PGI COMPILERS & TOOLS

ADVANCED OPENACC PERFORMANCE

Mathew Colgrove, PGI Compilers & Tools, NVIDIA

March 6th, 2018

NVIDIA

PROFILE, OPTIMIZE, PROFILE, OPTIMIZE, RINSE AND REPEAT

jsrun -n $nmpi -g 1 nvprof **-o clover.bm32.%p.prof** <path>/clover_leaf

nvprof -i clover.bm32.33405.prof >& clover.bm32.33405.txt

```
$ cat clover.bm32.33405.txt
======== Profiling result:
            Type Time(%)    Time     Calls       Avg       Min       Max  Name
 GPU activities: 7.54%  13.5475s      2955  4.5846ms  4.5446ms  4.6170ms  pdv_kernel_117_gpu
                 6.02%  10.8087s      2955  3.6578ms  3.6069ms  3.6777ms  pdv_kernel_80_gpu
                 5.75%  10.3340s      2955  3.4971ms  3.4545ms  3.5179ms  accelerate_kernel_65_gpu
                 4.88%  8.76880s      5910  1.4837ms  1.4514ms  1.5110ms  advec_mom_kernel_254_gpu
                 4.80%  8.62393s      5910  1.4592ms  1.4286ms  1.4850ms  advec_mom_kernel_189_gpu
                 4.74%  8.50618s      6208  1.3702ms  1.3358ms  1.4000ms  ideal_gas_kernel_51_gpu
                 4.44%  7.98215s      5910  1.3506ms  1.3307ms  1.3687ms  advec_mom_kernel_221_gpu
                 4.34%  7.79325s      5910  1.3187ms  1.2977ms  1.3403ms  advec_mom_kernel_157_gpu
                 4.24%  7.61488s      2955  2.5769ms  2.5658ms  2.5897ms  flux_calc_kernel_59_gpu
                 4.23%  7.59213s      2955  2.5692ms  2.5471ms  2.5891ms  advec_cell_k_255_gpu_1
                 3.91%  7.02675s      2955  2.3779ms  2.3744ms  2.3961ms  calc_dt_kernel_100_gpu
                 3.91%  7.02292s      2955  2.3766ms  2.3632ms  2.3954ms  advec_cell_168_gpu
                 .............
```

OPENACC DIRECTIVES FOR GPUS

```
% pgfortran -fast -ta=tesla -Minfo -c PdV_kernel.f90
pdv_kernel:
     ...
     77, Loop is parallelizable
     79, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         77, !$acc loop gang, vector(4) ! blockidx%y
                                        ! threadidx%y
         79, !$acc loop gang, vector(32)! blockidx%x
                                        ! threadidx%x
     ...
```

```
 75 !$ACC KERNELS
 76 !$ACC LOOP INDEPENDENT
 77     DO k=y_min,y_max
 78 !$ACC LOOP INDEPENDENT
 79       DO j=x_min,x_max
 80
 81         left_flux=  (xarea(j  ,k  )*(xvel0(j  ,k  )+xvel0(j  ,k+1)                &
 82                                     +xvel0(j  ,k  )+xvel0(j  ,k+1)))*0.25_8*dt*0.5
 83         right_flux= (xarea(j+1,k  )*(xvel0(j+1,k  )+xvel0(j+1,k+1)                &
 84                                     +xvel0(j+1,k  )+xvel0(j+1,k+1)))*0.25_8*dt*0.5
 85         bottom_flux=(yarea(j  ,k  )*(yvel0(j  ,k  )+yvel0(j+1,k  )                &
 86                                     +yvel0(j  ,k  )+yvel0(j+1,k  )))*0.25_8*dt*0.5
 87         top_flux=   (yarea(j  ,k+1)*(yvel0(j  ,k+1)+yvel0(j+1,k+1)                &
 88                                     +yvel0(j  ,k+1)+yvel0(j+1,k+1)))*0.25_8*dt*0.5
 89         total_flux=right_flux-left_flux+top_flux-bottom_flux
 90
 91         volume_change(j,k)=volume(j,k)/(volume(j,k)+total_flux)
 92
 93         min_cell_volume=MIN(volume(j,k)+right_flux-left_flux+top_flux-bottom_flux &
 94                            ,volume(j,k)+right_flux-left_flux                      &
 95                            ,volume(j,k)+top_flux-bottom_flux)
 97         recip_volume=1.0/volume(j,k)
 99         energy_change=(pressure(j,k)/density0(j,k)+viscosity(j,k)/density0(j,k))*...
101         energy1(j,k)=energy0(j,k)-energy_change
103         density1(j,k)=density0(j,k)*volume_change(j,k)
105       ENDDO
106     ENDDO
107 !$ACC END KERNELS
```

