Porting Large Fortran Codebases to GPUs

Andrew Corrigan and Rainald Löhner

Center for Computational Fluid Dynamics
Department of Computational and Data Sciences
George Mason University
Fairfax, VA USA
Collaborators

• Eric Mestreau
• Fernando Camelli
• Fernando Mut
Outline

• Background (FEFLO)
• Porting Strategies
• Performance
• Parallelization
• Array tracking
• Misc. Issues
• Results
• Conclusions
FEFLO

A large-scale, actively developed and deployed, legacy, Fortran computational fluid dynamics code
FEFLO-GPU
Goals

• Full GPU performance
• Port ~1 million lines of code (~11,000 parallel loops)
• Continue development in Fortran using established coding practices.
• A single, unified codebase.
Porting Strategies

- Manual porting
  - Rewrite the code from scratch

- Automatic porting
  - Use a translator or compiler to largely automate the task.
Manual Porting

• Too much effort required.
  o 1 million lines / 11,000 parallel loops.
  o Intricate array bookkeeping
• Perpetual process
  o Fortran development will continue
• Error Prone.
• Separate codebases
• CUDA or OpenCL or CUDA Fortran or ?

Not Feasible
Automatic Porting

• Continued Fortran development.
• Single codebase.
• Reliable: No new bugs.
  • Either works perfectly,
  • Or fails catastrophically (easy to catch).
  • Actually, catches many old bugs.
• Supports CUDA.
  • Excellent option now: maturity, library support.
  • CUDA Fortran and OpenCL are partially supported.
  • Extensible to future platforms.
Using a Python script

- O(1000) line Python script based on FParser
  - Developed in a few months.
  - Generates an optimized, running code.
    - Does much more than translate loops in isolation.
  - Generates CUDA kernels from existing OpenMP and vector loops.
- Tracks array usage across the entire code.
  - By far the most difficult task.
- Many other tasks.

FParser: [http://code.google.com/p/f2py](http://code.google.com/p/f2py) [Peterson, 2009]
Performance

The performance issues of primary concern for GPUs are

- Achieving fine-grained parallelism.
- Avoiding CPU $\leftrightarrow$ GPU data transfer.
- Achieving coalesced memory access.
- Exploiting shared memory.
  - Not considered here.
Fine-Grained Parallelism

- CPUs achieve high performance by reducing memory latency: accessing memory in cache.
- GPUs achieve high performance by hiding memory latency: overlap memory access with computation.

→ Need finer-grained parallelism to keep GPUs busy.
Fine-Grained Parallelism

- In the context of a CFD code, fine-grained parallelism corresponds to processing each cell, face, edge, or point in parallel.

→ If there are 1 million grid cells, then there should be 1 million threads running in parallel.

→ Domain decomposition is probably an insufficient level of parallelism.
Data Transfer

• CPUs and GPUs have separate memory spaces.
• Transfer between them is slow:
  • <10 GB/s
• Internal GPU bandwidth > 100 GB/s

→ Just porting “bottleneck subroutines” will often eliminate any potential performance gain.
→ All parallel loops should run on the GPU
→ Transfer of large arrays, ideally, should be limited to startup and shutdown.
Coalescing

• Memory bandwidth is more often than not a bottleneck.
• Coalesced memory access is typically the determining factor in comparison to cache behavior.
• For many applications performance scales with the degree to which coalescing achieved.
• Technical specifications of coalescing requirements imply that arrays are transposed.

→ Transposing arrays is crucial to avoid needlessly incurring a substantial performance penalty.
Coalescing

- Given a Fortran array $x(2,n)$ the standard layout in memory is $x(1,1)$, $x(2,1)$, $x(1,2)$, $x(2,2)$, etc…
Coalescing

• The preferred layout for meeting GPU coalescing requirements is 
  $x(1,1), x(1,2), \ldots, x(1,n), x(2,1), \ldots, x(2,n)$
Parallelization

- A CUDA kernel is generated for each OpenMP loop.
- OpenMP private items $\rightarrow$ Per-thread variables
- The array layout and indexes in kernel code are transposed.
- Reductions are performed after writing to a temporary buffer using Thrust

http://code.google.com/p/thrust [Hoberock, Bell, 2009]
This parallel loop over the points of a mesh, taken from FEFLO’s flux-corrected transport, compressible flow module, is straightforward, yet tedious to translate.

