ACCELERATING HPC APPLICATIONS ON ARM, ROME AND NVIDIA GPUS WITH PGI COMPILERS

Annemarie Southwell, GPU Compiler Engineering Manager
November 19, 2019
PGI FORTRAN, C AND C++
HPC Compilers for the Tesla Platform

Full C++17 | Fortran 2003/2008
Optimizing, SIMD vectorizing, OpenMP

Accelerated Computing Features
CUDA Fortran, OpenACC directives

Multi-platform solution
x86-64, OpenPOWER CPUs
NVIDIA Tesla GPUs
Linux, Windows, MacOS
PGI FORTRAN, C AND C++
HPC Compilers for the Tesla Platform

Full C++17 | Fortran 2003/2008
Optimizing, SIMD vectorizing, OpenMP

Accelerated Computing Features
CUDA Fortran, OpenACC directives

Multi-platform solution
x86-64, OpenPOWER and Arm CPUs
NVIDIA Tesla GPUs
Linux, Windows, MacOS

New! Support for Arm and Rome CPUs
PGI FORTRAN/C/C++ FOR ARM AND ROME

Xeon | EPYC | OpenPOWER | Arm

Use a single makefile and source code for Arm, Rome, Skylake or P9 CPUs + NVIDIA GPUs

Parallelize for multicore CPUs with MPI, OpenMP or OpenACC

Accelerate on NVIDIA GPUs with OpenACC and CUDA Fortran
QUANTUM ESPRESSO (QE)

AUSURF112

Multicore CPU vs GPU Performance

Runs completed with QEF/q-e-gpu at 3f8d6ce3
CUDA FORTRAN

real(8), device, allocatable, dimension(:) :: Xd, Yd, Rd
real(8), pinned, allocatable, dimension(:) :: X, Y
real(8) :: R

allocate (Xd(N), Yd(N), Rd((N+127)/128))
Xd = X(:)
Yd = Y(:)
call daxpy1<<<(N+127)/128, 128>>> (Yd,A,Xd,N)
call dsum1<<<(N+127)/128, 128>>> (Yd,Rd,N)
call dsum1<<<1, 128>>> (Rd,Rd,(N+127)/128)
Y(:) = Yd
R = Rd(1)
deallocate (Xd, Yd, Rd)

CPU Code

CUDA FORTRAN

attributes(global) subroutine daxpy1(Y,A,X,N)
real(8) :: Y(*), X(*)
integer, value :: N
real(8), value :: A
i = (blockidx%x-1) * blockdim%x + threadidx%x
if (i <= N) Y(i) = A * X(i) + Y(i)
end subroutine daxpy1

attributes(global) subroutine dsum1(Y,R,N)
real(8) :: Y(*), R(*)
integer, value :: N
real(8), shared :: part(128)
s = 0
j = (blockidx%x-1) * blockdim%x + threadidx%x
do i = j, N, blockdim%x * griddim%x
   s = s + Y(i)
enddo
j = threadidx%x ; k = blockdim%x
part(j) = s
do while (k > 1)
   k = k / 2
   if (j <= k) part(j) = part(j) + part(j+k)
call syncthreads()
enddo
R(blockidx%x) = part(1)
end subroutine dsum1

CUDA FORTRAN

attributes(global) subroutine daxpy1(Y,A,X,N)
real(8) :: Y(*), X(*)
integer, value :: N
real(8), value :: A
i = (blockidx%x-1) * blockdim%x + threadidx%x
if (i <= N) Y(i) = A * X(i) + Y(i)
end subroutine daxpy1

attributes(global) subroutine dsum1(Y,R,N)
real(8) :: Y(*), R(*)
integer, value :: N
real(8), shared :: part(128)
s = 0
j = (blockidx%x-1) * blockdim%x + threadidx%x
do i = j, N, blockdim%x * griddim%x
   s = s + Y(i)
enddo
j = threadidx%x ; k = blockdim%x
part(j) = s
do while (k > 1)
   k = k / 2
   if (j <= k) part(j) = part(j) + part(j+k)
call syncthreads()
enddo
R(blockidx%x) = part(1)
end subroutine dsum1

