Applied Parallel Computing LLC
http://parallel-computing.pro

Generic approach to Legacy Fortran code porting on GPU

Dmitry Mikushin (APC LLC), Alexey Ivakhnenko (APC LLC), Anna Kuznetsova (APC LLC),
Igor Zacharov (EPFL / Eurotech), Eric McIntosh (CERN)

October 20, 2015

Abstract

In this seminar we will present our methodology of unleashing the potential of GPU computing in legacy Fortran codes: to what degree the source code has to be modified to become usable on GPUs, how to turn single-threaded Fortran code into multi-threaded ensemble, how to pack all code into single GPU kernel to minimize synchronization stalls, how to map small loops onto parallel threads with custom directives and LLVM/NVVM, how to retain the preferred Fortran formatted outputs and other questions. The seminar is intended to introduce scientific code developers into techniques beyond the standard CUDA/OpenCL/OpenACC programming methodology, conserve the existing codebase and achieve high GPU utilization.

Fortran on GPUs

- Directives
  - OpenACC
  - OpenMP 4.0
- CUDA
  - CUDA Fortran (proprietary, free academic license from 2015 on)
  - GCC: GCC Fortran frontend ⇒ DragonEgg ⇒ LLVM IR ⇒ PTX (NVIDIA) ⇒ AMDIL (AMD)
  - Open64: GCC Fortran frontend ⇒ WHIRL ⇒ PTX (NVIDIA)

Examples

OpenACC (Fortran)

```
!$acc kernels loop independent
do i = 1, n
  y(i) = y(i) + a * x(i)
enddo
```

OpenMP 4.0 (C)

```
#pragma omp target data map(to:x[0:n]) map(tofrom:y[0:n])
{
    #pragma omp target
    #pragma omp for
    for (int i = 0; i < n; i++)
        y[i] += a * x[i];
}
```

CUDA Fortran

```
attributes(global) subroutine axpy(a, x, y, n)
implicit none

double precision, value :: a
double precision, dimension(:), device :: x, y
integer, value :: n
integer :: i

i = (blockIdx%x - 1) * blockDim%x + threadIdx%x
if (i <= n) then
  y(i) = y(i) + a * x(i)
endif

end subroutine axpy
...
call axpy<<<nblocks, nthreads>>>(
  a, x_dev, y_dev, n)
```

Examples

OpenACC (Fortran)

```fortran
!$acc kernels loop independent
do i = 1, n
  y(i) = y(i) + a * x(i)
enddo
```

OpenMP 4.0 (C)

```c
#pragma omp target data map(to:x[0:n]) map(tofrom:y[0:n])
{
    #pragma omp target
    #pragma omp for
    for (int i = 0; i < n; i++)
        y[i] += a * x[i];
}
```

CUDA Fortran

```fortran
attributes(global) subroutine axpy(a, x, y, n)
implicit none

double precision, value :: a
double precision, dimension(:), device :: x, y
integer, value :: n
integer :: i

i = (blockIdx%x - 1) * blockDim%x + threadIdx%x
if (i <= n) then
  y(i) = y(i) + a * x(i)
endif

end subroutine axpy
...
call axpy<<<nblocks, nthreads>>>(
  a, x_dev, y_dev, n)
```

Examples

OpenACC (Fortran)

```fortran
!$acc kernels loop independent
do i = 1, n
  y(i) = y(i) + a * x(i)
enddo
```

OpenMP 4.0 (C)

```c
#pragma omp target data map(to:x[0:n]) map(tofrom:y[0:n])
{
    #pragma omp target
    #pragma omp for
    for (int i = 0; i < n; i++)
        y[i] += a * x[i];
}
```

CUDA Fortran

```fortran
attributes(global) subroutine axpy(a, x, y, n)
implicit none

double precision, value :: a
double precision, dimension(:), device :: x, y
integer, value :: n
integer :: i

i = (blockIdx%x - 1) * blockDim%x + threadIdx%x
if (i <= n) then
  y(i) = y(i) + a * x(i)
endif

end subroutine axpy
...
call axpy<<<nblocks, nthreads>>>(
  a, x_dev, y_dev, n)
```

Open-source LLVM-based OpenMP 4.0 compiler for NVIDIA CUDA GPUs

Part I:

```
$ mkdir -p $HOME/forge/openmp4
$ cd $HOME/forge/openmp4

$ git clone https://github.com/clang-omp/llvm_trunk llvm
$ git clone https://github.com/clang-omp/compiler-rt_trunk llvm/projects/compiler-rt
$ git clone https://github.com/clang-omp/clang_trunk llvm/tools/clang

$ cd llvm/
$ mkdir build
$ cd build/
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/forge/openmp4/llvm/install ..
$ make -j12
$ make install
$ export PATH=$PATH:$HOME/forge/openmp4/llvm/build/bin/
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/forge/openmp4/llvm/install/lib

$ cd $HOME/forge/openmp4
$ git clone http://llvm.org/git/openmp.git
$ cd openmp/runtime/
```

Open-source LLVM-based OpenMP 4.0 compiler for NVIDIA CUDA GPUs

Part II:

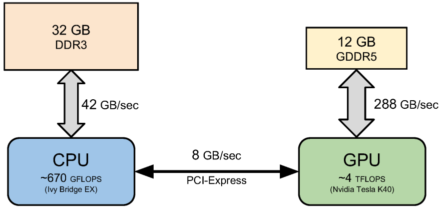
```
$ mkdir build
$ cd build/
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/forge/openmp4/llvm/install ..
$ make -j12
$ make install

$ cd $HOME/forge/openmp4
$ git clone https://github.com/clang-omp/libomptarget.git
$ cd libomptarget
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/forge/openmp4/llvm/install -DOMPTARGET_NVPTX_SM=30,35 ..
$ make -j12
$ cp -rf lib/libomptarget* $HOME/forge/openmp4/llvm/install/lib/
```

