Auto-generation of WRF code for GPUs

Thomas Nipen (tnipen@eos.ubc.edu) • **John Michalakes** (michalak@ucar.edu)

UBC NCAR

1. Introduction

The massively parallel architecture of graphical processing units (GPUs) can be used to speed up numerical weather prediction. Part of the challenge in using this technology is converting existing Fortran code to Compute Unified Device Architecture (CUDA) accelerated code.

We present findings from using a NOAA-developed translator to accelerate WRF on GPUs. The translated code is hand-tuned for optimal performance.

We also present initial experiences from using a pre-release version of the Portland Group Fortran compiler for accelerator hardware.

2. WRF SW radiation

We have targeted the Dudhia shortwave radiation scheme [1] for acceleration on GPUs. This scheme, comprising 600 lines of Fortran code, allows for cloudy and clear air absorption and scattering. The vertical columns are data independent.

We created a standalone shortwave radiation program, which simulates one call to the radiation scheme. The domain has horizontal dimension of 61 east-west and 74 north-south cells, and 27 vertical levels covering the Eastern United States.

3. GPUs

Graphical processing units have traditionally been used to accelerate the rendering of graphics in programs. Their special architecture can be exploited for non-graphics applications with high parallelism and arithmetic intensity. The NVIDIA line of GPUs have been investigated for their potential to accelerate the microphysics portion of WRF [2].

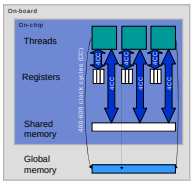
These GPUs achieve their computational advantage by saturating the many floating point units with concurrent threads. The memory latency of the large on-board global memory is quite high and accesses should therefore be minimized. Furthermore, the accesses should be such that the call can be coalesced, otherwise the number of memory transactions will be increased, effectively reducing the bandwidth.

The cards have on-chip registers and shared memory. Registers are per-thread-based and take 4 clock cycles to access. Shared memory are per-block-based and are equally fast, as long as access by threads do not cause bank conflicts. Conflicting memory accesses are serialized.

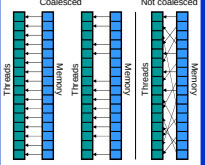
The NVIDIA cards can be programmed using a high level C-like language called CUDA. For more information visit: http://www.nvidia.com/

NVIDIA memory layout

NVIDIA GPUs have several layers of memory of varying capacity and latency:



Global memory accesses



4. PGI Accelerator Fortran & C99 compilers

http://www.pgroup.com/resources/accel.htm

The Portland Group is working on extending their Fortran and C compilers to compile for GPUs [3]. They use a directive-based approach, where compiler directives are placed in the source code to guide the compiler towards creating accelerated code.

The compiler directives are similar in style to OpenMP. `!$acc region` and `!$acc end region` are used to delineate sections of the code to be accelerated for GPUs. Without further guidance, the compiler will try to determine a way to parallelize the DO loops contained within `region` and `end region`. Wherever possible, the compiler will determine which variables must be passed to and from the GPU, and which variables must be temporarily allocated on the GPU.

Data movement and loop mappings can be hand-tuned by the developer using additional directives, for example to specify how loops should be parallelized. The compiler can aid in this by providing warnings of loops that cannot be parallelized due to data dependencies.

The compiler can also list arithmetic intensity for each for loop, which can be useful in determining how to parallelize the code. Loops with low arithmetic intensity are less likely to be successful in being accelerated because of the higher relative number of memory accesses.

The major advantage of this approach is that the compiler does most of the work associated with porting to GPUs. Also, two separate development trees, one in Fortran and one in CUDA, are not needed. This comes at the expense of having less control of the behavior of the GPU, and may in some cases limit the speedup possible.

At the time of writing, a working version of the SW code with accelerator directives was not finalized. However, we expect this to be a fruitful avenue of research.

