CUDA optimization

Philip Blakely

Laboratory for Scientific Computing, Cambridge

To consider first...

Is it worth optimizing it?

- In scientific computing, answer usually “yes”...
- unless it takes a lot of effort to implement ...
- or you’re only going to use it once.
- But - only optimize actual bottlenecks
- No point in optimizing part of the code that takes 1% of the run-time.
- Use the CUDA profiler `nvvp` to check where bottle-necks occur.

Are you making progress?

- Optimization can be fun - and time-consuming.
- Include timing to make sure optimizations actually work!
- Some techniques suggested here may either not make a large difference or may worsen performance if they affect some other aspect of the algorithm
- You need a feel of how different techniques can affect performance
- The techniques presented here are in rough order of potential improvements, starting with those most likely to give good speed-up.
- Start with a working code, and check at all stages whether the results are still correct.

What should be parallelised?

Amdahl's Law

Maximum speed-up is given by

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

where P is parallelizable fraction of code and N is no. of processors.

- Assuming N to be very large, if even 90% of the code is parallelizable, maximum speed-up possible is factor of 10.
- So, choose carefully which parts to parallelize...
- or, indeed, whether parallelization is worth it.

Change the algorithm

Outside the scope of this course...

- Best serial algorithm not necessarily the best parallel algorithm
- For example, there exist explicitly parallelised versions of sorting algorithms
- If you're following a one-element-per-cell approach, then this is *probably* the best approach
- Do some reading around your field/algorithm, see what has been done before
- Take care - a parallel algorithm suited to a small cluster may not scale well for massively parallel architecures.

Use more than one GPU

- Previously, a code could be sped up using multiple cores / CPUs
- This is still the case for GPUs
- See next lecture for combining MPI with GPUs
- Likely to give good speed-up

Device-Host memory transfer

- GPUs have their advantage in on-GPU memory-bandwidth, and in multi-processors
- Device$\leftrightarrow$Host memory transfers are relatively slow
- (slower than global memory to SM bandwidth by an order of magnitude)
- It may be an advantage to do more calculation on device rather than send data back to host and calculate there even if the extra calculation is not very well parallelisable

Global memory coalescence

- On-device memory-bandwidth is only attained if the access is coalesced
- Read the Programming Guide carefully
- Use the CUDA profiler
- Reconsider your data-layout
- Write small test-codes to optimize

Best not to access global memory too much:

- Make use of shared memory as much as possible
- Much faster access than global memory
- May be useful to do “in-place” operations (see euler.cu) to reduce shared-memory usage

Occupancy

- Recall that multiple thread blocks can reside on a multiprocessor and can be switched rapidly to avoid memory latency
- So multiple blocks can start while other blocks wait for memory reads
- So if many thread blocks can fit on a multiprocessor, we can hide latency
- The number of blocks that can be held on a multiprocessor is limited by
    - Shared memory use
    - Register use
- Reduce these to increase occupancy (but only up to about 25% is worth it)
- Low occupancy usually degrades performance

Calculating occupancy

Consider a typical NVIDIA GPU:

- Each multiprocessor has
  - 32,768 32-bit registers
  - 1536 simultaneous threads
  - 48KB shared memory
- so need < 22 registers per thread for 100% occupancy, i.e. all threads held in the SM simultaneously.
- and (for 512-thread blocks):
  $48 \times 1024 \times \frac{512}{1536} = 16384$ bytes of shared-memory per block

$$\text{Occupancy} = \frac{\begin{array}{c}\text{No. of active warps per multiprocessor,}\\ \text{given registers/shared memory}\end{array}}{\text{Max no. of warps per multiprocessor}}$$

Usually easier to use the occupancy spreadsheet
`/lsc/opt/cuda-8.0/tools/CUDA_Occupancy_Calculator.xls`

Check register and shared-memory usage

Compile using `-Xptxas -v` option:

```
nvcc euler.cu -o euler -O3 -Xptxas -v
...
ptxas info :  Compiling entry function
'_Z12maxWaveSpeedILj16ELj16EEv4GridPf4dim3' for 'sm_10'
ptxas info :  Used 11 registers, 1084+16 bytes smem, 40
bytes cmem[0], 32 bytes cmem[1]
```

means:

Function maxWaveSpeed uses

11 registers per thread,
1084 bytes user-allocated shared memory,
16 bytes shared memory for parameters (per block)
40 bytes of user-defined constant memory
32 bytes of compiler-generated constants in constant memory

Occupancy spreadsheet example

- Recall the Euler example from earlier
- The shared memory approach uses 4140 bytes of shared memory for 8×8 blocks and 36 registers
- Which of these affects occupancy?
- According to the Occupancy spreadsheet, both do.
- Reduce to 32 registers per thread: 3.79s
- Reduce y-dirn solve to 4×15 block (3884 bytes smem): 3.78s
- In this case, not a substantial difference.