Now we can compile *example.c* with OpenMP 4.0 directives using the following command:

```
LIBRARY_PATH=$(shell dirname $(shell which clang-3.8))/../lib clang-3.8 -fopenmp -omptargets=↩
     nvptx64sm_30-nvidia-linux -g -O3 -std=c99 $< -o $@
```

Realistic cases are actually much more complex

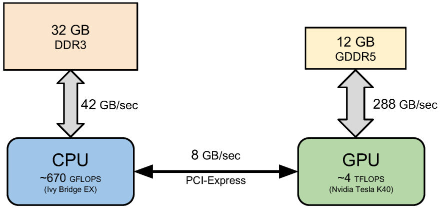
1. CPU↔GPU data transfers are very expensive ⇒ variables have to be coordinated to persist in GPU memory as much as possible ⇒ more code has to be ported onto GPU to avoid data transfers



OpenACC approach: define persistent data region and share the GPU data across all nested kernels

Realistic cases are actually much more complex

1. CPU$\leftrightarrow$GPU data transfers are very expensive $\Rightarrow$ variables have to be coordinated to persist in GPU memory as much as possible $\Rightarrow$ more code has to be ported onto GPU to avoid data transfers



```
#pragma acc data create (w0[0:szarray], w1[0:szarray])
{
    ...
    #pragma acc kernels present(w0[0:szarray], w1[0:szarray])
    ...
}
#pragma acc update device(w0[0:szarray], w1[0:szarray])
```

OpenACC approach: define persistent data region and share the GPU data across all nested kernels

Realistic cases are actually much more complex

2 Parallel workflow must be coordinated across multiple source files

- Nested routines must be aware they are called from within the GPU kernel:

- Older (simpler) OpenACC compilers do not inline very well

Realistic cases are actually much more complex

2 Parallel workflow must be coordinated across multiple source files

- Nested routines must be aware they are called from within the GPU kernel:

sincos.f90

```
!$acc kernels loop
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sincos_ijk(x(i, j, k), y(i, j, k))
    enddo
  enddo
enddo
```

function.f90

```
function sincos_ijk(x, y)

  implicit none
  real, intent(in) :: x, y
  real :: sincos_ijk

  sincos_ijk = sin(x) + cos(y)

end function sincos_ijk
```

- Older (simpler) OpenACC compilers do not inline very well

Realistic cases are actually much more complex

2. Parallel workflow must be coordinated across multiple source files
   - Nested routines must be aware they are called from within the GPU kernel:

sincos.f90

```
!$acc kernels loop
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sincos_ijk(x(i, j, k), y(i, j, k))
    enddo
  enddo
enddo
```

function.f90

```
function sincos_ijk(x, y)

  implicit none
  real, intent(in) :: x, y
  real :: sincos_ijk

  sincos_ijk = sin(x) + cos(y)

end function sincos_ijk
```

   - Older (simpler) OpenACC compilers do not inline very well

Realistic cases are actually much more complex

3. Order of loops has often to be changed:

```
!$acc kernels
!$acc loop collapse(2)
do j = j_start, j_end
  do i = i_start, i_end
    do k = kts,ktf
      mrdx=msfux(i,j)*rdx
      tendency(i,k,j)=tendency(i,k,j)-mrdx*0.25 &
        *((ru(i+1,k,j)+ru(i,k,j))*(u(i+1,k,j)+u(i,k,j)) &
        -(ru(i,k,j)+ru(i-1,k,j))*(u(i,k,j)+u(i-1,k,j)))
    enddo
  enddo
enddo
```

The j-i-k layout above is the most efficient arrangement for GPU execution:

- Iterations of outer loops are mapped to GPU threads
- Adjacent threads in Warp are used to process consecutive elements in same column
- Sequential inner loop processes consecutive columns with coordinated threads

Realistic cases are actually much more complex

3. Order of loops has often to be changed:

```
!$acc kernels
!$acc loop collapse(2)
do j = j_start, j_end
  do i = i_start, i_end
    do k = kts,ktf
      mrdx=msfux(i,j)*rdx
      tendency(i,k,j)=tendency(i,k,j)-mrdx*0.25 &
        *((ru(i+1,k,j)+ru(i,k,j))*(u(i+1,k,j)+u(i,k,j)) &
        -(ru(i,k,j)+ru(i-1,k,j))*(u(i,k,j)+u(i-1,k,j)))
    enddo
  enddo
enddo
```

The j-i-k layout above is the most efficient arrangement for GPU execution:

- Iterations of outer loops are mapped to GPU threads
- Adjacent threads in Warp are used to process consecutive elements in same column
- Sequential inner loop processes consecutive columns with coordinated threads

Realistic cases are actually much more complex

3 Order of loops has often to be changed:

```
!$acc kernels
!$acc loop collapse(2)
do j = j_start, j_end
  do i = i_start, i_end
    do k = kts,ktf
      mrdx=msfux(i,j)*rdx
      tendency(i,k,j)=tendency(i,k,j)-mrdx*0.25 &
        *((ru(i+1,k,j)+ru(i,k,j))*(u(i+1,k,j)+u(i,k,j)) &
        -(ru(i,k,j)+ru(i-1,k,j))*(u(i,k,j)+u(i-1,k,j)))
    enddo
  enddo
enddo
```

The j-i-k layout above is the most efficient arrangement for GPU execution:

- Iterations of outer loops are mapped to GPU threads
- Adjacent threads in Warp are used to process consecutive elements in same column
- Sequential inner loop processes consecutive columns with coordinated threads

Realistic cases are actually much more complex

3 Order of loops has often to be changed:

```
!$acc kernels
!$acc loop collapse(2)
do j = j_start, j_end
  do i = i_start, i_end
    do k = kts,ktf
      mrdx=msfux(i,j)*rdx
      tendency(i,k,j)=tendency(i,k,j)-mrdx*0.25 &
        *((ru(i+1,k,j)+ru(i,k,j))*(u(i+1,k,j)+u(i,k,j)) &
        -(ru(i,k,j)+ru(i-1,k,j))*(u(i,k,j)+u(i-1,k,j)))
    enddo
  enddo
enddo
```

