A Strategy for Automatically Generating High Performance CUDA Code for a GPU Accelerator from a Specialized Fortran Code Expression

Pei-Hung Lin, Jagan Jayaraj, and Paul R. Woodward

University of Minnesota
Spectrum of Processor Architecture

• Many architectures out there

• From computing perspective, not very different though!

• Classification parameter
  – Ratio of on-chip memory to peak processing power
Programming for Performance

Decreasing Ratio of On-chip Memory to Processing Power

CPU  Cell  GPU

Increasing Programming Burden
General Programming Strategy

• Improve data re-use on-chip

• Reduce memory traffic

• Increase the computation to communication ratio

• Vectorize the code
General Programming Strategy

1. Prefetch a small block of data to process
2. Unpack the data to produce a number of short vectors that reside in the on-chip memory
3. Update the data via many vector floating point operations
4. Repack the updated data into a new data record
5. Write the new record back to main memory
GPU Specific Issues

• This same procedure should work equally well on the GPU’s SM, except that its on-chip memory may not be able to accommodate all the vector temporaries involved in the computation in step 3.

• Our programs must be modified so that they essentially use only the register file and do not rely upon the local store for data.

• We need to control the allocation of this precious on-chip “register” storage.
Why Explicit Memory Management?

• We could let the CUDA compiler allocate our register storage for us in this model. But we would be unable to exploit the 16 KB shared memory.

• Also, the shared memory must be used if we are to index our on-chip vectors with offsets.
Memory Management

• All vector temporaries are mapped to two sets of effective register variables.
• r01, r02, . . . , will be in register variables; the other set, s01, s02, etc., will be in shared memory. We will allow 20 register and shared memory variables in each thread.
• Temporaries are moved from one memory set to the other as needed, and it must spill these variables to global memory and retrieve them as needed.
Memory Management

• Using the 40 different “register variables” (20 from register and 20 from shared memory) exclusively, code will become entirely unreadable.

• Code transformation through automatic code translation becomes essential.
Preliminary Results

- **Benchmark:**
  - Code fragment from the original PPM code
  - Represents roughly half the flops in the original single-fluid PPM gas dynamics code

- In 64-way SIMD Fortran form, it achieves 11.7 Gflop/s/core (50% of peak) on the Intel Nehalem quad-core processor (2.93 GHz).
Preliminary results on the GPU

• With 64 threads per thread block and one thread block per each SM on Nvidia Tesla C1060, it achieves 49 Gflop/s.

• When 3 thread blocks execute simultaneously on each SM, each thread block takes 21 registers/thread and 4660 bytes of shared memory, and achieves 100 Gflop/s.
Ongoing Experiments

- Godunov hydrodynamics code
- Grid resolution: 32x96x32
- Before our code conversion:
  - 64 threads per thread block
  - 2 thread blocks per SM
  - 63 registers per thread
  - 23 KB shared memory per SM
Rayleigh-Taylor Instability
Rayleigh-Taylor Instability
Why translators?

• Manual code optimizations and memory management are extremely difficult
• Back-end we built for Cell processors insulated us from writing tedious code with SPU intrinsics
• Exciting to build new tools than to repeat the same transformation over and over again!
Translator

- ANTLR based translator
- Translator parses programs into abstract syntax tree (AST) and maintains symbol tables on its own.
- Code transformations are directed by user directives inserted in source code.
- Un-parse transformed tree into target platforms (Fortran, C, CUDA…etc).
Translator status

- Current features
  - Inlining
  - Unrolling
  - Loop fusion
  - Temporary array reduction
  - Cell C and Fortran Altivec (Power7) back-ends
Translator status