LOOP SCHEDULING

Collapse:

```
!$ACC KERNELS LOOP INDEPENDENT COLLAPSE(2)
      DO k=y_min,y_max
        DO j=x_min,x_max
          left_flux=  (xarea(j  ,k  )*(xvel0(j  ,k  )+xvel0(j  ,k+1)+xvel0(j  ,k  )+xvel0(j  ,k+1)))*0.25_8*dt*0.5
            ....
        ENDDO
      ENDDO
```

Explicit loop schedules:

```
!$ACC KERNELS LOOP INDEPENDENT GANG
      DO k=y_min,y_max
!$ACC LOOP INDEPENDENT VECTOR(512)
        DO j=x_min,x_max
          left_flux=  (xarea(j  ,k  )*(xvel0(j  ,k  )+xvel0(j  ,k+1)+xvel0(j  ,k  )+xvel0(j  ,k+1)))*0.25_8*dt*0.5
            ....
        ENDDO
      ENDDO
```

MEMORY COALESCING

Coalesced access:

A group of 32 contiguous threads (“warp”) accessing adjacent words

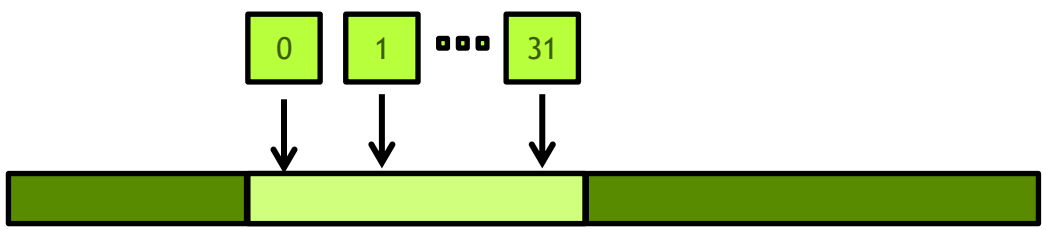
Few transactions and high utilization

Uncoalesced access:

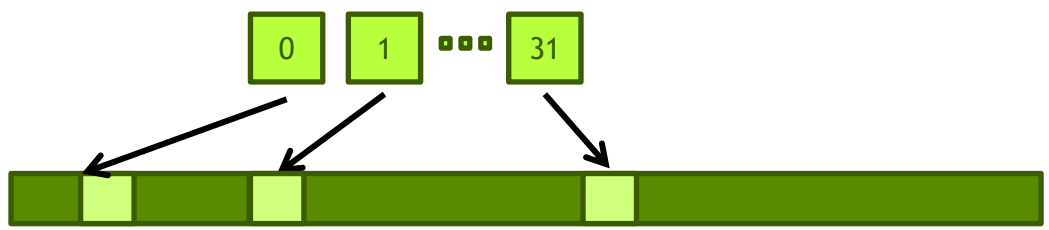
A warp of 32 threads accessing scattered words

Many transactions and low utilization

For best performance threadIdx.x should access contiguously



Coalesced



Uncoalesced

OPENACC ASYNC AND WAIT CLAUSES

async(n): launches work asynchronously in queue *n*

wait(n): blocks host until all operations in queue *n* have completed

Can significantly reduce launch latency, enables pipelining and concurrent operations

```
#pragma acc parallel loop async(1)
for(int i=0; i<N; i++)
  ...
#pragma acc parallel loop async(1)
for(int i=0; i<N; i++)
  ...
#pragma acc wait(1)
```

PINNED MEMORY

```
% pgfortran -fast -ta=tesla:pinned -Minfo -c PdV_kernel.f90
```