NVIDIA GPU Code

real(8), device, allocatable, dimension(:) :: Xd, Yd, Rd
real(8), pinned, allocatable, dimension(:) :: X, Y
real(8) :: R

allocate (Xd(N), Yd(N), Rd((N+127)/128))
Xd = X(:)
Yd = Y(:)
call daxpy1<<<(N+127)/128, 128>>> (Yd,A,Xd,N)
call dsum1<<<(N+127)/128, 128>>> (Yd,Rd,N)
call dsum1<<<1, 128>>> (Rd,Rd,(N+127)/128)
Y(:) = Yd
R = Rd(1)
deallocate (Xd, Yd, Rd)

CPU Code
CUDA FORTRAN

!$CUF KERNEL DO Directives

Real(8), device, allocatable, dimension(:) :: Xd, Yd, Rd
real(8), pinned, allocatable, dimension(:) :: X, Y
real(8) :: R

. . .

allocate (Xd(N), Yd(N), Rd((N+127)/128))
Xd = X(:)
Yd = Y(:)
R = 0.0
!$cuf kernel do(1) <<<*,*>>>
do i = 1, n
  Yd(i) = A * Xd(i) + Yd(i)
  R = R + Yd(i)
enddo

Y(:) = Yd

deallocate (Xd, Yd)

. . .
OPENACC: DIRECTIVES FOR ACCELERATED COMPUTING

How does it work?

#pragma acc enter data copyin(NH[0:n*n], OH[0:n*n])

for (iter = 0; iter < 100; ++iter) {
    #pragma acc parallel loop independent copyin(OH[0:n*n]) copy(NH[0:n*n])
    for (i = 1; i < n-1; ++i) {
        #pragma acc loop independent
        for (j = 1; j < n-1; ++j) {
            NH[j+i*n] = 0.25 * (OH[(j-1)+i*n] + OH[(j+1)+i*n] + OH[j+(i-1)*n] + OH[j+(i+1)*n]);
        }
    }
    tmp = OH; OH = NH; NH = tmp;
}

#pragma acc wait
#pragma acc exit data copyout(NH[0:n*n]) delete(OH[0:n*n])
OPENACC: DIRECTIVES FOR ACCELERATED COMPUTING

Compiling with the -ta=tesla:managed option

```c
#pragma acc enter data copyin(NH[0:n*n], OH[0:n*n])

for (iter = 0; iter < 100; ++iter) {
    #pragma acc parallel loop independent async
    for (i = 1; i < n-1; ++i) {
        #pragma acc loop independent
        for (j = 1; j < n-1; ++j) {
            NH[j+i*n] = 0.25 * (OH[(j-1)+i*n] + OH[(j+1)+i*n] +
                               OH[j+(i-1)*n] + OH[j+(i+1)*n]);
        }
    }
    tmp = OH; OH = NH; NH = tmp;
}

#pragma acc wait
#pragma acc exit data copyout(NH[0:n*n]) delete(OH[0:n*n])
```

C `malloc`, C++ `new`, Fortran `allocate` all mapped to CUDA Unified Memory
for (iter = 0; iter < 100; ++iter) {
    #pragma acc parallel loop independent async
    for (i = 1; i < n-1; ++i) {
        #pragma acc loop independent
        for (j = 1; j < n-1; ++j) {
            NH[j+i*n] = 0.25 * (OH[(j-1)+i*n] + OH[(j+1)+i*n] +
                              OH[j+(i-1)*n] + OH[j+(i+1)*n]);
        }
    }
    tmp = OH; OH = NH; NH = tmp;
}
#pragma acc wait

C malloc, C++ new, Fortran allocate all mapped to CUDA Unified Memory
OPENACC IS FOR MULTICORE CPUS AND GPUS

98 !$acc parallel
99 !$acc loop independent
100 DO k=y_min-depth,y_max+depth
101 !$acc loop independent
102 DO j=1,depth
103 density0(x_min-j,k)=left_density0(left_xmax+1-j,k)
104 ENDDO
105 ENDDO
106 !$acc end parallel

% pgfortran -ta=multicore -fast -Minfo=acc -c \update_tile_halo_kernel.f90
100, Loop is parallelizable
Generating Multicore code
102, Loop is parallelizable