The j-i-k layout above is the most efficient arrangement for GPU execution:

- Iterations of outer loops are mapped to GPU threads
- Adjacent threads in Warp are used to process consecutive elements in same column
- Sequential inner loop processes consecutive columns with coordinated threads

Realistic cases are actually much more complex

3. Order of loops has often to be changed:

```fortran
!$acc kernels
!$acc loop collapse(2)
do j = j_start, j_end
  do i = i_start, i_end
    do k = kts,ktf
      mrdx=msfux(i,j)*rdx
      tendency(i,k,j)=tendency(i,k,j)-mrdx*0.25 &
        *((ru(i+1,k,j)+ru(i,k,j))*(u(i+1,k,j)+u(i,k,j)) &
        -(ru(i,k,j)+ru(i-1,k,j))*(u(i,k,j)+u(i-1,k,j)))
    enddo
  enddo
enddo
```

The j-i-k layout above is the most efficient arrangement for GPU execution:

- Iterations of outer loops are mapped to GPU threads
- Adjacent threads in Warp are used to process consecutive elements in same column
- Sequential inner loop processes consecutive columns with coordinated threads

Approaches taken in Fortran code porting on GPUs

- COSMO Regional NWP model, MeteoSwiss

  Source: Modern Fortran 90/2003, 195K cloc

  Method:

  Dynamics: code rewrite into C++, using library with CUDA backend (59% of runtime)

  Physics: OpenACC directives in existing Fortran code (22% of runtime)

  8+ developers, project started in 2012

  2.5$\times$ overall speedup of hybrid version

- WRF Regional NWP model, NCAR

  Source: Fortran 77/90, 440K cloc

  Method:

  Partial port of physics models in OpenACC

  2$\times$ - **4.5**$\times$ speedup on isolated selected code blocks

  3 NVIDIA engineers + contractors, project started in 2013

Approaches taken in Fortran code porting on GPUs

- COSMO Regional NWP model, MeteoSwiss

  Source: Modern Fortran 90/2003, 195K cloc

  Method:

  Dynamics: code rewrite into C++, using library with CUDA backend (59% of runtime)

  Physics: OpenACC directives in existing Fortran code (22% of runtime)

  8+ developers, project started in 2012

  2.5$\times$ overall speedup of hybrid version

- WRF Regional NWP model, NCAR

  Source: Fortran 77/90, 440K cloc

  Method:

  Partial port of physics models in OpenACC

  2$\times$ **- 4.5**$\times$ speedup on isolated selected code blocks

  3 NVIDIA engineers + contractors, project started in 2013

Our code is even more complex: Programmability

1. Dozens of common blocks

```
  common /fordes/ nda,ndamaxi
  common / da / cc(lst),eps,epsmac
  common / dai / i1(lst),i2(lst),                                    &
 &ie1(lea),ie2(lea),ieo(lea),ia1(0:lia),ia2(0:lia),ifi(lea),        &
 &idano(lda),idanv(lda),idapo(lda),idalm(lda),idall(lda),           &
 &nst,nomax,nvmax,nmmax,nocut,lfi
```

No support for common blocks in CUDA Fortran and OpenACC

$\Rightarrow$ no choice, but to rewrite all common blocks as modules!

Our code is even more complex: Programmability

1. Dozens of common blocks

```fortran
 common /fordes/ nda,ndamaxi
 common / da / cc(lst),eps,epsmac
 common / dai / i1(lst),i2(lst),                                   &
&ie1(lea),ie2(lea),ieo(lea),ia1(0:lia),ia2(0:lia),ifi(lea),        &
&idano(lda),idanv(lda),idapo(lda),idalm(lda),idall(lda),           &
&nst,nomax,nvmax,nmmax,nocut,lfi
```

No support for common blocks in CUDA Fortran and OpenACC

⇒ no choice, but to rewrite all common blocks as modules!

Our code is even more complex: Programmability

1. Dozens of common blocks

```
 common /fordes/ nda,ndamaxi
 common / da / cc(lst),eps,epsmac
 common / dai / i1(lst),i2(lst),                                   &
&ie1(lea),ie2(lea),ieo(lea),ia1(0:lia),ia2(0:lia),ifi(lea),        &
&idano(lda),idanv(lda),idapo(lda),idalm(lda),idall(lda),           &
&nst,nomax,nvmax,nmmax,nocut,lfi
```

No support for common blocks in CUDA Fortran and OpenACC

$\Rightarrow$ no choice, but to rewrite all common blocks as modules!

Our code is even more complex: Programmability

2 Interleaved compute and I/O

```
      write(*,10080) hda(1,1,jord,0),hda(0,1,jord,0)
      ...
10080 format(/,'H_1,0     = ',2x,e16.8,7x,'H_0,1     = ', 2x,e16.8)
```

Modern CUDA C++ has printf support since Compute Capability 2.0

Unlike CUDA C++, CUDA Fortran does not support formatted output

$\Rightarrow$ Provide I/O support by other means:

- Implement Fortran formatted I/O in software
- GPU kernel: Aggregate the format strings and data from individual GPU threads to output buffer
- Host CPU: Receive the output buffer and print its contents according to formats

Our code is even more complex: Programmability

2 Interleaved compute and I/O

```
      write(*,10080) hda(1,1,jord,0),hda(0,1,jord,0)
      ...
10080 format(/,'H_1,0     = ',2x,e16.8,7x,'H_0,1     = ', 2x,e16.8)
```

Modern CUDA C++ has printf support since Compute Capability 2.0

Unlike CUDA C++, CUDA Fortran does not support formatted output

$\Rightarrow$ Provide I/O support by other means:

- Implement Fortran formatted I/O in software
- GPU kernel: Aggregate the format strings and data from individual GPU threads to output buffer
- Host CPU: Receive the output buffer and print its contents according to formats