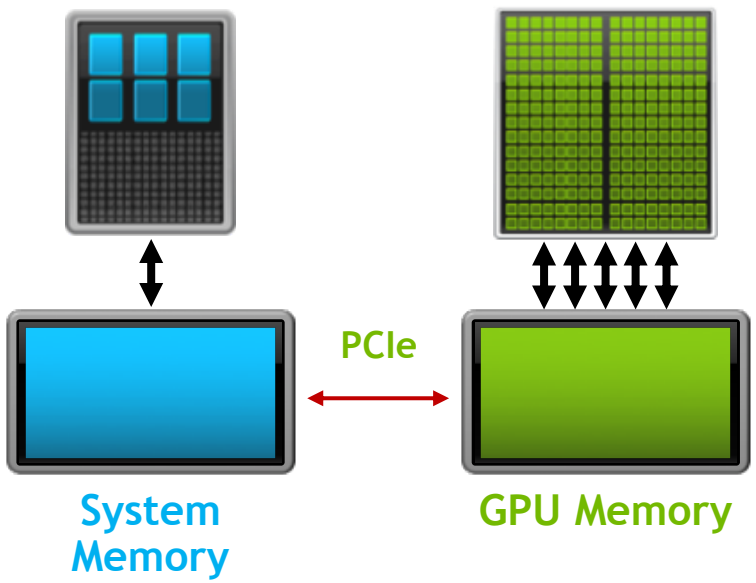
- Data transfers between the host and device must be done from CPU non-paged (“pinned”) memory. By default, PGI creates buffers in pinned memory to perform the transfer.
- Using “pinned” the host data is allocated directly in non-paged memory eliminating the need to copy data from virtual to pinned memory.
- Caveats:
  - Need enough physical memory for the pinned memory
  - Allocation of pinned memory has a high overhead cost

