CUDA Fortran for Material Science

Everett Phillips, ***Massimiliano Fatica***, Josh Romero, Gregory Ruetsch

Nvidia Corporation

Filippo Spiga

QE Foundation/ University of Cambridge

Trieste 2/27-3/1

OUTLINE

- Motivation
- Porting strategy

Motivation

- GPUs are very attractive in High Performance Computing:
  - Massive multithreaded many-core chips
  - High flops count (both SP and DP): ~5TF in DP for P100
  - High memory bandwidth, ECC: ~500 GB/s on P100
  - Programming languages: CUDA C, **CUDA Fortran**, OpenACC, Python, OpenCL, MATLAB
  - Tools: libraries (BLAS, FFT, LAPACK, RNG),

    debuggers (Totalview, DDT) , profilers (Nvidia Visual Profiler)

Motivation

- Several efforts (in the past and ongoing) to port material science codes to GPU (QE, VASP, Gaussian,..) using different approaches (CUDA C, OpenACC)
- In this workshop we will use Quantum Espresso to show how to port Fortran code to the GPU
- We will describe a new implementation of PW, all done in CUDA Fortran:

  Modify source as little as possible

  Single source code for CPU and GPU versions

This is going to be very hand on/interactive: few slides, a lot of example code!!!

PORTING STRATEGY

Porting Strategy

Since the code is in Fortran 90, natural choices are CUDA Fortran or OpenACC

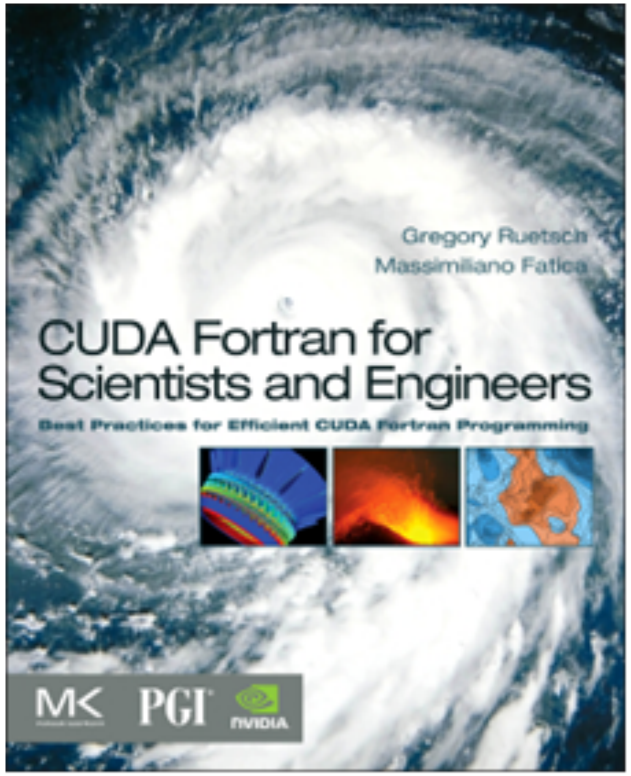
Choice of CUDA Fortran motivated by:

- Personal preference
- Use of CUF kernels made effort comparable to OpenACC
- Explicit data movement is important to optimize CPU/GPU data transfers and network traffic
- Easier to work around compiler/library bugs
- ExplicitCUDA Fortran kernels when/if needed
- Nice interface for most of the CUDA libraries

CUDA Fortran

- CUDA is a scalable model for parallel computing

- CUDA Fortran is the Fortran analog to CUDA C
  - Program has host and device code similar to CUDA C
  - Host code is based on the runtime API
  - Fortran language extensions to simplify data management
- CUDA Fortran implemented in the PGI compiler
- **Free Community edition (starting with 16.10)**


Gregory Ruetsch
Massimiliano Fatica
CUDA Fortran for Scientists and Engineers
Best Practices for Efficient CUDA Fortran Programming
MK
PGI
NVIDIA

Kernel Loop Directives (CUF Kernels)

Automatic kernel generation and invocation of host code region (arrays used in loops must reside on GPU)

```
program incTest
  use cudafor
  implicit none
  integer, parameter :: n = 256
  integer :: a(n), b
  integer, device :: a_d(n)

  a = 1; b = 3; a_d = a

  !$cuf kernel do <<<*,*>>>
  do i = 1, n
     a_d(i) = a_d(i)+b
  enddo

  a = a_d
  if (all(a == 4)) write(*,*) 'Test Passed'
end program incTest
```

Kernel Loop Directives (CUF Kernels)

- **Multidimensional arrays**

```
!$cuf kernel do(2) <<< *,* >>>
do j = 1, ny
  do i = 1, nx
    a_d(i,j) = b_d(i,j) + c_d(i,j)
  enddo
enddo
```

- **Can specify parts of execution parameter**

```
!$cuf kernel do(2) <<<(*,*),(32,4)>>>
```