Reduce registers

- Use `--maxrregcount` to force fewer registers to be used.
- Register overspill goes into local memory, which has same latency as global memory.
- However, if reducing registers increases occupancy, it can be beneficial.
- Hand-tuning the code to reduce registers can work
- Approximately equivalent to reducing local variables in function

Make the compiler do the work

- The compiler is very good at optimizing if you give it a chance
- Make as many constants available at compile-time as possible
- Can use templates to do this (see `euler.cu`)

```
template<int coord> void __global__ getFlux();

getFlux<0><<<gridDim, blockDim>>>();
```

- Branching and calculations dependent on `coord` can be optimized out.
- The CUDA compiler appears to optimize certain constructs fairly aggressively, such as constant-size loops
- Exact mathematical expressions such as $\sin(9.8)$ do not appear to be optimized out
  (there could be accuracy issues with doing so anyway)

Loop counters

- In loops, use signed integers, rather than unsigned.
- May be the opposite to what you thought
- Overflow on signed integers is undefined (in C++) so compiler can get away without overflow checks for signed arithmetic

```
for (i = 0; i < n; i++) {
    out[i] = in[offset + stride*i];
}
```

`stride*i` could overflow, so it's up to you to ensure that it doesn't.

Branching

- Recall that threads in a warp will run in step where possible.
- If different threads take different branches, threads taking different branches execute sequentially.
- So, make sure all threads take the same branch
- Or reduce amount of branching as much as possible.

Diagnose this using the CUDA visual profiler.

Precision

Warning

The following suggestions may well affect the accuracy of your code. Use them only if you *know* this is not a problem. Test your code with and without these suggestions and compare results.

- Use single precision unless `double` is absolutely necessary.
- Factor of 8 slowdown on older cards, factor of at least 2 even on latest cards
- Use the `--use_fast_math` compiler option - faster versions of `exp`, `sin`, `cos` etc.
- Use the `--ftz`, `--prec-dic`, and `--prec-sqrt` compiler options.

For those really concerned about IEEE 754 accuracy, see Appendix C of the Programming Guide.

Instruction counting

- When most other methods have been applied, instruction counting may be useful.
- The CUDA Programming Guide contains clock-cycle counts for various instructions (5.4.1)
- For example: (compute capability 6.0)
  - 32-bit floating-point add/multiply: 64 instructions/clock-cycle
  - 64-bit floating-point add/multiply: 32 instructions/clock-cycle
  - 32-bit floating-point reciprocal, reciprocal square-root, sine/cosine: 16 instructions/clock-cycle
- Type conversions: 16 instructions/clock-cycle

General rule: Reduce the number of expensive operations.
Use temporary variables to help the compiler if necessary.

Special functions

- Can sometimes replace branching by special floating point or bit-wise operations
  - `copysign(x,y)` and `signbit(x)`
  - `sincos(x, sptr, cptr)`
- For integer operations: See CUDA_Math_API.pdf
  Includes bit-counting functions such as `__popc`.
- Or search for "Bit Twiddling Hacks" (S.E. Anderson):
  `r = y ^ ((x ^ y) & -(x < y)); // min(x, y)`
  (Interesting read even if you don't need these techniques)

C++

C++ can be used to make code more readable while allowing compiler to optimize as usual

Using expression templates:

`a = b + 3.2 * c - d*e;` can be expanded to

```
__global__ (float* a, float* b, float* c, float* d,
            float* e, int N)
{
  const int i = ...;
  a[i] = b[i] + 3.2f * c[i] - d[i]*e[i];
}
```

by the compiler alone.

See "C++ Templates" - Vandevoorde & Josuttis for details

Optimization approaches

First - check whether algorithm is bandwidth or instruction bound

- Imagine two stripped-down versions of the code:
  1. No calculation - just appropriate memory reads/writes
  2. No memory reads/writes - just calculation (be careful the compiler can't optimize them away)
- Compare times for all three versions (including original)
- In the ideal case, the full version is much faster than sum of its parts
- Why? Executing instructions can hide memory latency
- See whether memory access or calculations take longer, and optimize accordingly
- In practice, the Visual Profiler does an equivalent check for you when using Guided Profiling.

