*/    IMAD.U16 R7, R0H, R3L, R7;
/*00a0*/    SHL R7, R7, 0x10;
/*00a8*/    IADD32I R6, R6, 0x1;
/*00b0*/    IMAD.U16 R0, R0L, R3L, R7;
/*00b8*/    MOV R3, R4;
/*00c0*/    ISET.S32.C0 o [0x7f], g [0x5], R6, NE;
/*00c8*/    GST.S128 global14 [R5], R0;
/*00d0*/    BRA C0.NE, 0x50;
/*00d8*/    NOP;
```


14

Example Assembly (1.3)

```
        Function : _Z8kernelP4int4i
/*0000*/  ISET.S32.C0 o [0x7f], g [0x5], R124,
LE;
/*0008*/  RET C0.NE;
/*0010*/  G2R.U16 R0H, g [0x1].U16;
/*0018*/  I2I.U32.U16 R1, R0L;
/*0020*/  IMAD.U16 R0, g [0x6].U16, R0H, R1;
/*0028*/  SHL R0, R0, 0x4;
/*0030*/  IADD R5, g [0x4], R0;
/*0038*/  IADD32I R0, R5, 0xc;
/*0040*/  GLD.U32 R4, global14 [R0];
/*0048*/  MOV.SFU R6, R124;
/*0050*/  GLD.S128 R0, global14 [R5];
/*0058*/  IMUL32.U16.U16 R3, R0L, R1H;
/*005c*/  IMUL32.U16.U16 R7, R4L, R2H;
/*0060*/  IMAD.U16 R3, R0H, R1L, R3;
/*0068*/  IMAD.U16 R7, R4H, R2L, R7;
/*0070*/  SHL R3, R3, 0x10;
/*0078*/  SHL R7, R7, 0x10;
/*0080*/  IMAD.U16 R0, R0L, R1L, R3;
/*0088*/  IMAD.U16 R3, R4L, R2L, R7;
/*0090*/  IMUL.U16.U16 R7, R0L, R3H;
/*0098*/  IMAD.U16 R7, R0H, R3L, R7;
/*00a0*/  SHL R7, R7, 0x10;
/*00a8*/  IADD32I R6, R6, 0x1;
/*00b0*/  IMAD.U16 R0, R0L, R3L, R7;
/*00b8*/  MOV R3, R4;
/*00c0*/  ISET.S32.C0 o [0x7f], g [0x5], R6, NE;
/*00c8*/  GST.S128 global14 [R5], R0;
/*00d0*/  BRA C0.NE, 0x50;
/*00d8*/  NOP;
```

Branching Example

```
__global__ void kernel(int* data){
      if(threadIdx.x==0)
      {
            data[threadIdx.x]=1;
      }
      else if(threadIdx.x==1)
      {
            data[threadIdx.x]=2;
      }
}
```


16

Branching Assembly

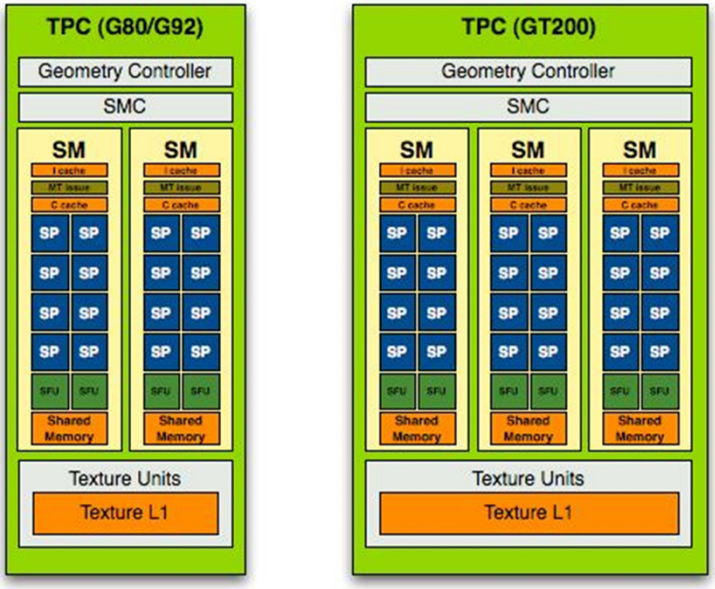
```
                Function : _Z8kernelPi
/*0000*/        I2I.U32.U16.C0 R0, R0L;
/*0008*/        BRA C0.NE, 0x38;
/*0010*/        SHL R1, R0, 0x2;
/*0018*/        MVI R0, 0x1;
/*0020*/        IADD R1, g [0x4], R1;
/*0028*/        GST.U32 global14 [R1], R0;
/*0030*/        RET;
/*0038*/        ISET.C0 o [0x7f], R0, c [0x1] [0x0], NE;
/*0040*/        RET C0.NE;
/*0048*/        SHL R1, R0, 0x2;
/*0050*/        MVI R0, 0x2;
/*0058*/        IADD R1, g [0x4], R1;
/*0060*/        GST.U32 global14 [R1], R0;
```

Easy way to speed up code

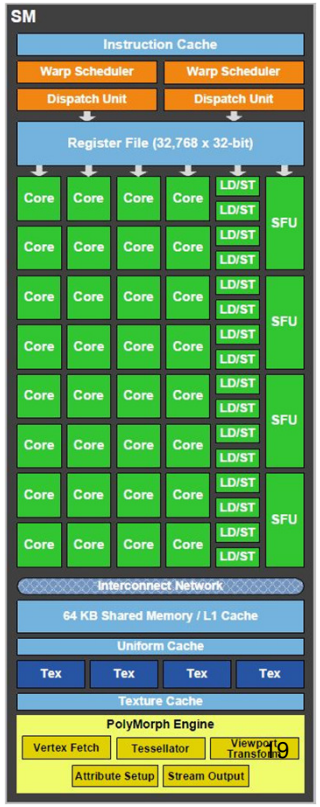
TEXTURE CACHE

The texture processor cluster

- Each TPC has several SM's and it's own texture memory
- No more TPC on Fermi Architecture



Fermi ->



Texture Memory

- A method of caching global memory reads
- Uses texture cache next to SMs
- Cannot write to a texture
  - i.e writing to global memory cannot be cached
- Useful if data access is random but data is reused by different threads in the same texture processor cluster or SM

“Binding” a texture (the simple way)

- Map global memory to a texture
- Allows mapped memory to be cached
- Keyword: **cudaBindTexture(…)**
  - **cudaUnbindTexture** to free

- Memory needs to be a linear array
  - 2D arrays/textures more complicated

21

Simple Example:

```
texture<int> texData;                    //global scope
...
int *devData
cudaMalloc((void**) & devData,size);
cudaBindTexture( NULL, texData, devData, size);
...                                      //Run kernel
cudaUnbindTexture(texData);
cudaFree(devData);

__global__ void kernel(...){
...=tex1Dfetch(texData,index);           //access
}
```


22

Complicated method Part 1

- Necessary if using 2D textures
- Useful for image processing
  - Image is essentially a 2D matrix
- Look in SDK for more examples

```
//Variable needs to be at global scope
texture<float, 2, cudaReadModeElementType> texData;

__global__ void kernel(...){
...=tex2D(texData, u, v);     //access element (u,v)
}
```

Complicated Method Part 2

```
// allocate array and copy data
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc
(32, 0, 0, 0, cudaChannelFormatKindFloat);

cudaArray* cu_array;
cudaMallocArray( &cu_array, &channelDesc, width,
height);

cudaMemcpyToArray( cu_array, 0, 0, h_data, size,
cudaMemcpyHostToDevice);
```


24

Complicated Method Part 3

```
// set texture parameters
texData.addressMode[0] = cudaAddressModeWrap;
texData.addressMode[1] = cudaAddressModeWrap;
texData.filterMode = cudaFilterModeLinear;
//access with normalized texture coordinates
texData.normalized = true;

// Bind the array to the texture
cutilSafeCall( cudaBindTextureToArray(texData,
cu_array, channelDesc));