- Best used when data is transferred often but has few allocations/deallocations

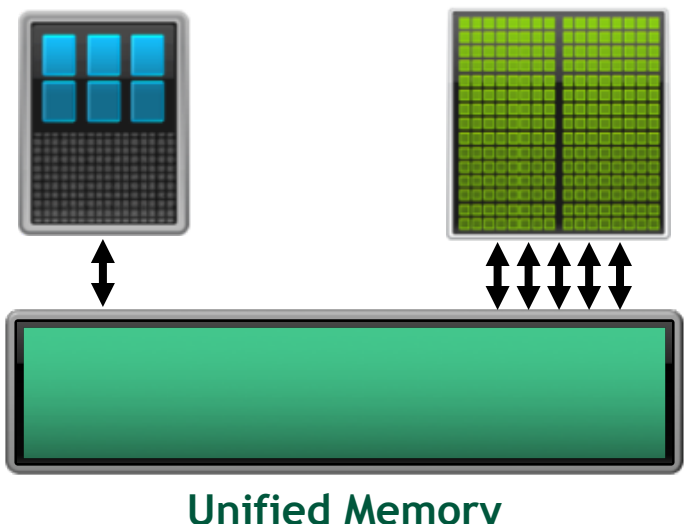
Programming GPU-Accelerated Systems

CUDA Unified Memory for Dynamically Allocated Data

GPU Developer View



GPU Developer View With CUDA Unified Memory

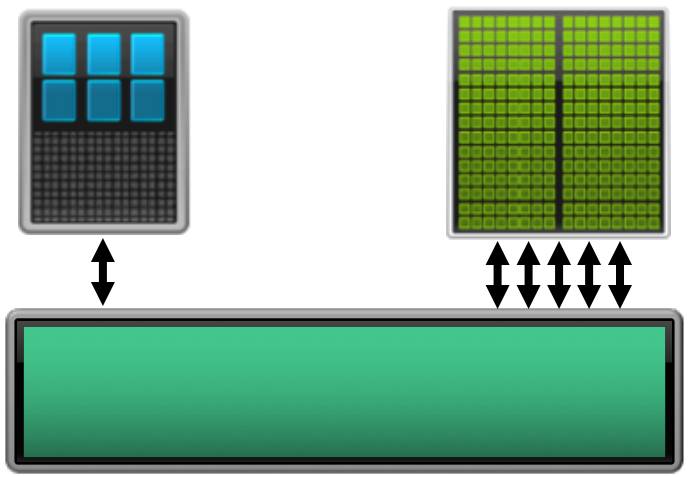


PGI OpenACC and CUDA Unified Memory

Compiling with the -ta=tesla:managed option

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          c[i] = a[i] + b[i];
          ...
      }
  }
  ...
}
```

GPU Developer View With CUDA Unified Memory



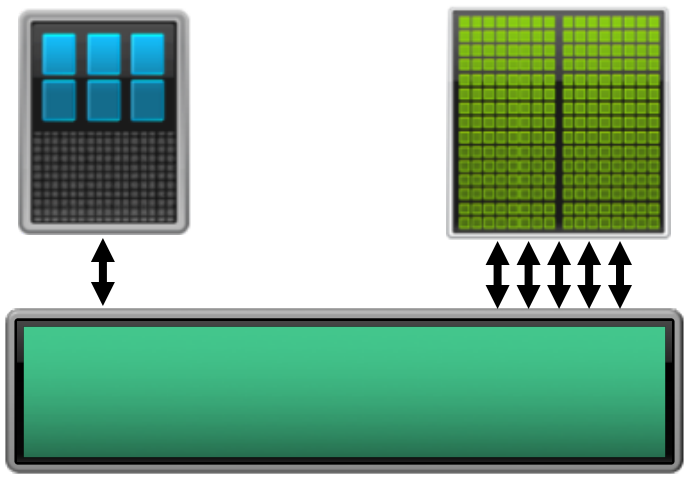
C `malloc`, C++ `new`, Fortran `allocate` all mapped to CUDA Unified Memory

PGI OpenACC and CUDA Unified Memory

Compiling with the -ta=tesla:managed option

```
...
#pragma acc parallel
{
#pragma acc loop gang vector
    for (i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
        ...
    }
}
...
```

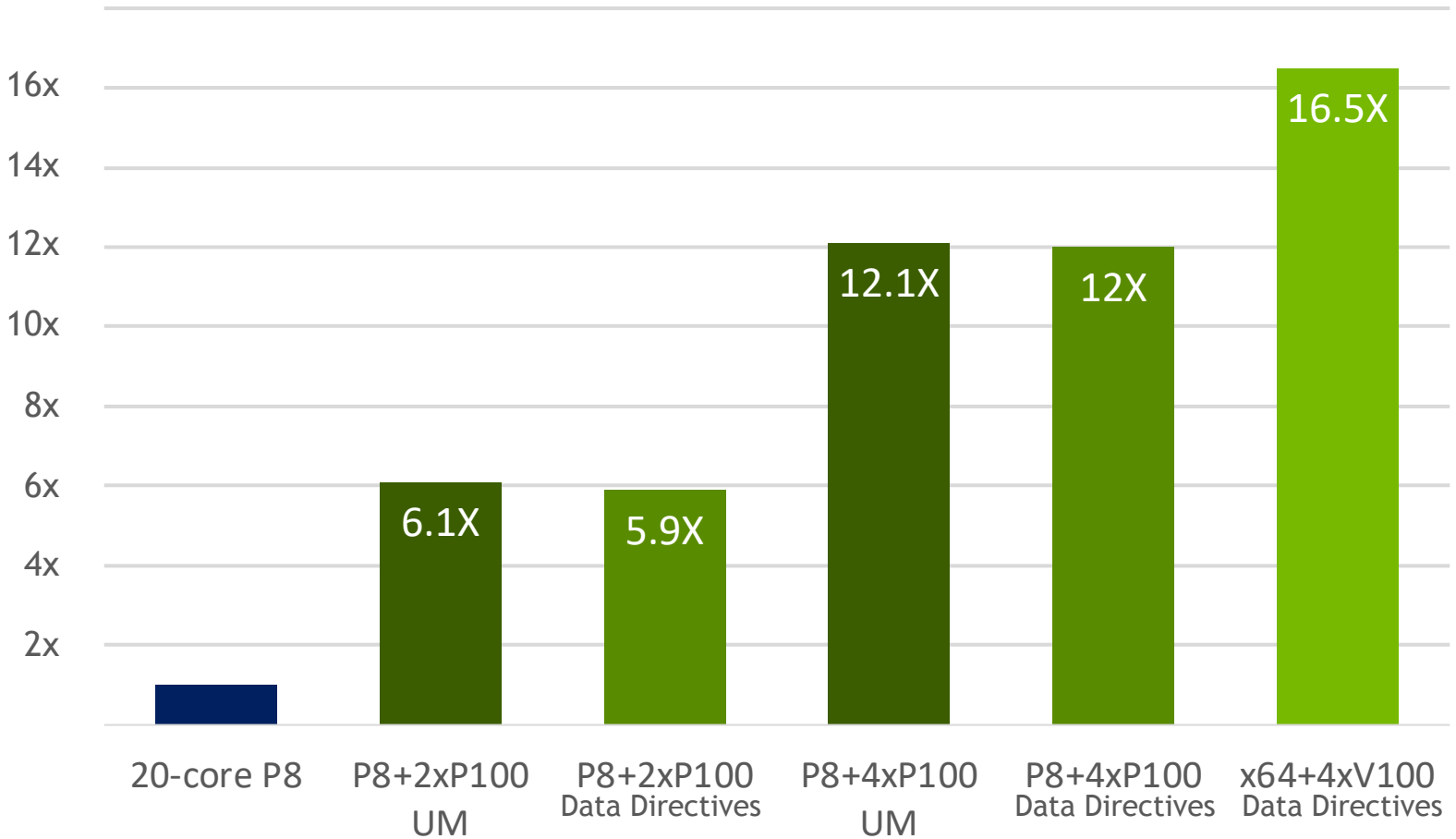
GPU Developer View With CUDA Unified Memory



C `malloc`, C++ `new`, Fortran `allocate` all mapped to CUDA Unified Memory

GTC Performance using OpenACC

OpenPOWER | NVLink | Unified Memory | P100 | V100



P8 : IBM POWER8NVL, 2 sockets, 20 cores, NVLINK
UM : No Data Directives in sources, compiled with -ta=tesla:managed

PGI

PGI OPENACC POOL ALLOCATOR

Environment Variable	Use
PGI_ACC_POOL_ALLOC	Disable the pool allocator. The pool allocator is enabled by default; to disable it, set PGI_ACC_POOL_ALLOC to 0.
PGI_ACC_POOL_SIZE	The default size is 1GB but other sizes can be used. The actual pool size is set such that the size is the nearest, smaller number in the Fibonacci series compared to the provided or default size. If necessary, the pool allocator will add more pools but only up to the PGI_ACC_POOL_THRESHOLD value.
PGI_ACC_POOL_THRESHOLD	Set the percentage of total device memory that the pool allocator can occupy. The default is set to 50% but other percentages can be used.
PGI_ACC_POOL_ALLOC_MAXSIZE	Set the maximum size for allocations. The default maximum size for allocations is 500MB but another size (i.e., 100KB, 10MB, 250MB, etc.) can be used as long as it is greater than or equal to 16B.
PGI_ACC_POOL_ALLOC_MINSIZE	Set the percentage of total device memory that the pool allocator can occupy. The default is set to 50% but other percentages can be used.

FORTRAN 2018 DO CONCURRENT

Fortran 2018 DO CONCURRENT

+ True Parallel Loops
+ Loop-scope shared/private data
– No support for reductions
– No support for atomics
– No support for data management