- **Compiler recognizes use of scalar reduction and generates one result**

```
rsum = 0.0
!$cuf kernel do <<<*,*>>>
do i = 1, nx
  rsum = rsum + a_d(i)
enddo
```

Details

- F2003 sourced allocation:

  allocate(array_b, source=array_a)

  - Allocates *array_b* with the same bounds of *array_a*
  - Initializes *array_b* with values of *array_a*
  - If *array_b* is defined with the *device* attribute, allocation will be on the GPU and host-to-device data transfer occurs

- Variables renaming from modules:

```
#ifdef USE_CUDA
  use cudafor
  use local_arrays, only: vx => vx_d, vy => vy_d, vz => vz_d
#else
  use local_arrays, only: vx,vy,vz
#endif
```

- Use attribute(device):

```
 subroutine ExplicitTermsVX(qcap)
   implicit none
   real(fp_kind), dimension(1:nx,xstart(2):xend(2),xstart(3):xend(3)),intent(OUT)  :: qcap
#ifdef USE_CUDA
    attributes(device) :: vx,vy,vz,temp,qcap,udx3c
 #endif
```

- Use of generic interfaces:

```
Interface updateQuantity
        module procedure updateQuantity_cpu
        module procedure updateQuantity_gpu
end interface updateQuantity
```

Code Example

```fortran
subroutine CalcMaxCFL(cflm)




 use param, only: fp_kind, nxm, dy, dz, udx3m
 use local_arrays, only: vx,vy,vz

 use decomp_2d
 use mpih
 implicit none
 realintent(out)    :: cflm
 integer :: j,k,jp,kp,i,ip
 real :: qcf

 cflm=0.00000001d0




 !$OMP  PARALLEL DO &
 !$OMP   DEFAULT(none) &
 !$OMP   SHARED(xstart,xend,nxm,vz,vy,vx) &
 !$OMP   SHARED(dz,dy,udx3m) &
 !$OMP   PRIVATE(i,j,k,ip,jp,kp,qcf) &
 !$OMP   REDUCTION(max:cflm)

 do i=xstart(3),xend(3)
    ip=i+1
    do j=xstart(2),xend(2)
       jp=j+1
       do k=1,nxm
          kp=k+1
          qcf=( abs((vz(k,j,i)+vz(k,j,ip))*0.5d0*dz) &
                 +abs((vy(k,j,i)+vy(k,jp,i))*0.5d0*dy) &
                 +abs((vx(k,j,i)+vx(kp,j,i))*0.5d0*udx3m(k)))

          cflm = max(cflm,qcf)
       enddo
    enddo
 enddo

 !$OMP END PARALLEL DO

 call MpiAllMaxRealScalar(cflm)

 return
end
```

```fortran
subroutine CalcMaxCFL(cflm)

#ifdef USE_CUDA
 use cudafor
 use param, only: fp_kind, nxm, dy => dy_d, dz => dz_d, udx3m => udx3m_d
 use local_arrays, only: vx => vx_d, vy => vy_d, vz => vz_d
#else
 use param, only: fp_kind, nxm, dy, dz, udx3m
 use local_arrays, only: vx,vy,vz
#endif
 use decomp_2d
 use mpih
 implicit none
 real(fp_kind),intent(out)    :: cflm
 integer :: j,k,jp,kp,i,ip
 real(fp_kind) :: qcf

 cflm=real(0.00000001,fp_kind)

#ifdef USE_CUDA
 !$cuf kernel do(3) <<<*,*>>>
#endif
 !$OMP  PARALLEL DO &
 !$OMP   DEFAULT(none) &
 !$OMP   SHARED(xstart,xend,nxm,vz,vy,vx) &
 !$OMP   SHARED(dz,dy,udx3m) &
 !$OMP   PRIVATE(i,j,k,ip,jp,kp,qcf) &
 !$OMP   REDUCTION(max:cflm)

 do i=xstart(3),xend(3)
    ip=i+1
    do j=xstart(2),xend(2)
       jp=j+1
       do k=1,nxm
          kp=k+1
          qcf=( abs((vz(k,j,i)+vz(k,j,ip))*real(0.5,fp_kind)*dz) &
                 +abs((vy(k,j,i)+vy(k,jp,i))*real(0.5,fp_kind)*dy) &
                 +abs((vx(k,j,i)+vx(kp,j,i))*real(0.5,fp_kind)*udx3m(k)))

          cflm = max(cflm,qcf)
       enddo
    enddo
 enddo

!$OMP END PARALLEL DO

 call MpiAllMaxRealScalar(cflm)

 return
end
```

Profiling

Profiling is very important to understand bottlenecks and to spot opportunities for better interaction between the CPU and the GPU

For GPU codes, profiling information can be generated with Nvprof and visualized with Nvvp

For CPU+GPU codes, it is possible to annotate the profiling timelines using the NVIDIA Tools Extension (NVTX) library

NVTX from Fortran and CUDA Fortran:

https://devblogs.nvidia.com/parallelforall/customize-cuda-fortran-profiling-nvtx/