• Work in progress / To be done
  – Pipelining
  – Variable liveness analysis
  – Memory management
  – CUDA back-end
c IDEC$ VECTOR ALWAYS
 c IDEC$ VECTOR ALIGNED
 c do 1105  I = -3,n+5
   do j = 1,16
     duyl(i,jam0) = uy(i,j7m0) - uy(i,j7m1)
     abs duyl(i,ibm0) = abs(duyl(i,jam0))
     duyl(i,jam1) = uy(i,j7m1) - uy(i,j7m2)
     abs duyl(i,ibm1) = abs(duyl(i,jam1))
     duyl(i,jam2) = uy(i,j7m2) - uy(i,j7m3)
     abs duyl(i,ibm2) = abs(duyl(i,jam2))
     duyl(i,jam3) = uy(i,j7m3) - uy(i,j7m4)
     abs duyl(i,ibm3) = abs(duyl(i,jam3))
   enddo
1105 continue
 c
c loop1100s = loop1100s + 1
 c
c ninloop = loop1100s * 48
 c ncvnngms = ncvnngms + ninloop
 c nadds = nadds + ninloop
 c nflops = nflops + 48
 c The last two lines of the loop, which have been commented out,
can be executed if we are on a strict SIMD computing platform.
Otherwise, it is more efficient to leave them out.
c
go to 9000
 endif
 c
 c LABEL was 9000
endif
 c dec$ vector always
   do j = 1,16/4,1
     duyl(i,jam0) = (vec_sub(uy(i,j7m0),uy(i,j7m1)))
     abs duyl(i,ibm0) = vec_abs(duyl(i,jam0))
     duyl(i,jam1) = (vec_sub(uy(i,j7m1),uy(i,j7m2)))
     abs duyl(i,ibm1) = vec_abs(duyl(i,jam1))
     duyl(i,jam2) = (vec_sub(uy(i,j7m2),uy(i,j7m3)))
     abs duyl(i,ibm2) = vec_abs(duyl(i,jam2))
     duyl(i,jam3) = (vec_sub(uy(i,j7m3),uy(i,j7m4)))
     abs duyl(i,ibm3) = vec_abs(duyl(i,jam3))
   enddo
   nflops = (nflops + 48)
goto 1
 c LABEL was 9000
endif
 c dec$ vector always
   do j = 1,16/4,1
     duyl(i,jam0) = (vec_sub(uy(i,j7m0),uy(i,j7m1)))
     abs duyl(i,ibm0) = vec_abs(duyl(i,jam0))
     duyl(i,jam1) = (vec_sub(uy(i,j7m1),uy(i,j7m2)))
     abs duyl(i,ibm1) = vec_abs(duyl(i,jam1))
     duyl(i,jam2) = (vec_sub(uy(i,j7m2),uy(i,j7m3)))
     abs duyl(i,ibm2) = vec_abs(duyl(i,jam2))
     duyl(i,jam3) = (vec_sub(uy(i,j7m3),uy(i,j7m4)))
     abs duyl(i,ibm3) = vec_abs(duyl(i,jam3))
   enddo
   nflops = (nflops + 64)
endif
 c dec$ vector always
   do j = 1,16/4,1
     adiff = vec_sub(uy(i,j7m1),uy(i,j7m3))
     azdiff = vec_sub(uy(i,j7m3),uy(i,j7m0))
     duyl(i,jam0) = vec_add(duyl(i,jam0),adiff)
     duyl(i,jam1) = vec_add(duyl(i,jam1),adiff)
     duyl(i,jam2) = vec_add(duyl(i,jam2),adiff)
     duyl(i,jam3) = vec_add(duyl(i,jam3),adiff)
     duyl(i,ibm0) = vec_add(duyl(i,ibm0),azdiff)
     duyl(i,ibm1) = vec_add(duyl(i,ibm1),azdiff)
     duyl(i,ibm2) = vec_add(duyl(i,ibm2),azdiff)
     duyl(i,ibm3) = vec_add(duyl(i,ibm3),azdiff)
   enddo
File testcode.f was deleted!!
File test_lineContRemoved.f was deleted!!
(cppm$ dma) -> (cppm$ DMA)
(cppm$ dma) -> (cppm$ DMA)
(cppm$ dma) -> (cppm$ DMA)
Fortran output file: output_orig.f was generated!!
Fortran output file: output_SIMD.f was generated!!
Conclusion

• We are implementing the Godunov hydrodynamics example and hope to achieve the same 100 Gflop/s performance as the fragment code.

• Because of the much smaller on-chip memory on the GPU relative to its potential computational capability, a strategy for a form of register allocation and code rearrangement that minimizes the number of registers used is necessary.