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

Introduction to OpenACC

Dr. Aleksei Ivakhnenko

March 11, 2018

Outline

- Understanding the PGI compiler output
  - Compiler flags and environment variables
  - Compiler limitations in dependencies tracking
- Organizing data persistence regions using data directives
- Data clauses
- Hands-on:
  - “Fill-in” exercise on implementing efficient CUFFT + OpenACC 2D Poisson solver version.
  - Adding OpenACC directives step by step

PGI accelerator info

```
$ /opt/pgi/linux86-64/2016/bin/pgaccelinfo
CUDA Driver Version:           8000
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  ↩
       361.62
                               Tue May 24 20:21:31 PDT 2016

Device Number:                 0
Device Name:                   Tesla K40c
Device Revision Number:        3.5
Global Memory Size:            12884705280
Number of Multiprocessors:     15
Number of SP Cores:            2880
Number of DP Cores:            960
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
```

```
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    745 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   No
Memory Clock Rate:             3004 MHz
Memory Bus Width:              384 bits
L2 Cache Size:                 1572864 bytes
Max Threads Per SMP:           2048
Async Engines:                 2
Unified Addressing:            Yes
Managed Memory:                Yes
PGI Compiler Option:           -ta=tesla:cc35
```

Compiler info flags

```
-Minfo[=all|accel|ccff|ftn|inline|intensity|ipa|loop|lre|mp|opt|par|pfo|stat|time|
       unified|vect]
                    Generate informational messages about optimizations
```

- `all`: -Minfo=accel,inline,ipa,loop,lre,mp,opt,par,unified,vect
- `accel`: Enable Accelerator information
- `ccff`: Append information to object file
- `ftn`: Enable Fortran-specific information
- `inline`: Enable inliner information
- `intensity`: Enable compute intensity information
- `ipa`: Enable IPA information
- `loop`: Enable loop optimization information
- `lre`: Enable LRE information
- `mp`: Enable OpenMP information
- `opt`: Enable optimizer information
- `par`: Enable parallelizer information
- `pfo`: Enable profile feedback information
- `stat`: Same as -Minfo=time
- `time`: Display time spent in compiler phases
- `unified`: Enable unified binary information
- `vect`: Enable vectorizer information

Compiler target flags

```
-ta=tesla:{cc20|cc30|cc35|cc50|cc60|cuda7.0|cuda7.5|cuda8.0|fastmath|[no]flushz|[↩
    no]fma|keepbin|keepgpu|keepptx|[no]lineinfo|[no]llvm|loadcache:{L1|L2}|↩
    maxregcount:<n>|pinned|[no]rdc|safecache|[no]unroll|managed|beta}|nvidia
                    Choose target accelerator
```

- `tesla|nvidia:` - Select NVIDIA Tesla accelerator target
  - `cc20`: Compile for compute capability 2.0
  - `cc30`: Compile for compute capability 3.0
  - `cc35`: Compile for compute capability 3.5
  - `cc50`: Compile for compute capability 5.0
  - `cc60`: Compile for compute capability 6.0
  - `cuda7.0`: Use CUDA 7.0 Toolkit compatibility (default)
  - `cuda7.5`: Use CUDA 7.5 Toolkit compatibility
  - `cuda8.0`: Use CUDA 8.0 Toolkit compatibility
  - `fastmath`: Use fast math library
  - `keepbin`: Keep kernel .bin files
  - `keepgpu`: Keep kernel source files
  - `keepptx`: Keep kernel .ptx files
  - `[no]llvm`: Use LLVM back end
  - `pinned`: Use CUDA Pinned Memory
  - `[no]unroll`: Enable automatic inner loop unrolling (default at -O3)
  - `beta`: Enable beta code generation

Compiler target flags

```
-ta=tesla:{cc20|cc30|cc35|cc50|cc60|cuda7.0|cuda7.5|cuda8.0|fastmath|[no]flushz|[↩
    no]fma|keepbin|keepgpu|keepptx|[no]lineinfo|[no]llvm|loadcache:{L1|L2}|↩
    maxregcount:<n>|pinned|[no]rdc|safecache|[no]unroll|managed|beta}|nvidia
                    Choose target accelerator
```