Our code is even more complex: Programmability

2 Interleaved compute and I/O

```
      write(*,10080) hda(1,1,jord,0),hda(0,1,jord,0)
      ...
10080 format(/,'H_1,0     = ',2x,e16.8,7x,'H_0,1     = ', 2x,e16.8)
```

Modern CUDA C++ has printf support since Compute Capability 2.0

Unlike CUDA C++, CUDA Fortran does not support formatted output

⇒ Provide I/O support by other means:

- Implement Fortran formatted I/O in software
- GPU kernel: Aggregate the format strings and data from individual GPU threads to output buffer
- Host CPU: Receive the output buffer and print its contents according to formats

Our code is even more complex: Programmability

2 Interleaved compute and I/O

```
      write(*,10080) hda(1,1,jord,0),hda(0,1,jord,0)
      ...
10080 format(/,'H_1,0    = ',2x,e16.8,7x,'H_0,1    = ', 2x,e16.8)
```

- Modern CUDA C++ has printf support since Compute Capability 2.0
- Unlike CUDA C++, CUDA Fortran does not support formatted output
- ⇒ Provide I/O support by other means:
  - Implement Fortran formatted I/O in software
  - GPU kernel: Aggregate the format strings and data from individual GPU threads to output buffer
  - Host CPU: Receive the output buffer and print its contents according to formats

Our code is even more complex: Programmability

2. Interleaved compute and I/O

```
      write(*,10080) hda(1,1,jord,0),hda(0,1,jord,0)
      ...
10080 format(/,'H_1,0    = ',2x,e16.8,7x,'H_0,1    = ', 2x,e16.8)
```

Modern CUDA C++ has printf support since Compute Capability 2.0

Unlike CUDA C++, CUDA Fortran does not support formatted output

$\Rightarrow$ Provide I/O support by other means:

Implement Fortran formatted I/O in software

GPU kernel: Aggregate the format strings and data from individual GPU threads to output buffer

Host CPU: Receive the output buffer and print its contents according to formats

Our code is even more complex: Programmability

2 Interleaved compute and I/O

```
      write(*,10080) hda(1,1,jord,0),hda(0,1,jord,0)
      ...
10080 format(/,'H_1,0     = ',2x,e16.8,7x,'H_0,1     = ', 2x,e16.8)
```

Modern CUDA C++ has printf support since Compute Capability 2.0

Unlike CUDA C++, CUDA Fortran does not support formatted output

$\Rightarrow$ Provide I/O support by other means:

- Implement Fortran formatted I/O in software
- GPU kernel: Aggregate the format strings and data from individual GPU threads to output buffer
- Host CPU: Receive the output buffer and print its contents according to formats

Our code is even more complex: Programmability

2 Interleaved compute and I/O

```
      write(*,10080) hda(1,1,jord,0),hda(0,1,jord,0)
      ...
10080 format(/,'H_1,0     = ',2x,e16.8,7x,'H_0,1     = ', 2x,e16.8)
```

Modern CUDA C++ has printf support since Compute Capability 2.0

Unlike CUDA C++, CUDA Fortran does not support formatted output

$\Rightarrow$ Provide I/O support by other means:

- Implement Fortran formatted I/O in software
- GPU kernel: Aggregate the format strings and data from individual GPU threads to output buffer
- Host CPU: Receive the output buffer and print its contents according to formats

Our code is even more complex: Programmability

3. Compatibility of data layouts on CPU and GPU

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name=”__MOD_da_001_cc”) :: cc
      bind(C,name=”__MOD_da_002_eps”) :: eps
      bind(C,name=”__MOD_da_003_epsmac”) :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

With equal memory layouts, we can transfer experiment state between host and device with a single copy operation (upon initialization, finalization and/or e.g. for dynamic offloading of inefficient setup back to host)

Memory layout for modules could be different for data in CPU and in GPU memory

⇒ Use `ISO_C_BINDING` to enforce particular layout of data in modules

Our code is even more complex: Programmability

3 Compatibility of data layouts on CPU and GPU

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name="__MOD_da_001_cc") :: cc
      bind(C,name="__MOD_da_002_eps") :: eps
      bind(C,name="__MOD_da_003_epsmac") :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

With equal memory layouts, we can transfer experiment state between host and device with a single copy operation (upon initialization, finalization and/or e.g. for dynamic offloading of inefficient setup back to host)

Memory layout for modules could be different for data in CPU and in GPU memory

⇒ Use ISO_C_BINDING to enforce particular layout of data in modules

Our code is even more complex: Programmability

3 Compatibility of data layouts on CPU and GPU

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name="__MOD_da_001_cc") :: cc
      bind(C,name="__MOD_da_002_eps") :: eps
      bind(C,name="__MOD_da_003_epsmac") :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

With equal memory layouts, we can transfer experiment state between host and device with a single copy operation (upon initialization, finalization and/or e.g. for dynamic offloading of inefficient setup back to host)

Memory layout for modules could be different for data in CPU and in GPU memory

⇒ Use ISO_C_BINDING to enforce particular layout of data in modules

Our code is even more complex: Programmability

3 Compatibility of data layouts on CPU and GPU

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name="__MOD_da_001_cc") :: cc
      bind(C,name="__MOD_da_002_eps") :: eps
      bind(C,name="__MOD_da_003_epsmac") :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

With equal memory layouts, we can transfer experiment state between host and device with a single copy operation (upon initialization, finalization and/or e.g. for dynamic offloading of inefficient setup back to host)

Memory layout for modules could be different for data in CPU and in GPU memory

$\Rightarrow$ Use `ISO_C_BINDING` to enforce particular layout of data in modules

Our code is even more complex: Programmability

4 Thread safety

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name="__MOD_da_001_cc") :: cc
      bind(C,name="__MOD_da_002_eps") :: eps
      bind(C,name="__MOD_da_003_epsmac") :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

Unlike C++ classes, Fortran common blocks and modules exist as single instance

Ensemble simulations: individual instance of all datasets for each of parallel experiments

$\Rightarrow$ Clone space for datasets using low-level tweaks, to avoid too many changes into the code