It already exhibits fine-grained parallelism so a direct translation is sufficient.
__global__
void locfct_loop2(double* delun, int delun_s1, double* mmatm,  
int npami, int npamx)
{

double cmmat;
const unsigned int ip = blockDim.x*blockIdx.x+threadIdx.x+npami;
if(ip > npamx) return;

cmmat=mmatm[ip-1];
delun[ip-1+delun_s1*(1-1)]=cmmat*delun[ip-1+delun_s1*(1-1)];
delun[ip-1+delun_s1*(2-1)]=cmmat*delun[ip-1+delun_s1*(2-1)];
delun[ip-1+delun_s1*(3-1)]=cmmat*delun[ip-1+delun_s1*(3-1)];
delun[ip-1+delun_s1*(4-1)]=cmmat*delun[ip-1+delun_s1*(4-1)];
delun[ip-1+delun_s1*(5-1)]=cmmat*delun[ip-1+delun_s1*(5-1)];
}

• This CUDA kernel is a direct translation of the original OpenMP loop.
• The indexes are transposed to ensure coalescing.
• Array indexes are decremented by 1 to use 0-based indexing.
• The required per-thread variables \texttt{ip}, \texttt{cmmat} were detected from the OpenMP directive and locally declared.
• The required arrays \texttt{delun} and \texttt{mmatm} and parameters \texttt{npami}, and \texttt{npamx} are automatically detected and passed in.
extern "C"

void locfct_loop2_(da_double2* delun, da_double1* mmatm,
                 int* npami, int* npamx)
{
    dim3 dimGrid=dim3(round_up(*npamx)-(*npami)+1),1,1);
    dim3 dimBlock=dim3(256,1,1);
    locfct_loop2<<<dimGrid,dimBlock>>>(
        (delun->a,delun->shape[1],mmatm->a,*npami,*npamx);
}

call locfct_loop2(delun,mmatm,npami,npamx)

• This kernel wrapper function invokes the CUDA kernel.

• A call to this wrapper function replaces the original parallel loop in the Fortran code.

• delun and mmatm are now GPU arrays and array shape and offset information is tracked using a simple C-struct/Fortran-derived type.
Parallelization

• The previous example already exhibited fine-grained parallelism and was directly converted.
• All point loops in FEFLO are treated this way.
• The edge loops in FEFLO are parallelized with OpenMP but only in a coarse-grained way.

→ Requires restructuring the loops, manually or automatically, to expose fine-grained parallelism.
Parallelization

- Due to FEFLO’s **uniform coding conventions**, **automatic** restructuring was possible for edge loops, requiring an additional ~200 lines of FEFLO-specific conversion code.
- This typically involved parallelizing inner loop(s), indicated by
  - Not containing any sub-loops.
  - Vectorization directives.
  - Certain loop variable names
- It is conceivable a similar approach could be applied to other codes.
Tracking Arrays

- Uses a transposed GPU layout for coalescing requirements
- Determines memory space placement (GPU or CPU).
- Enforces consistent placement to avoid expensive data transfer.
- Handles memory transfer when explicitly requested.
- Handles different sub-array semantics depending on the context.
- Placement of arrays in constant memory.
- And more...
Array Placement

- CPUs and GPUs have separate memory spaces, memory transfer is slow and avoided.

**Criterion:** An array used in a single parallel loop is designated as a GPU array throughout the entire code.

→ The converter strictly enforces this and reports any inconsistent usage as errors.
Array Transfer

• Some CPU $\leftrightarrow$ GPU transfer is necessary:
• Serial Code
  • Certain portions of the code (e.g., mesh generation) are intentionally left as serial, CPU code, and *not* converted.
  • Also needed for incremental GPU porting.
  • Calls made to these subroutines are automatically wrapped with data transfer and transposition calls.
• Input/Output
• Results of reduction loops
• When explicitly requested via custom directives.
Sub-arrays

- In Fortran, a particular memory layout is relied upon when passing an array to another subroutine expecting a sub-array or an array with a different shape.