- `tesla|nvidia:` - Select NVIDIA Tesla accelerator target
- `managed`: Use CUDA Managed Memory
- `[no]flushz`: Enable flush-to-zero mode on the GPU
- `[no]fma`: Generate fused mul-add instructions (default at -O3)
- `[no]lineinfo`: Generate GPU line information
- `loadcache`: Choose what hardware level cache to use for global memory loads - L1 or L2
- `maxregcount:<n>`: Set maximum number of registers to use on the GPU
- `[no]rdc`: Generate relocatable device code
- `safecache`: Allows variable-sized arrays in cache directives and assumes they fit GPU shared memory

Environment variables

- `ACC_NOTIFY` — to see if anything has been executed on the GPU
- `PGI_ACC_TIME` — to see time/counts/configuration of GPU data transfers and kernel launches
- `ACC_DEVICE_TYPE` — sets the default device type to use
- `ACC_DEVICE_NUM` — sets the default device number to use

Dependency tracking

PGI compiler tracks data dependencies:

- Generates output info for dependent data preventing parallel execution
  - ✓ Helps to find code regions which have to be rewritten
- Possible misinterpreting of code leads to sequential execution
  - ✓ Separate compilation
  - ✓ Unlike allocatable arrays in Fortran, C/C++ pointers do not provide array length info

Dependency tracking: failure

1: dependency.cpp

```
int n = 1000;

float* a = (float*)malloc(n*sizeof(float));
float* b = (float*)malloc(n*sizeof(float));
float* c = (float*)malloc(n*sizeof(float));

for (int i = 0; i<n; i++)
{
    a[i] = (float)rand() / RAND_MAX;
    b[i] = (float)rand() / RAND_MAX;
}

vecadd(a, b, c, n);

free(a);
free(b);
free(c);
```

2: dependency_vecadd.hpp

```
void vecadd(
    float *a, float *b, float *c, int n);
```

3: dependency_vecadd.cpp

```
void vecadd(
    float *a, float *b, float *c, int n)
{
    #pragma acc parallel
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Dependency tracking: failure

```
$ pgc++ dependency_vecadd.cpp -c -acc -Minfo=accel -ta=nvidia,time -o
dependency_vecadd.o
dependency_vecadd.cpp:
vecadd(float *, float *, float *, int):
      2, Accelerator kernel generated
         Generating Tesla code
         Generating copyout(c[:n])
         Generating copyin(a[:n],b[:n])
      4, Complex loop carried dependence of b->,a-> prevents parallelization
         Loop carried dependence of c-> prevents parallelization
         Loop carried backward dependence of c-> prevents vectorization
```

Dependency tracking: solution 1

4: dependency.cpp

```
int n = 1000;

float* a = (float*)malloc(n*sizeof(float));
float* b = (float*)malloc(n*sizeof(float));
float* c = (float*)malloc(n*sizeof(float));

for (int i = 0; i<n; i++)
{
    a[i] = (float)rand() / RAND_MAX;
    b[i] = (float)rand() / RAND_MAX;
}

vecadd(a, b, c, n);

free(a);
free(b);
free(c);
```

5: dependency_vecadd.hpp

```
void vecadd(
    float *a, float *b, float *c, int n);
```

6: dependency_vecadd.cpp

```
void vecadd(
    float *a, float *b, float *c, int n)
{
    #pragma acc parallel loop independent
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Dependency tracking: solution 2

7: dependency.cpp

```
int n = 1000;

float* a = (float*)malloc(n*sizeof(float));
float* b = (float*)malloc(n*sizeof(float));
float* c = (float*)malloc(n*sizeof(float));

for (int i = 0; i<n; i++)
{
    a[i] = (float)rand() / RAND_MAX;
    b[i] = (float)rand() / RAND_MAX;
}

vecadd(a, b, c, n);

free(a);
free(b);
free(c);
```

8: dependency_vecadd.hpp

```
void vecadd(
    float * restrict a, float * restrict b,
    float * restrict c, int n);
```