Our code is even more complex: Programmability

4 Thread safety

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name="__MOD_da_001_cc") :: cc
      bind(C,name="__MOD_da_002_eps") :: eps
      bind(C,name="__MOD_da_003_epsmac") :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

Unlike C++ classes, Fortran common blocks and modules exist as single instance

Ensemble simulations: individual instance of all datasets for each of parallel experiments

$\Longrightarrow$ Clone space for datasets using low-level tweaks, to avoid too many changes into the code

Our code is even more complex: Programmability

4 Thread safety

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name="__MOD_da_001_cc") :: cc
      bind(C,name="__MOD_da_002_eps") :: eps
      bind(C,name="__MOD_da_003_epsmac") :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

Unlike C++ classes, Fortran common blocks and modules exist as single instance

Ensemble simulations: individual instance of all datasets for each of parallel experiments

$\Rightarrow$ Clone space for datasets using low-level tweaks, to avoid too many changes into the code

Our code is even more complex: Programmability

4 Thread safety

```
      module da
      use dabnews_consts
      implicit none
      double precision :: cc(lst),eps,epsmac
#ifndef GPU
      bind(C,name="__MOD_da_001_cc") :: cc
      bind(C,name="__MOD_da_002_eps") :: eps
      bind(C,name="__MOD_da_003_epsmac") :: epsmac
#endif
#ifdef GPU
      attributes(device) :: cc,eps,epsmac
#endif
      save
      end module da
```

Unlike C++ classes, Fortran common blocks and modules exist as single instance

Ensemble simulations: individual instance of all datasets for each of parallel experiments

$\Rightarrow$ Clone space for datasets using low-level tweaks, to avoid too many changes into the code

Our code is even more complex: Performance

1. Getting enough degree of parallelism: why to go ensemble?

   The majority of parallel loops in the SixTrack code is "for each alive particle"

   Having one thread handling a single particle, up to 2 warps (64 threads) could be utilized on GPU

   At least several hundred warps are needed to utilize available resources of high-end GPU

   $\Rightarrow$ The only way to spend GPU power efficiently is to run many experiments in parallel

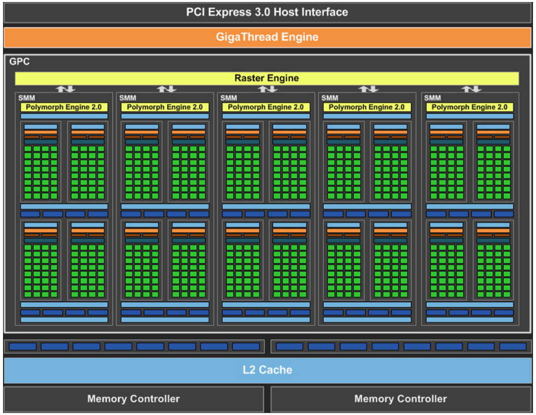


Figure: NVIDIA Maxwell architecture

Our code is even more complex: Performance

1. Getting enough degree of parallelism: why to go ensemble?
   - The majority of parallel loops in the SixTrack code is “for each alive particle”
   - Having one thread handling a single particle, up to 2 warps (64 threads) could be utilized on GPU
   - At least several hundred warps are needed to utilize available resources of high-end GPU
   - $\Rightarrow$ The only way to spend GPU power efficiently is to run many experiments in parallel

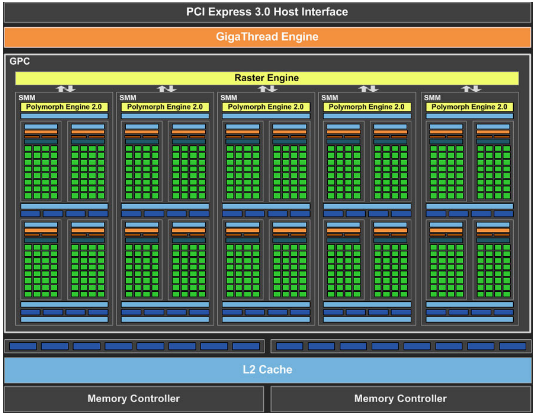


Figure: NVIDIA Maxwell architecture

Our code is even more complex: Performance

1. Getting enough degree of parallelism: why to go ensemble?

    The majority of parallel loops in the SixTrack code is “for each alive particle”

    Having one thread handling a single particle, up to 2 warps (64 threads) could be utilized on GPU

    At least several hundred warps are needed to utilize available resources of high-end GPU

    ⇒ The only way to spend GPU power efficiently is to run many experiments in parallel

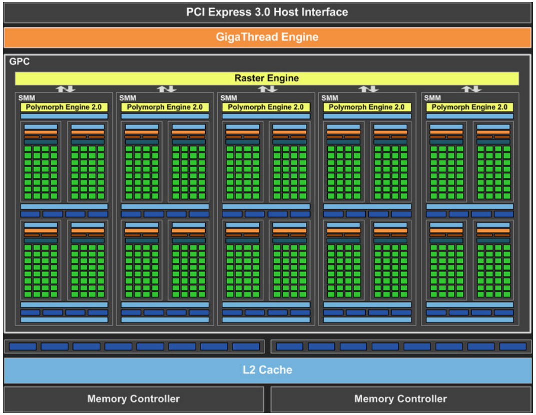


Figure: NVIDIA Maxwell architecture

Our code is even more complex: Performance

1. Getting enough degree of parallelism: why to go ensemble?

    The majority of parallel loops in the SixTrack code is "for each alive particle"
    Having one thread handling a single particle, up to 2 warps (64 threads) could be utilized on GPU
    At least several hundred warps are needed to utilize available resources of high-end GPU
    $\Rightarrow$ The only way to spend GPU power efficiently is to run many experiments in parallel