TIMING

Use CPU timers:

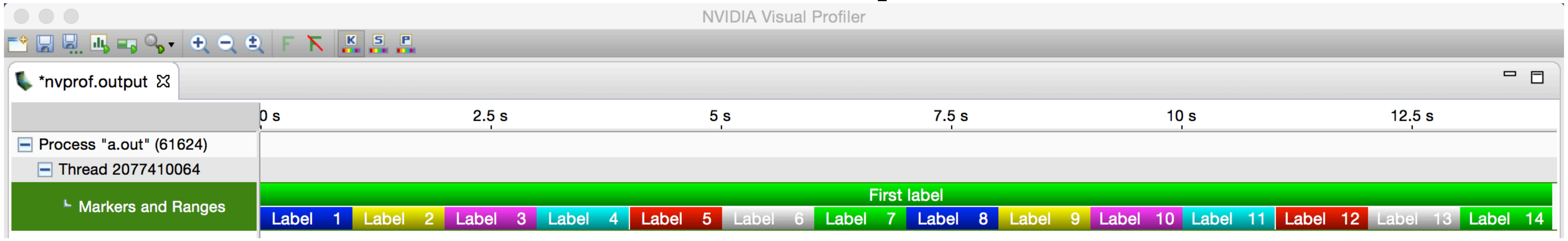
remember to add explicit synchronization barrier cudaDeviceSynchronize()
without the barrier, the timer will measure the kernel launch time not the kernel execution time

Use CUDA events:

light-weight alternatives to CPU timers via the CUDA event API

call to create events, destroy events, record events and compute the elapsed time in milliseconds between two recorded events.

NVTX Example



```
program main
  use nvtx
  character(len=4) :: itcount

  ! First range with standard color
  call nvtxStartRange("First label”)

  do n=1,14
    ! Create custom label for each marker
    write(itcount,'(i4)') n

    ! Range with custom  color
    call nvtxStartRange("Label "//itcount,n)

    ! Add sleep to make markers big
    call sleep(1)

    call nvtxEndRange
  end do

  call nvtxEndRange
end program main
```

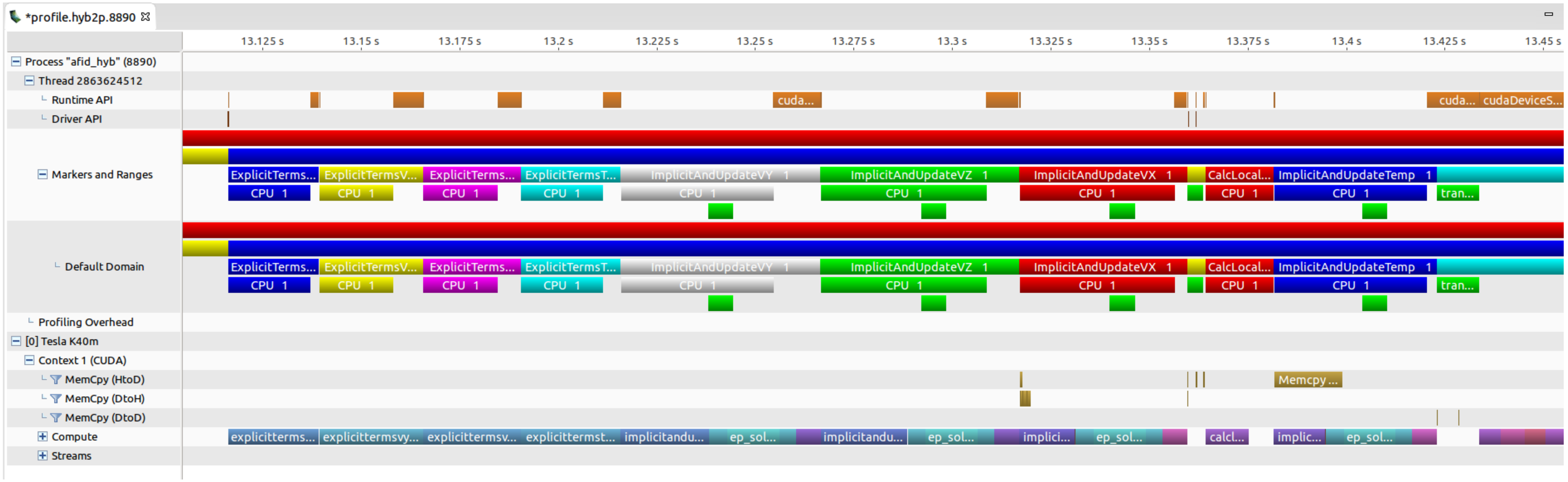
```
$ pgf90 nvtx.cuf -L/usr/local/cuda/lib -lnvToolsExt

$ nvprof -o profiler.output ./a.out

 NVPROF is profiling process 10653, command: ./a.out

 Generated result file: /Users/mfatica/profiler.output
```

NVVP Example



Profiler output for the hybrid version