9: dependency_vecadd.cpp

```
void vecadd(
    float * restrict a, float * restrict b,
    float * restrict c, int n)
{
    #pragma acc parallel
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Dependency tracking: solution 3

```
$ pgc++ dependency_vecadd.cpp -c -acc -Minfo=accel -ta=nvidia,time -o dependency_vecadd.o ↩
    -Msafeptr
dependency_vecadd.cpp:
vecadd(float *, float *, float *, int):
      2, Accelerator kernel generated
         Generating Tesla code
      4, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      2, Generating copyout(c[:n])
         Generating copyin(a[:n],b[:n])
```

Directive syntax

- Fortran:

```
!$acc parallel loop [clause ...]
      do loop
[!$acc end parallel loop]

!$acc kernels loop [clause ...]
      do loop
[!$acc end kernels loop]
```

- C:

```
#pragma acc parallel loop [clause ...]
      for loop

#pragma acc kernels loop [clause ...]
      for loop
```

A note on OpenACC array indices ranges

Note that in OpenACC for C and for Fortran arrays indices ranges are specified differently:

- Fortran:
  Syntax: `array(begin : end)`
  Examples: `a(:, :), a(1:100, 2:n)`

- C:
  Syntax: `array[begin : length]`
  Examples: `a[2:n] // a[2], a[3], ..., a[2+n-1]`

A note on OpenACC array indices ranges

Note that in OpenACC for C and for Fortran arrays indices ranges are specified differently:

- Fortran:
  Syntax: `array(begin : end)`
  Examples: `a(:, :), a(1:100, 2:n)`

- C:
  Syntax: `array[begin : length]`
  Examples: `a[2:n] // a[2], a[3], ..., a[2+n-1]`

Data clauses

```
copy* (list)
create (list)
present (list)
present_or_copy* (list)
present_or_create (list)
deviceptr (list)
private (list)
firstprivate (list)

*<blank>|in|out
```

Data region example

If **parallel** directives are separated, this code would require individual data copies for all kernels:

```
int a [1000];
int b [1000];
#pragma acc parallel
for (int i = 0; i < 1000; i++)
{
    a[i] = i - 100 + 23;
}
#pragma acc parallel
for (int j = 0; j < 1000; j++)
{
    b[j] = a[j] - j - 10 + 213;
}
```

Data region example

```
$ pgcc no_region.c -acc -Minfo=accel -ta=nvidia,time -o no_region
no_region.c:
main:
     10, Accelerator kernel generated
         Generating Tesla code
     11, #pragma acc loop vector(128) /* threadIdx.x */
     10, Generating copyout(a[:])
     11, Loop is parallelizable
     16, Accelerator kernel generated
         Generating Tesla code
     17, #pragma acc loop vector(128) /* threadIdx.x */
     16, Generating copyout(b[:])
         Generating copyin(a[:])
     17, Loop is parallelizable
```

Data region example

Copying time is $19 + 51 + 15 = 85$:

```
Accelerator Kernel Timing data
/scratch/aivahnenko/openacc/examples/data_regions/no_region.c
  main NVIDIA devicenum=0
    time(us): 97
    10: compute region reached 1 time
        10: kernel launched 1 time
            grid: [1] block: [128]
             device time(us): total=5 max=5 min=5 avg=5
            elapsed time(us): total=594 max=594 min=594 avg=594
    10: data region reached 1 time
    16: compute region reached 1 time
        16: kernel launched 1 time
            grid: [1] block: [128]
            device time(us): total=7 max=7 min=7 avg=7
            elapsed time(us): total=58 max=58 min=58 avg=58
    16: data region reached 2 times
        16: data copyin transfers: 1
            device time(us): total=19 max=19 min=19 avg=19
        16: data copyout transfers: 1
            device time(us): total=51 max=51 min=51 avg=51
    21: data region reached 1 time
        21: data copyout transfers: 1
            device time(us): total=15 max=15 min=15 avg=15
```

Data region example

If **parallel** directives are inside the data region, this code would require only one data transfer:

```
int a [1000];
int b [1000];
#pragma acc data copyout (a[0:1000],b[0:1000])
{
    #pragma acc parallel
    for (int i=0; i<1000; i++)
    {
        a[i]=i-100+23;
    }
    #pragma acc parallel
    for (int j=0; j<1000; j++)
    {
        b[j]=a[j]-j-10+213;
    }
}
```

Data region example

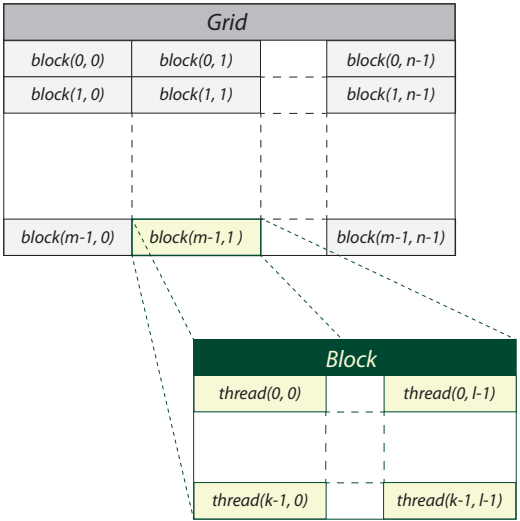
```
$ pgcc region.c -acc -Minfo=accel -ta=nvidia,time -o region
region.c:
main:
      9, Generating copyout(a[:],b[:])
     11, Accelerator kernel generated
         Generating Tesla code
     12, #pragma acc loop vector(128) /* threadIdx.x */
         Loop is parallelizable
     17, Accelerator kernel generated
         Generating Tesla code
     18, #pragma acc loop vector(128) /* threadIdx.x */
         Loop is parallelizable
```

Data region example

Copying time is 63 (26% lower):

```
Accelerator Kernel Timing data
/scratch/aivahnenko/openacc/examples/data_regions/region.c
  main NVIDIA devicenum=0
    time(us): 76
    9: data region reached 1 time
    11: compute region reached 1 time
        11: kernel launched 1 time
            grid: [1] block: [128]
             device time(us): total=6 max=6 min=6 avg=6
            elapsed time(us): total=547 max=547 min=547 avg=547
    17: compute region reached 1 time
        17: kernel launched 1 time
            grid: [1] block: [128]
             device time(us): total=7 max=7 min=7 avg=7
            elapsed time(us): total=35 max=35 min=35 avg=35
    23: data region reached 1 time
        23: data copyout transfers: 2
             device time(us): total=63 max=52 min=11 avg=31
```

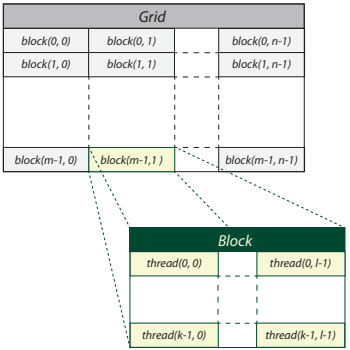
Mapping to CUDA blocks and threads



Mapping to CUDA blocks and threads

- Inlying loops generate multi-dimensional blocks and grids
  - In general: gang -> blocks, vector -> threads
- Example: 2D loop -> grid[100x200], block[16x32]

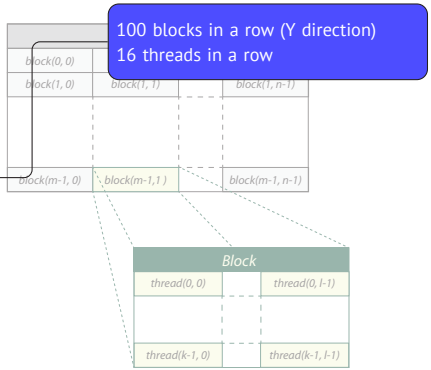
```
...
#pragma acc kernels loop gang(100), vector(16)
for( ... )
{
    #pragma acc loop gang(200), vector(32)
    for( ... )
}
...
```



Mapping to CUDA blocks and threads

- Inlying loops generate multi-dimensional blocks and grids
  - In general: gang -> blocks, vector -> threads
- Example: 2D loop -> grid[100x200], block[16x32]

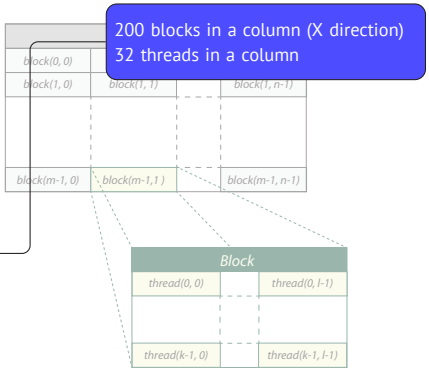
```
...
#pragma acc kernels loop gang(100), vector(16)
for( ... )
{
    #pragma acc loop gang(200), vector(32)
    for( ... )
}
...
```



Mapping to CUDA blocks and threads

- Inlying loops generate multi-dimensional blocks and grids
  - In general: gang -> blocks, vector -> threads
- Example: 2D loop -> grid[100x200], block[16x32]