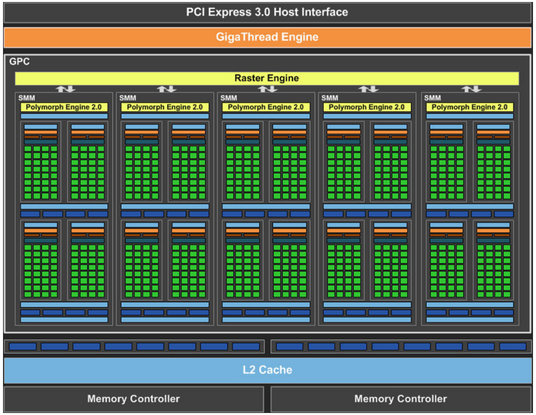


Figure: NVIDIA Maxwell architecture

Our code is even more complex: Performance

1. Getting enough degree of parallelism: why to go ensemble?

    The majority of parallel loops in the SixTrack code is “for each alive particle”

    Having one thread handling a single particle, up to 2 warps (64 threads) could be utilized on GPU

    At least several hundred warps are needed to utilize available resources of high-end GPU

    $\Rightarrow$ The only way to spend GPU power efficiently is to run many experiments in parallel

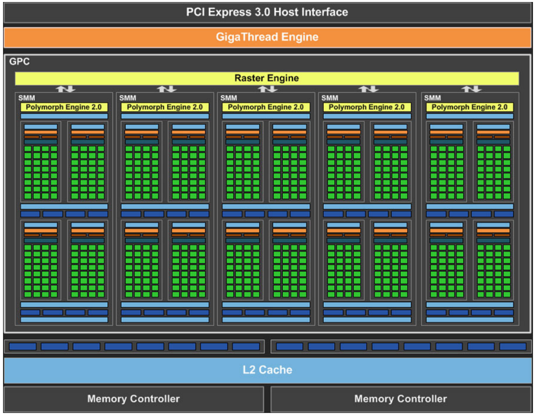
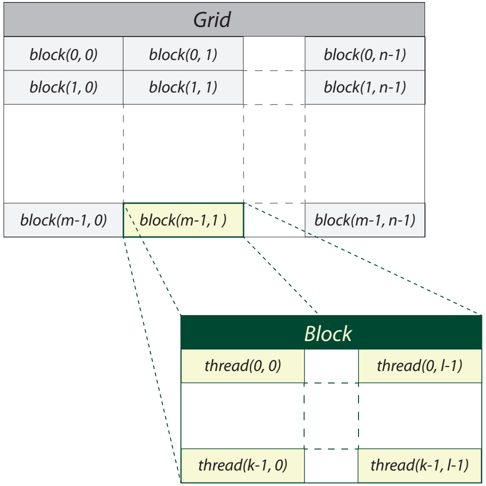


Figure: NVIDIA Maxwell architecture

Our code is even more complex: Performance

2. Getting enough degree of parallelism: how to map ensemble onto GPU compute grid?

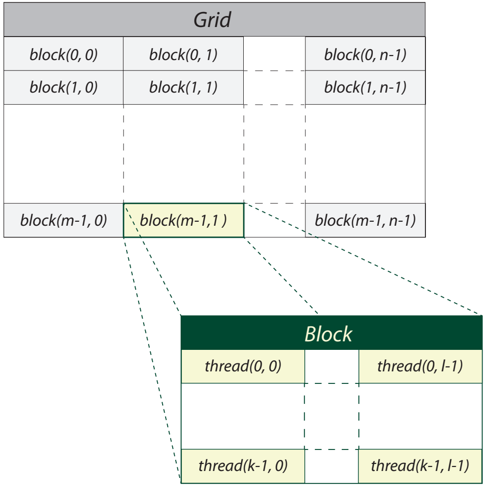
   - Single GPU kernel for each experiment Up to 32-64 simultaneous kernels (hardware limit), using CUDA streams or dynamic parallelism
     ⇒ Still not enough degree of parallelism
   - Single block of GPU kernel for each experiment, single GPU kernel shared between all experiments
     ⇒ One experiment occupies 1 block, no hardware limit, enough parallelism with significant number of experiments



Our code is even more complex: Performance

2 Getting enough degree of parallelism: how to map ensemble onto GPU compute grid?

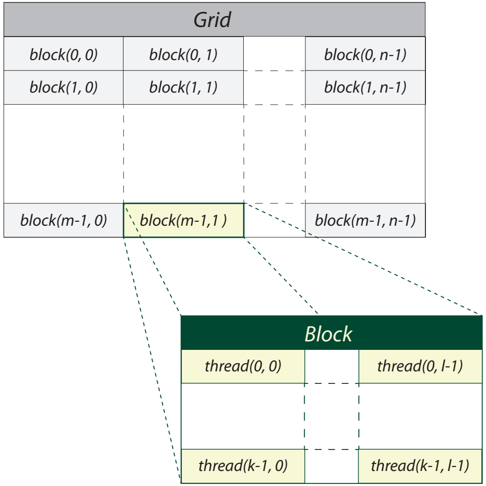
- Single GPU kernel for each experiment
  Up to 32-64 simultaneous kernels (hardware limit), using CUDA streams or dynamic parallelism
  ⇒ Still not enough degree of parallelism
- Single block of GPU kernel for each experiment, single GPU kernel shared between all experiments
  ⇒ One experiment occupies 1 block, no hardware limit, enough parallelism with significant number of experiments



Our code is even more complex: Performance

2 Getting enough degree of parallelism: how to map ensemble onto GPU compute grid?

- Single GPU kernel for each experiment
  Up to 32-64 simultaneous kernels (hardware limit), using CUDA streams or dynamic parallelism
  ⇒ Still not enough degree of parallelism
- Single block of GPU kernel for each experiment, single GPU kernel shared between all experiments
  ⇒ One experiment occupies 1 block, no hardware limit, enough parallelism with significant number of experiments