Bandwidth bound

If your kernel is bandwidth bound:

- Check for global-memory read coalescence
- Use shared memory as programmer-designed cache
- Use constant memory if there are e.g. stencil coefficients used in all kernels
- Rethink data-structures - Struct-Of-Arrays versus Array-Of-Structs

Compute-bound

If your kernel is compute bound (less likely):

- Strength-reduction
  1. Replace multiplication by shift operation or addition if possible
  2. Replace division by multiplication by reciprocal
  3. Reduce number of operations - use algebraic identities if possible

     ```
     a*x*x*x + b*x*x + c*x + d == d + x*(c + x*(b + a*x))
     ```

  4. Reduce use of `pow()` if possible
  5. Avoid expensive recomputation - precompute values and store in temporary variables
- Reduce (divergent) branching

Latency-bound

If sum of memory-access and compute versions is about the same as standard kernel, you're latency bound, i.e. the hardware cannot overlap the memory-access and computation.

- Move memory-reads to early in kernel - allows instructions that don't need data to execute and hide memory-read time
- Reduce number of `__syncthreads()` - while maintaining correctness
- Increase thread-block occupancy
  - Reduce shared-memory use
  - Reduce register-use
- (Allows threads to be swapped out while their data is fetched from global memory).
- Increase the amount of data processed per kernel launch

Shared memory

- There is an L1 cache at the same level as shared memory
- There is 64kB of shared-memory / L1 cache available:
  - 48kB of shared-memory and 16kB L1 cache
  - 16kB of shared-memory and 48kB L1 cache
- Selectable at run-time by

  ```
  cudaDeviceSetCacheConfig(config)
  ```

  where `config` is one of:
  - `cudaFuncCachePreferNone`
  - `cudaFuncCachePreferShared`
  - `cudaFuncCachePreferL1`
  - `cudaFuncCachePreferEqual`

  and sets a preference only.
- This may be over-ridden if, for example, a kernel requires more shared-memory than 16kB.

Shared memory

- Later cards may have more shared memory per Streaming Multiprocessor
- Cards of CC 3.7 have 96kB shared memory per SM, but still limit to 64kB per thread block
- This allows increased occupancy
- Cards of CC 5.2 have 96kB shared memory.
- The `cudaDeviceGetAttribute()` function is available to test for maximum shared memory.

Dynamic shared memory

- Instead of fixing the amount of shared memory used by a kernel at compile time, it can be determined at run-time
- If you have a shared-memory array declared at file-scope:

  ```
  extern __shared__ float myArray[];
  ```

  and call a kernel as:

  ```
  kernel<<<gridDim, blockDim, sharedSize, stream>>>(...);
  ```

  then the pointer `myArray` will point to a block of shared-memory of size `sharedSize` bytes.
- (Note that `stream` is optional.)
- The Guide is not explicit on what happens if there is more than one `extern __shared__` pointer.

Dynamic global memory

- It is possible to dynamically allocate heap memory in a kernel:

```
__global__ f(...){
float* arr = (float*)malloc(sizeof(float)*N);
free(arr);
}
```

- The memory must be both allocated and freed by a kernel; it cannot be freed from the host
- The maximum heap-size must be set on the host using `cudaDeviceSetLimit()`.
- It is permitted to allocate shared memory using this method as well.
- It is probably safer (and certainly more efficient) to allocate all global memory on the host and to know about shared memory size at compile-time

C++ support

- Recent NVIDIA cards all support full C++ (arbitrary pointer dereference)
- Unified address space for all variables and pointers
- Virtual functions, function pointers are allowed on device
- `new` and `delete` are supported (probably not efficiently)

However, you should probably keep code at this level on the CPU and use the GPU for pure computation.

Software development

- If writing anything longer than a single algorithm, see Nick Maclaren's lectures
- `nvcc` can compile all C++ code
- C++ control code should be kept separate from kernels and kernel calls
- Run-time API calls can be put in C++ code and compiled with `gcc` - suffix `.C`
- Kernels and the `<<<...>>>` calls should be put in `.cu` files and compiled with `nvcc`
- A `__device__` or `__global__` function does not have to be in the same file as the call to it.