```
...
#pragma acc kernels loop gang(100), vector(16)
for( ... )
{
    #pragma acc loop gang(200), vector(32)
    for( ... )
}
...
```



Mapping to CUDA blocks and threads

```
#pragma acc kernels
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
#pragma acc kernels loop gang(100) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
    #pragma acc loop gang vector
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

Mapping to CUDA blocks and threads

```
// n/32 blocks, 32 threads each by default
#pragma acc kernels
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
#pragma acc kernels loop gang(100) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
    #pragma acc loop gang vector
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

Mapping to CUDA blocks and threads

```
// n/32 blocks, 32 threads each by default
#pragma acc kernels
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
// 100 blocks of 128 threads,
// each thread executes one iteration of a loop with kernels directive
#pragma acc kernels loop gang(100) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
    #pragma acc loop gang vector
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

Mapping to CUDA blocks and threads

```
// n/32 blocks, 32 threads each by default
#pragma acc kernels
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
// 100 blocks of 128 threads,
// each thread executes one iteration of a loop with kernels directive
#pragma acc kernels loop gang(100) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
// 100 blocks of 128 threads,
// each thread executes one iteration of a loop with parallel directive
#pragma acc parallel num_gangs(100) vector_length(128)
{
    #pragma acc loop gang vector
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

Mapping to CUDA blocks and threads

```
#pragma acc parallel num_gangs(100)
{
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

```
#pragma acc parallel num_gangs(100)
{
    #pragma acc loop gang
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

Mapping to CUDA blocks and threads

```
// 100 blocks, 32 threads each by default
#pragma acc parallel num_gangs(100)
{
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

```
#pragma acc parallel num_gangs(100)
{
    #pragma acc loop gang
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

Mapping to CUDA blocks and threads

```
// 100 blocks, 32 threads each by default
#pragma acc parallel num_gangs(100)
{
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

```
// 100 blocks, 32 threads each by default
#pragma acc parallel num_gangs(100)
{
    #pragma acc loop gang
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

Mapping to CUDA blocks and threads

Each thread can execute one or more iterations of the loop. This depends on the mapping parameters. Executing several iterations can increase the performance by lowering the initialization time:

```
#pragma acc kernels loop gang(100) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
#pragma acc kernels loop gang(50) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

Mapping to CUDA blocks and threads

Each thread can execute one or more iterations of the loop. This depends on the mapping parameters. Executing several iterations can increase the performance by lowering the initialization time:

```
// 100 blocks, 128 threads each
#pragma acc kernels loop gang(100) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
#pragma acc kernels loop gang(50) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

Mapping to CUDA blocks and threads

Each thread can execute one or more iterations of the loop. This depends on the mapping parameters. Executing several iterations can increase the performance by lowering the initialization time:

```
// 100 blocks, 128 threads each
#pragma acc kernels loop gang(100) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```

```
// 50 blocks, 128 threads each
#pragma acc kernels loop gang(50) vector(128)
for (int i = 0; i < n; i++)
    y[i] += a*x[i];
```