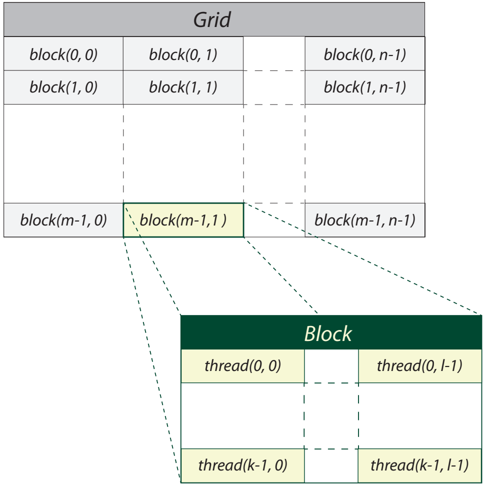
Our code is even more complex: Performance

2 Getting enough degree of parallelism: how to map ensemble onto GPU compute grid?

- Single GPU kernel for each experiment Up to 32-64 simultaneous kernels (hardware limit), using CUDA streams or dynamic parallelism

  ⇒ Still not enough degree of parallelism
- Single block of GPU kernel for each experiment, single GPU kernel shared between all experiments

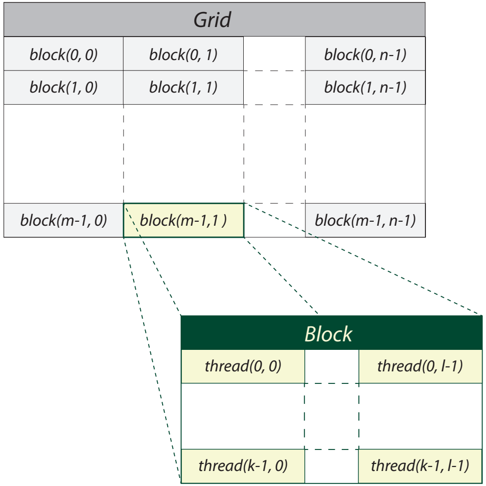
  ⇒ One experiment occupies 1 block, no hardware limit, enough parallelism with significant number of experiments



Our code is even more complex: Performance

2 Getting enough degree of parallelism:
how to map ensemble onto GPU compute grid?

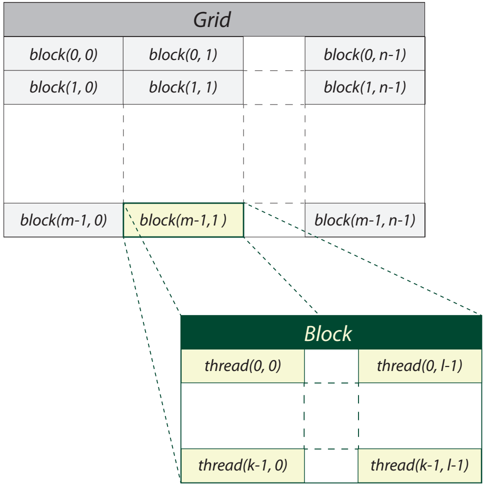
- Single GPU kernel for each experiment
  Up to 32-64 simultaneous kernels (hardware limit), using CUDA streams or dynamic parallelism
  $\Rightarrow$ Still not enough degree of parallelism
- Single block of GPU kernel for each experiment, single GPU kernel shared between all experiments
  $\Rightarrow$ One experiment occupies 1 block, no hardware limit, enough parallelism with significant number of experiments



Our code is even more complex: Performance

2. Getting enough degree of parallelism: how to map ensemble onto GPU compute grid?

   - Single GPU kernel for each experiment
     Up to 32-64 simultaneous kernels (hardware limit), using CUDA streams or dynamic parallelism
     ⇒ Still not enough degree of parallelism
   - Single block of GPU kernel for each experiment, single GPU kernel shared between all experiments
     ⇒ One experiment occupies 1 block, no hardware limit, enough parallelism with significant number of experiments



Our code is even more complex: Performance

2. Cost of kernel launches
   - Single kernel for each individual loop, the rest of compute - on host
     $\Rightarrow$ Stalls due to GPU threads synchronization after each kernel launch
     We actually do not need to synchronize experiments of ensemble with each other
   - Single kernel for the whole SixTrack code
     $\Rightarrow$ Each experiment is computed independently, without stalls on synchronization
     $\Rightarrow$ GPU kernel interleaves serial and parallel code

Our code is even more complex: Performance

2. Cost of kernel launches
   - Single kernel for each individual loop, the rest of compute - on host
     $\Rightarrow$ Stalls due to GPU threads synchronization after each kernel launch
     We actually do not need to synchronize experiments of ensemble with each other
   - Single kernel for the whole SixTrack code
     $\Rightarrow$ Each experiment is computed independently, without stalls on synchronization
     $\Rightarrow$ GPU kernel interleaves serial and parallel code

Our code is even more complex: Performance

2. Cost of kernel launches
   - Single kernel for each individual loop, the rest of compute - on host
     ⇒ Stalls due to GPU threads synchronization after each kernel launch
     We actually do not need to synchronize experiments of ensemble with each other
   - Single kernel for the whole SixTrack code
     ⇒ Each experiment is computed independently, without stalls on synchronization
     ⇒ GPU kernel interleaves serial and parallel code

Our code is even more complex: Performance

2. Cost of kernel launches
   - Single kernel for each individual loop, the rest of compute - on host
     ⇒ Stalls due to GPU threads synchronization after each kernel launch
     We actually do not need to synchronize experiments of ensemble with each other
   - Single kernel for the whole SixTrack code
     ⇒ Each experiment is computed independently, without stalls on synchronization
     ⇒ GPU kernel interleaves serial and parallel code

Our code is even more complex: Performance

2. Cost of kernel launches
   - Single kernel for each individual loop, the rest of compute - on host
     $\Rightarrow$ Stalls due to GPU threads synchronization after each kernel launch
     We actually do not need to synchronize experiments of ensemble with each other
   - Single kernel for the whole SixTrack code
     $\Rightarrow$ Each experiment is computed independently, without stalls on synchronization
     $\Rightarrow$ GPU kernel interleaves serial and parallel code