Dilemma: *Is a logically offset, non-contiguous sub-array intended* OR *is a contiguous sub-array intended?*

→ Due to the transposed, coalesced GPU array layout, the two cases are *NOT* always equivalent and can lead to subtle bugs if the wrong approach is taken.
Sub-arrays: Case 1

subroutine rfilfmc(m,n,rma) 
implicit real*8 (a-h,o-z) 
real*8 rma(m,n) 
... 

program main 
real*8 x(3,100) 
call rfilfmc(3,95,x(1,5)) 
end

• In Fortran 77 sub-arrays may be passed to other subroutines with an offset index

• In this example a 3x95 subarray of a 3x100 array is being passed to a subroutine, starting at index (1,5).
Sub-arrays: Case 1

subroutine rfifmc(m,n,rma)
  implicit real*8 (a-h,o-z)
  real*8 rma(m,n)
...

program main
  real*8 x(3,100)
  call rfifmc(3,95,x(1,5))
end

A logical, non-contiguous offset is only meaningful if all but the last dimensions of the array and sub-array are equal.

And the offset is only made in the last dimension.
subroutine rfilvc(n,rva)
implicit real*8 (a-h,o-z)
real*8 rva(n)
...

program main
real*8 x(2,100)
call rfilvc(200,x)
end

• In Fortran 77, sub-arrays are allowed to be passed to other subroutines with a different shape

• In this the example a 2x100 array is being passed as a contiguous 1D array of length 200 to a subroutine.
Sub-arrays: Case 2

• A contiguous offset is meaningful if a non-contiguous logical offset has not already been performed.

To avoid obscure bugs this behavior is only invoked when a sub-array is explicitly requested and the logical offset of Case 1 is not possible.

```fortran
subroutine rfilvc(n,rva)
  implicit real*8 (a-h,o-z)
  real*8 rva(n)
end

program main
real*8 x(2,100)
call rfilvc(200,x(1,1))
end
```
Sub-arrays

- All of these issues are handled automatically by the converter.
  - Each case must be distinguished based on FEFLO-specific conventions.
  - Pointer arithmetic corresponding to multi-dimensional offsets performed.
  - Array dimensions and offsets are tracked.
  - Various conversion-time and run-time checks are performed.

- Relies upon FEFLO-specific conventions.
  - This issue would seem to hinder the efforts of a fully general Fortran GPU compiler from using a coalesced memory layout while simultaneously avoiding injecting unnecessary transposition or transfer calls.

  - A complicated but essential requirement for achieving full GPU performance.
Custom GPU Code

- Automatic translation in this case produced the same code that would have resulted from a manual translation, without the bugs.

- Any loops/subroutines which not can be handled automatically can be overridden with custom implementations.

- In the case of FEFLO, the cases that arose were general-purpose, well-studied algorithms, with implementations provided by Thrust.
Summary of Performance Issues

• Fine-Grained Parallelism
  • Point loops are translated directly.
  • Edge loops are restructured automatically to expose fine-grained parallelism.
  • Difficult data-parallel algorithms are overridden with custom implementations based on Thrust

• Avoiding GPU $\leftrightarrow$ CPU data transfer
  • Arrays are restricted to one memory space.
  • Memory transfer is only performed when explicitly requested.

• Coalesced Memory Access
  • Arrays use a transposed layout, throughout the entire code.
  • Numbering schemes tailored to meet coalescing requirements are an open problem and have the potential to drastically improve performance.
Multiple Output Targets

- Completely rewriting FEFLO the next time a new architecture comes out is not a good option.
- OpenCL is not a completely satisfactory solution to this issue. → Portable code, not necessarily portable performance.
- The converter has varying degrees of support for outputting to:
  - CUDA
  - PGI CUDA Fortran
  - OpenCL
- Targets can be added very rapidly.
MPI Integration

- CUDA = Fine-grained parallelism
  - Granularity of individual mesh points, edges, elements, etc.
- MPI = Coarse-grained parallelism
  - Decomposes meshes into sub-domains based on partitioning.

→ Complementary forms of parallelism.

→ Use existing MPI code to achieve multi-GPU parallelization.