//Texture read to use!!
```


25

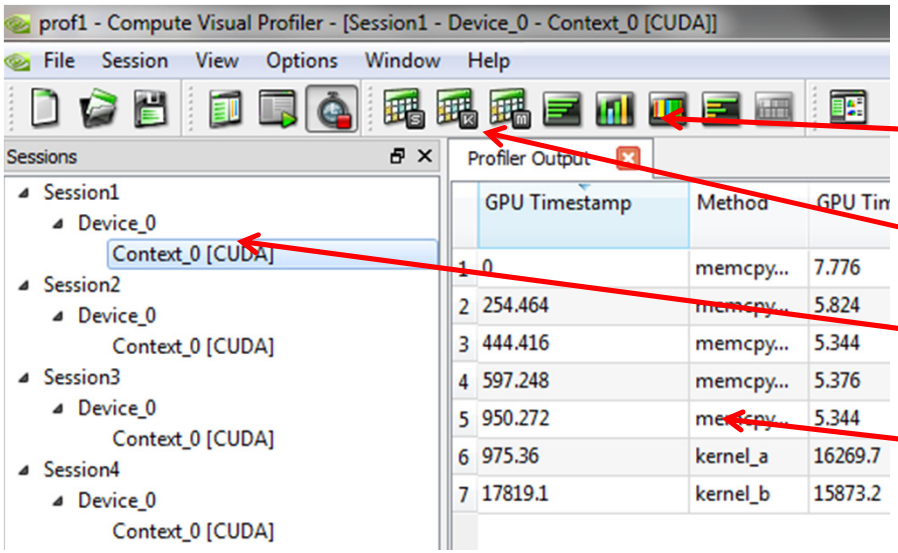
Using the Compute Visual Profiler

PROFILING CODE

Compute Visual Profiler

- Included in CUDA SDK
- Useful tool for profiling code
- Uses the GPU's built in counters
- Needs multiple passes
  - Each pass computes different parameters
- Only one SM is profiled
  - Some variables extrapolated

User Interface



- Plots
- Profiler Views
- Sessions
- Main Prof View

Profiler Output View

- GPU Timestamp
- Function Name
- GPU time
- CPU time
- Occupancy
- Grid/Block Size
- Shared Memory used per block
- Registers used
- Branched instructions
- Total Instructions

29

Summary Table

- Shows the amount of relative time each kernel took

Method	#Calls	GPU time	%GPU time	gld efficiency	gst efficiency	instruction throughput
kernel_a	1	413750	60.57	0.516387	0.491375	0.355267
kernel_b	1	269039	39.38	0.983576	0.919356	0.54636
memcpyHtoD	5	259.584	0.03			

30

Instruction Throughput

- Alternative to Occupancy
- Ratio of achieved instruction rate to peak single issue instruction rate.
- Calculated as:
  - gpu_time * clock_frequency / (instructions)
- Can be more than 1

Kernel, Memcopy Table Views

- Function Name
- # of Calls
- Grid / Block Size
- Shared memory per block
- Registers per thread
- Memory Transfer Type
- Memory Transfer Size

32

Plots

- GPU Time Summary Plot
- GPU Time Height Plot
- GPU Time Width Plot
- Comparison Plot

- Cuda API Trace Plot

End: CUDA Optimization Tips

Begin: Schedule Related Issues

34

Summary of Important Dates

- 03/29 – Midterm Project progress report due
- 04/06 – A one to two page PDF due, states your Final Project topic
- 04/11 – Three slides outlining your Final Project are due
- 04/13 – Midterm Project is due
  - Sample 2008 Midterm Project report available online
- 04/19 – Midterm Exam
- 05/09 – Final Project is due
- 05/10 – Individual presentations of Final Project results/outcomes

35

Midterm Project: Progress Report
[What's Needed…]

- You will have to provide an overview of the algorithm that you plan to implement.
- Things that I'm interested in:
  - Flow diagrams
  - Data structures that you plan to use
  - Explain how your algorithm maps upon the underlying SIMD architecture
  - Possible limiting factors that work against your solution implementation (for instance, if all threads executing a kernel need to synchronize, or to perform atomic operations, etc.)
  - Etc.
- Indicate the use of any third party CUDA libraries such as thrust, for instance.
  - The use of existing libraries is encouraged as long as they don't completely solve your problem...

Final Project Related

- Initial plan called for each one of you to make a five minute presentation of the Final Project topic you chose
- I will be out travelling on April 12, there will no class that Tuesday
- We will have a makeup class on May 3
  - Developer from MathWorks (MATLAB) will have a two hour lecture
    - First hour: GPU Computing in MATLAB
    - Second hour: Parallel Computing Toolbox and MDCS (MATLAB Distributed Computing Server)

Final Project Related

- One to two page PDF doc with your proposal for the Final Project due on 04/06
  - Use the Learn@UW drop-box for submission
- It should contain:
  - Problem statement
  - Reasons for choosing this Final Project topic and any preliminary results
  - Summary of outcomes and deliverables at the end of the semester (your contract with me)
- Prepare a presentation that has *three slides* :
  - First slide: your name, department, and problem statement
  - Reasons for choosing this Final Project topic and any preliminary results
  - Summary of outcomes and deliverables at the end of the semester (your contract with me)
- NOTE: I will compile all your presentations in one big presentation that I will go through on April 14 (20X3=60 slides)
  - It's important to use the same theme for the presentation, use the one I've been using throughout the semester (download a pptx from the class website and use it to generate your three slides…)

Schedule Highlights

- Two lectures dedicated to parallel computing using MPI
- Two lectures dedicated to parallel computing using OpenMP
- One lecture for Final Project discussions
- Midterm Exam 04/19
- Guest lectures at the end of the semester:
  - Matt Knepley – U of Chicago researcher , MPI (PETSC) related
  - Brian Davis – using cMake & Debugging CUDA
  - Ginger Ross – USAF researcher, discussion of HPC hardware, including a 500 TFlops machine USAF operates
  - Narfi Stefansson – MathWorks, GPU in MATLAB
  - Rob Farber – Pacific Northwest National Lab, GPU Computing