Our code is even more complex: Performance

2. Cost of kernel launches
   - Single kernel for each individual loop, the rest of compute - on host
     $\Rightarrow$ Stalls due to GPU threads synchronization after each kernel launch
     We actually do not need to synchronize experiments of ensemble with each other
   - Single kernel for the whole SixTrack code
     $\Rightarrow$ Each experiment is computed independently, without stalls on synchronization
     $\Rightarrow$ GPU kernel interleaves serial and parallel code

Our code is even more complex: Performance

2. Cost of kernel launches
   - Single kernel for each individual loop, the rest of compute - on host

     $\Rightarrow$ Stalls due to GPU threads synchronization after each kernel launch

     We actually do not need to synchronize experiments of ensemble with each other
   - Single kernel for the whole SixTrack code

     $\Rightarrow$ Each experiment is computed independently, without stalls on synchronization

     $\Rightarrow$ GPU kernel interleaves serial and parallel code

Compiler modifications to interleave serial and parallel code

```
subroutine experiment()

  call serial_code1()

  ...
  !$par do
  do i = 1, nparticles
    ...
  enddo
  !$par enddo
  ...

  call serial_code2()

end subroutine experiment
```

```
subroutine experiment()

  if (threadIdx%x .eq. 1) then
    call serial_code1()
  endif
  ...
  do i = 1, nparticles
    ...
  enddo
  ...
  if (threadIdx%x .eq. 1) then
    call serial_code2()
  endif

end subroutine experiment
```

- LLVM IR code could be intercepted from NVCC CUDA compiler backend
- LLVM IR for GPU kernel could be modified to annotate serial portions with *if (threadIdx%x .eq. 0)* condition
- Unannotated code executes in parallel for each thread of block (default CUDA behavior)

Compiler modifications to interleave serial and parallel code

```fortran
subroutine experiment()

  call serial_code1()

  ...
  !$par do
  do i = 1, nparticles
    ...
  enddo
  !$par enddo
  ...

  call serial_code2()

end subroutine experiment
```

```fortran
subroutine experiment()

  if (threadIdx%x .eq. 1) then
    call serial_code1()
  endif
  ...
  do i = 1, nparticles
    ...
  enddo
  ...
  if (threadIdx%x .eq. 1) then
    call serial_code2()
  endif

end subroutine experiment
```

- LLVM IR code could be intercepted from NVCC CUDA compiler backend
- LLVM IR for GPU kernel could be modified to annotate serial portions with *if (threadIdx%x .eq. 0)* condition
- Unannotated code executes in parallel for each thread of block (default CUDA behavior)

Compiler modifications to interleave serial and parallel code

```
subroutine experiment()

  call serial_code1()

  ...
  !$par do
  do i = 1, nparticles
    ...
  enddo
  !$par enddo
  ...

  call serial_code2()

end subroutine experiment
```

```
subroutine experiment()

  if (threadIdx%x .eq. 1) then
    call serial_code1()
  endif
  ...
  do i = 1, nparticles
    ...
  enddo
  ...
  if (threadIdx%x .eq. 1) then
    call serial_code2()
  endif

end subroutine experiment
```

- LLVM IR code could be intercepted from NVCC CUDA compiler backend
- LLVM IR for GPU kernel could be modified to annotate serial portions with *if (threadIdx%x .eq. 0)* condition
- Unannotated code executes in parallel for each thread of block (default CUDA behavior)

Compiler modifications to interleave serial and parallel code

```
subroutine experiment()

  call serial_code1()

  ...
  !$par do
  do i = 1, nparticles
    ...
  enddo
  !$par enddo
  ...

  call serial_code2()

end subroutine experiment
```

```
subroutine experiment()

  if (threadIdx%x .eq. 1) then
    call serial_code1()
  endif
  ...
  do i = 1, nparticles
    ...
  enddo
  ...
  if (threadIdx%x .eq. 1) then
    call serial_code2()
  endif

end subroutine experiment
```

- LLVM IR code could be intercepted from NVCC CUDA compiler backend
- LLVM IR for GPU kernel could be modified to annotate serial portions with *if (threadIdx%x .eq. 0)* condition
- Unannotated code executes in parallel for each thread of block (default CUDA behavior)

Conclusion

- Code has to be modernized to at least Fortran 90, in order to move to GPU
- CUDA Fortran does not support all required language features (formatted I/O, strings), but this could be workarounded
- Ensemble simulations are crucial for GPU performance
- Compiler could be customized to produce interleaved parallel and serial code guided by user-defined directives

Conclusion

- Code has to be modernized to at least Fortran 90, in order to move to GPU
- CUDA Fortran does not support all required language features (formatted I/O, strings), but this could be workarounded
- Ensemble simulations are crucial for GPU performance
- Compiler could be customized to produce interleaved parallel and serial code guided by user-defined directives

Conclusion

- Code has to be modernized to at least Fortran 90, in order to move to GPU
- CUDA Fortran does not support all required language features (formatted I/O, strings), but this could be workarounded
- Ensemble simulations are crucial for GPU performance
- Compiler could be customized to produce interleaved parallel and serial code guided by user-defined directives

Conclusion

- Code has to be modernized to at least Fortran 90, in order to move to GPU
- CUDA Fortran does not support all required language features (formatted I/O, strings), but this could be workarounded
- Ensemble simulations are crucial for GPU performance
- Compiler could be customized to produce interleaved parallel and serial code guided by user-defined directives

Conclusion

- Code has to be modernized to at least Fortran 90, in order to move to GPU
- CUDA Fortran does not support all required language features (formatted I/O, strings), but this could be workarounded
- Ensemble simulations are crucial for GPU performance
- Compiler could be customized to produce interleaved parallel and serial code guided by user-defined directives

References

- Open-source LLVM-based OpenMP 4.0 compiler for NVIDIA CUDA GPUs
- Progress Porting WRF to GPU using OpenACC
- O. Fuhrer. Getting COSMO ready for Piz Daint
- Enabling on-the-fly manipulations with LLVM IR code of CUDA sources