Code comparison (excerpt)

Fortran code with directives

```
153 !$acc region
154 j_loop: DO J=jts,jte
155 i_loop: DO I=its,ite

...

272     DO K=kme-1-kte+kms, kme-1-kts+kms
273       ! NK=kme-1-K+kms
274 ! P in the unit of 10mb
275       RO(I,K,J)=P3D(I,K,J)/(R*T3D(I,K,J))
276       XWVP(I,K,J)=RO(I,K,J)*QV3D(I,K,J)*dz8w(I,K,J)*1000.
277 ! KG/M**2
278       XATP(I,K,J)=RO(I,K,J)*dz8w(I,K,J)
279   ENDDO

...

316     DO 200 NK=kme-1-kte+kms, kme-1-kts+kms
317       !NK=kme-1-K+kms
318       WW=WW+XLWP(I,NK,J)
319       UV=UV+XWVP(I,NK,J)
320 ! WGM IS WW/COS(THETA) (G/M**2)
321 ! UGCM IS UV/COS(THETA) (G/CM**2)
322       WGM=WW/XMU
323       UGCM=UV*0.0001/XMU
324 !
325       OLDABS=TOTABS
326 ! WATER VAPOR ABSORPTION AS IN LACIS AND HANSEN (1974)
327       TOTABS=2.9*UGCM/((1.+141.5*UGCM)**0.635+5.925*UGCM)
```

Compiler output

```
153, Generating copyin(qv3d(its:ite,kts:kte,jts:jte))
     Generating copyin ...

155, Loop is parallelizable
157, Loop is parallelizable
     Accelerator kernel generated
     155, !$acc do parallel
     157, !$acc do vector
          Non-coalesced memory accesses for albedo
          Non-coalesced memory accesses for sdown
272, Intensity = 1.25
     Loop is parallelizable

316, Intensity = [symbolic], and not printable,
          try the -Mpfi -Mpfo options
     Loop carried scalar dependence for ww
     Loop carried scalar dependence for uv
     Loop carried scalar dependence for totabs
     Complex loop carried dependence of sdown prevents
       parallelization
     Loop carried dependence of iil prevents
       parallelization
     ...
```

Summary of directives*

- `!$acc region` Generate accelerated region.
- `!$acc do parallel` Following `do` loop should be organized in a grid.
- `!$acc do vector` Following `do` loop should be organized in blocks.
- `!$acc do seq` Following `do` loop should be run sequentially on the GPU.
- `!$acc do host` Following `do` loop should be run sequentially on the host.
- `!$acc copyin(var)` Copy `var` from host to GPU.
- `!$acc copyout(var)` Copy `var` back from GPU to host.
- `!$acc local(var)` Declares `var` to reside purely on the GPU, and will not be copied back or forth.
- `!$acc end region` End of accelerated region

* This list is not exhaustive

5. Fortran to CUDA translator

http://www-ad.fsl.noaa.gov/ac/Accelerators.html

This tool, developed by Mark Govett at NOAA, removes the tedious process of translating Fortran code into CUDA code. Transferring data and setting up the kernel launch is handled fully by the translator. Issues related to parallelization are left for the users to define for themselves. The number of threads per block, shared memory, and coalesced memory accesses must be dealt with by hand. Further hand modifications are necessary for features that the translator does not support.

Although some modification will inevitably be necessary, this tool can save a tremendous amount of development time.

Below is a comparison of the original code and the CUDA code generated by the translator.

Kernel code comparison (excerpt)

Fortran

```
j_loop: DO J=jts,jte
i_loop: DO I=its,ite


DO K=kts,kte
   NK=kme-1-K+kms
   TTEN1D(K)=0.

   T1D(K)=T3D(I,NK,J)
   P1D(K)=P3D(I,NK,J)
   RHO81D(K)=rho_phy(I,NK,J)
   DZ(K)=dz8w(I,NK,J)
ENDDO

IF (PRESENT(F_QV) .AND.          &
    PRESENT(QV3D)) THEN
   IF (F_QV) THEN
      DO K=kts,kte
         NK=kme-1-K+kms
         QV1D(K)=QV3D(I,NK,J)
         QV1D(K)=max(0.,QV1D(K))
      ENDDO
   ENDIF
ENDIF
```