  - The MPI wrapper subroutines are not processed by the converter, and the converter automatically places appropriate data transfer calls.
Manual Effort Required

- Exposing fine-grained parallelism sufficient for running on GPUs.
- Ensuring consistent array placement
  - Any errors regarding inconsistencies in array usage reported by the converter must be resolved.
- Removing assumptions regarding memory layout
  - Certain sub-array tricks must be prohibited or only interpreted based on conventions being followed.
Limitations

• Requires code to already express fine-grained parallelism.
  • Conversion of arbitrary, serial Fortran code is not attempted.
• Requires code to primarily use data on the GPU or CPU, not both.
• Shared memory management code is not generated.
  • Not relevant to FEFLO, but important for other codes.
• Only the subset of Fortran needed by FEFLO supported.
  • Support could be broadened as needed.
• C/C++ not supported
Results

• Many solver options are ported.
  • All parallel loops are automatically converted to GPU code.
  • No large data transfer during time-stepping.
• Compressible Cases:
  • Shock Tube
  • Blast
  • NACA 0012 Air Foil
• Incompressible Cases:
  • Pipe Flow
  • Dam Break with a free surface.
Shock Tube:
GPU and CPU comparison
Blast in a Room

- Compressible Euler
- Ideal Gas Equation of State
- Flux-Corrected Transport
- 1 million elements
- 60 Time Steps
- Double Precision

<table>
<thead>
<tr>
<th>CPU/GPU</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7 940 (1)</td>
<td>35</td>
</tr>
<tr>
<td>Core i7 940 (2)</td>
<td>32</td>
</tr>
<tr>
<td>Core i7 940 (4)</td>
<td>18</td>
</tr>
<tr>
<td>Core i7 940 (8)</td>
<td>17</td>
</tr>
<tr>
<td>GTX 285</td>
<td>10</td>
</tr>
</tbody>
</table>
NACA 0012 Air Foil

- Steady State Compressible Euler
- Ideal Gas Equation of State
- HLLC Riemann Solver
- 1 million elements
- 100 Time Steps
- Double Precision

<table>
<thead>
<tr>
<th>CPU/GPU</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7 940 (1)</td>
<td>184</td>
</tr>
<tr>
<td>Core i7 940 (2)</td>
<td>104</td>
</tr>
<tr>
<td>Core i7 940 (4)</td>
<td>60</td>
</tr>
<tr>
<td>Core i7 940 (8)</td>
<td>52</td>
</tr>
<tr>
<td>GTX 285</td>
<td>32</td>
</tr>
</tbody>
</table>
Pipe

- Steady-State Incompressible Navier-Stokes + Heat Transfer
- Advection: Roe solver
- Pressure: Poisson (Projection), DPCG(Scalar Products)
- 0.6 million elements
- 100 Time Steps
- Double Precision

<table>
<thead>
<tr>
<th>CPU/GPU</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7 940 (1)</td>
<td>300</td>
</tr>
<tr>
<td>Core i7 940 (2)</td>
<td>179</td>
</tr>
<tr>
<td>Core i7 940 (4)</td>
<td>126</td>
</tr>
<tr>
<td>Core i7 940 (8)</td>
<td>121</td>
</tr>
<tr>
<td>GTX 285</td>
<td>115</td>
</tr>
</tbody>
</table>
Dam Break

- Transient Incompressible Navier-Stokes
- VOF for Free Surface
- 0.7 million elements
- 100 Time Steps
- Double Precision

<table>
<thead>
<tr>
<th>CPU/GPU</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7 940 (1)</td>
<td>93</td>
</tr>
<tr>
<td>Core i7 940 (2)</td>
<td>58</td>
</tr>
<tr>
<td>Core i7 940 (4)</td>
<td>42</td>
</tr>
<tr>
<td>Core i7 940 (8)</td>
<td>42</td>
</tr>
<tr>
<td>GTX 285</td>
<td>42</td>
</tr>
</tbody>
</table>
Conclusions

• It is possible to automatically generate running GPU code from a large-scale legacy Fortran code, which allows for continued development in single codebase.

• Sufficient fine-grained parallelism must be expressed in the original Fortran code.

• Coding conventions should be employed consistently to ease any necessary custom restructuring of the code, or to allow for assumptions to be made when tracking arrays across subroutine calls.