```
 75
 76
 77     DO CONCURRENT (k=y_min:y_max, j=x_min:x_max) &
 78                         LOCAL (right_flux,left_flux,top_flux,bottom_flux,total_flux, &
                                    min_cell_volume,energy_change,recip_volume)
 79
 80
 81         left_flux=  (xarea(j  ,k  )*(xvel0(j  ,k  )+xvel0(j  ,k+1)                &
 82                                     +xvel0(j  ,k  )+xvel0(j  ,k+1)))*0.25_8*dt*0.5
 83         right_flux= (xarea(j+1,k  )*(xvel0(j+1,k  )+xvel0(j+1,k+1)                &
 84                                     +xvel0(j+1,k  )+xvel0(j+1,k+1)))*0.25_8*dt*0.5
 85         bottom_flux=(yarea(j  ,k  )*(yvel0(j  ,k  )+yvel0(j+1,k  )                &
 86                                     +yvel0(j  ,k  )+yvel0(j+1,k  )))*0.25_8*dt*0.5
 87         top_flux=   (yarea(j  ,k+1)*(yvel0(j  ,k+1)+yvel0(j+1,k+1)                &
 88                                     +yvel0(j  ,k+1)+yvel0(j+1,k+1)))*0.25_8*dt*0.5
 89         total_flux=right_flux-left_flux+top_flux-bottom_flux
 90
 91         volume_change(j,k)=volume(j,k)/(volume(j,k)+total_flux)
 92
 93         min_cell_volume=MIN(volume(j,k)+right_flux-left_flux+top_flux-bottom_flux &
 94                            ,volume(j,k)+right_flux-left_flux                      &
 95                            ,volume(j,k)+top_flux-bottom_flux)
 97         recip_volume=1.0/volume(j,k)
 99         energy_change=(pressure(j,k)/density0(j,k)+viscosity(j,k)/density0(j,k))*...
101         energy1(j,k)=energy0(j,k)-energy_change
103         density1(j,k)=density0(j,k)*volume_change(j,k)
105
106     ENDDO
107
```