Linking across object files

- Early versions of CUDA Toolkit did not allow calling of a kernel from a different file (i.e. no linking for `__device__` functions.
- When compiling an object file with device code, use:
  `nvcc -dc a.cu -o a.o`
  Similar to `-c` option for `gcc`
- When linking separate objects containing device code, use:
  `nvcc -dlink a.o b.o -o myProg`
- However, the CUDA compiler now cannot inline device functions called between files, and must introduce a general function-call stack.
- This results in more register usage.
- This may significantly impact performance (e.g. reduce by a third).

Compiler options

Read `CUDA_Compiler_Driver_NVCC.pdf` for compiler options

- `-O<n>` - Optimize to level n. Typically use $n = 3$
- `-arch` *gpuarch* Assume given GPU architecture:
  compute_50, compute_52, compute_60
  when optimizing
- `-code` *code* Compile code for target GPU compute capability:
  sm_50, sm_52, sm_52, sm_60

CUDA Libraries

As with all programming, the best approach is not to write any code! NVIDIA ships some libraries as part of the CUDA Toolkit:

- Thrust - replacement for C++ STL
- CUBLAS - BLAS for CUDA
- CUSPARSE - Sparse matrix calculations
- CUSOLVER - Covers BLAS, Sparse, and Sparse refactorization
- CUFFT - FFT calculations
- CURAND - Random number generation
- nvGRAPH - Graph creation and manipulation

CUDA Libraries ctd

And some are available from NVIDIA's GitHub repository:

- GVDB-voxels - Sparse volume compute and rendering
- AMGX - Multigrid (elliptic) linear solvers
- NCCL - Inter-GPU communication algorithms
- CUTLASS - matrix-multiplication via C++ templates
- Kokkos - C++ abstractions for running CUDA code, e.g. array classes etc.
- ... and various Deep Learning algorithms.

Thrust

http://docs.nvidia.com/cuda/thrust/index.html

C++ has STL for various algorithms on containers:

- Sorting a list of strings
- Summing over a list of integers
- Various operations on stacks, heaps, maps.

Thrust creates versions of these that run on a GPU.

```
// Taken from Thrust examples
#input <thrust/count.h>
#input <thrust/device_vector.h>
...
// put three 1s in a device_vector
thrust::device_vector<int> vec(5,0);
vec[1] = 1;
vec[3] = 1;
vec[4] = 1;

// count the 1s
int result = thrust::count(vec.begin(), vec.end(), 1);
// result == 3
```

cuBLAS

- BLAS (Basic Linear Algebra Subprograms) is a library of functions for operating on vectors and matrices.
- Includes operations on symmetric, triangular, and Hermitian matrices.
- Well-defined standard and interface for FORTRAN, C, and C++.
- First published in 1979, many implementations exist
- `cuBLAS` implements some of these functions in a precompiled CUDA library - available with `<cublas.h>` header and `-lcublas` library.
- If you only need vector/matrix operations - use this!

cuBLAS ctd

cuBLAS example

```
cublasAlloc(N, sizeof(float), (void **)&x);
cublasAlloc(N, sizeof(float), (void **)&y);

cublasSetVector(N, sizeof(float), xHost, 1, x, 1);
cublasSetVector(N, sizeof(float), yHost, 1, y, 1);

clock_t start = clock();
float tot = 0;
for(int i=0 ; i < M ; i++)
{
  tot += cublasSdot(N, x, 1, y, 1);
}

clock_t end = clock();
```

cuFFT

- Fast Fourier Transforms are a common application.
- `cuFFT` implements these in CUDA - use `cufft.h` header and `-lcufft` library.
- Includes:
  - 1D, 2D, 3D FFTs - real and complex data
  - In-place and out-of-place transforms
  - Double-precision on compatible hardware

Jacket

- Jacket - CUDA wrapper for Matlab
- Allows most Matlab operations to be carried out on GPU

-
  ```
  >> GPU_matrix = gdouble( CPU_matrix );
  >> GPU_matrix = fft( GPU_matrix );
  >> CPU_matrix = double( GPU_matrix );
  ```
- Drawback - costs money (but then, so does Matlab)...
- See `www.accelereyes.com` for more details

Summary

More information is available from:

- CUDA Best Practices Guide (similar material to but more depth than this lecture)
- Whitepapers in the CUDA SDK (usually well written)
- Research articles on CUDA (may contain useful code snippets)
- Source code for CUDA applications often made freely available by authors (harder to read).