CUDA (`modifications for parallelization`)

```
//for (j_=jts;j<=jte;j++) {
//    for (i=its;i<=ite;i++) {
i = blockIdx.x * blockDim.x + threadIdx.x + 1;
j = blockIdx.y * blockDim.y + threadIdx.y + 1;

for (k=kts;k<=kte;k++) {
   nk = kme - 1 - k + kms;
   ttensd[FTNREF1D(k,kts)] = 0.;

   t1d[FTNREF1D(k,kts)] = t3d[FTNREF3D(i,nk,j,di,dk,ims,kms,jms)];
   p1d[FTNREF1D(k,kts)] = p3d[FTNREF3D(i,nk,j,di,dk,ims,kms,jms)];
   rho81d[FTNREF1D(k,kts)] = rho_phy[FTNREF3D(i,nk,j,di,dk,ims,kms,jms)];
   dz[FTNREF1D(k,kts)] = dz8w[FTNREF3D(i,nk,j,di,dk,ims,kms,jms)];
}

//if ( present(f_qv) && present(qv3d) ) {

   if ( f_qv ) {
      for (k=kts;k<=kte;k++) {
         nk = kme - 1 - k + kms;
         qv1d[FTNREF1D(k,kts)] = qv3d[FTNREF3D(i,nk,j,di,dk,ims,kms,jms)];
         qv1d[FTNREF1D(k,kts)] = MAX(0.,qv1d[FTNREF1D(k,1)]);
      }
   }
//}
```

Translator features

- Makes required cudaMalloc and cudaMemcpy calls
- Sets up execution configuration of kernel
- Converts multi-dimensional Fortran arrays to 1D CUDA arrays with explicit indexing

Hand modifications required

- Define block sizes
- Define grid dimensions
- Replace do loops with thread indices
- Use shared memory (if desired)
- Restructure code (if necessary)

6. Hand-tuning for performance

We took the translated code and tested various hand optimizations. Our overall strategy was to minimize the number of expensive memory accesses.

The original shortwave code copies the 3D variables into 1D variables for each column. This was likely done to avoid having the subroutine alter the input arrays. However, on the GPU the arrays are already duplicated from main memory, so any alterations are not transferred back. This causes a lot of extra expensive memory read and writes on the GPU. By removing the 1D arrays, the memory accesses are guaranteed to coalesce.

Shared memory can also be exploited. With 64 threads per block, the 16KB of shared memory per block can only hold two arrays with 27 levels. 6 calls (per vertical level) were made to the `dz8w` (grid spacing) variable, thus 4 global memory accesses (per vertical level) can be saved by copying this variable to shared memory. No other variable was reused enough to make it worth the move to shared memory.

The effect of these changes are shown in section **7. Results**.

Fine tuning	Description
Removing 1D arrays	Removed copies made of 3D arrays into local 1D arrays. This lower the number of memory reads to global memory and guarantees coalesced read/writes.
Shared memory	Placed as many variables in shared memory as possible. Faster memory reads/writes.

7. Results

We compared the performance of one Tesla C870 card hosted by an Intel Xeon E5420 processor.

We saw good computational speedup on the device. As the SW routine is quite small, the transferring of data back and forth to the GPU limits any overall speedup attained by the GPU. We expect that when combined with the computationally expensive RRTM longwave radiation scheme (around 650ms execution time), the transfer time can be hidden. Furthermore, the launch of each kernel call has overhead, which can also be hidden.

Ignoring transfer and initialization time, the raw translated code gave a 3.3x speed improvement relative to the CPU-only configuration.

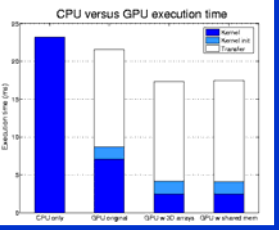
Architecture comparison

Architecture	Tesla C870	Intel Xeon E5420 *
Clock rate	1.35Ghz	2.50Ghz
Total memory	1.50GB	2.00GB**
Shared memory	16KB/block	

* using 1 out of 4 cores
** per core

Removing 1D arrays and coalescing memory calls brings the speedup to 9.5x, and moving `dz8w` to shared memory gave a 9.6x speed improvement.

The small added speed improvement of using shared memory can be attributed to the fact that `dz8w` was not very heavily reused.



Acknowledgment

We wish to thank Prof. Manish Vachharajani and NVIDIA for access to a GPU cluster at the University of Colorado.

References

[1] Dudhia, J., 1989: Numerical study of convection observed during the winter monsoon experiment using a mesoscale two-dimensional model, *J. Atmos. Sci.*, **46**, 3077–3107

[2] Michalakes, J. and M. Vachharajani, 2008: GPU Acceleration of Numerical Weather Prediction. *Parallel Processing Letters* Vol. 18 No. 4. World Scientific.. pp. 531--548.

[3] The Portland Group Inc., 2008: PGI Fortran & C Accelerator Compilers and Programming Model Technology Preview.