$ acc loop gang

% pgfortran -ta=tesla -fast -Minfo=acc -c \update_tile_halo_kernel.f90
100, Loop is parallelizable
102, Loop is parallelizable

Accelerator kernel generated
Generating Tesla code
100, !$acc loop gang, vector(4) ! blockIdx%y threadIdx%y
102, !$acc loop gang, vector(32) ! blockIdx%x threadIdx%x
Performance measured October, 2019. Skylake: 2x20 core Intel Xeon Gold 6148 CPUs @ 2.4GHz w/ 376GB memory, hyperthreading enabled with two NVIDIA Tesla V100-PCIe-16GB GPU @ 1.53GHz. Rome: 2x24 core AMD EPYC 7352 CPUs @ 3.0GHz w/ 256GB memory. POWER9: 2x20 core IBM POWER9 DD2.2 @ 3.4GHz w/ 128GB memory.

SPEC® is a registered trademark of the Standard Performance Evaluation Corporation (www.spec.org).
The new IBM GRAF system provides hourly-updating, high-resolution forecasts at a global scale. To handle the increased load it runs on an IBM POWER9-based supercomputer with NVIDIA V100 GPUs using OpenACC. It’s the first time a global weather prediction system is being run operationally on GPUs.

“IBM GRAF

Todd Hutchinson, Head of Computational Meteorological Analysis and Prediction at The Weather Company, part of IBM

See Todd’s NVIDIA Theater talk 2:30pm Wednesday
For VASP, OpenACC is the way forward for GPU acceleration. Performance is similar to CUDA C, and OpenACC dramatically decreases GPU development and maintenance efforts.

“See a demo and learn about VASP for GPUs in the NVIDIA Developer Zone.”
PGI FORTRAN/C/C++ FOR ARM AND ROME

Xeon | EPYC | OpenPOWER | Arm

Fortran 2003/08, C11, C++17 compilers

CUDA Fortran, OpenACC, OpenMP, NVCC host compiler

Integrated LLVM code generators for Xeon, EPYC, POWER9 and Arm

Recompile and Run
PGI FORTRAN/C/C++ FOR ARM AND ROME

Xeon | EPYC | OpenPOWER | Arm

Fortran 2003/08, C11, C++17 compilers

CUDA Fortran, OpenACC, OpenMP, NVCC host compiler

Integrated LLVM code generators for Xeon, EPYC, POWER9 and Arm

Recompile and Run???
PGI FORTRAN/C/C++ FOR ARM AND ROME
Xeon | EPYC | OpenPOWER | Arm

Fortran 2003/08, C11, C++17 compilers

CUDA Fortran, OpenACC, OpenMP, NVCC host compiler

Integrated LLVM code generators for Xeon, EPYC, POWER9 and Arm

Recompile and Run???
PORTING FROM X86-64 TO ARM OR OPENPOWER

C/C++ ABI DIFFERENCES
- signed vs unsigned default char
- long double
- C varargs

NUMERICAL DIFFERENCES
- Intrinsics accuracy may differ across targets
- FMA vs. no FMA

PLATFORM DIFFERENCES
- Large memory model
- C varargs

TARGET-SPECIFIC FEATURES
- Inline *asm* statements
- SSE/AVX intrinsics
THE FUTURE OF HPC GPU PROGRAMMING
Math Libraries | Standard Languages | Directives | CUDA

GPU Programming with C++17
David Olsen, 11:00am Thursday

std::transform(par, x, x+n, y, y, [=] (float x, float y) {
    return y + a*x;
});

do concurrent (i = 1:n)
    y(i) = y(i) + a*x(i)
enddo

#pragma acc data copy(x,y)
{
    ...
    std::transform(par, x, x+n, y, y, [=](float x, float y){
        return y + a*x;
    });
    ...
}

__global__
void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] += a*x[i];
}

int main(void) {
    ...
    cudaMemcpy(d_x, x, ...);
    cudaMemcpy(d_y, y, ...);
    saxpy<<<(N+255)/256,256>>>(...);
    cudaMemcpy(y, d_y, ...);

GPU Accelerated C++17 and Fortran 2018

Incremental Performance Optimization with OpenACC

Maximize GPU Performance with CUDA C++/Fortran