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

Advanced OpenACC Programming

Dr. Aleksei Ivakhnenko

March 11, 2018

Outline

- Profiling GPU kernels in OpenACC application:
  - *time* option
  - `PGI_ACC_TIME` environment variable
  - Profiling with pgprof
- GPU architecture, efficient GPU code
- Expressing locality in loop nest using `tile` clause
- Organizing asynchrony using `async` clause and `wait` directive
- Atomic directive
  - Allowed operations
  - Restrictions
- Advanced code optimization practices, by examples.
  - Restrictions
  - Common mistakes
  - Workarounds

Profiling

- Profiling gives a solid understanding of how an application spends the execution time
  - Most heavy regions, bottlenecks
- OpenACC profiling options:
  - `time` compiler option
  - `PGI_ACC_TIME` environment variable
  - NVIDIA Visual Profiler (nvvp/pgprof)
  - NVIDIA command-line profiler (nvprof)

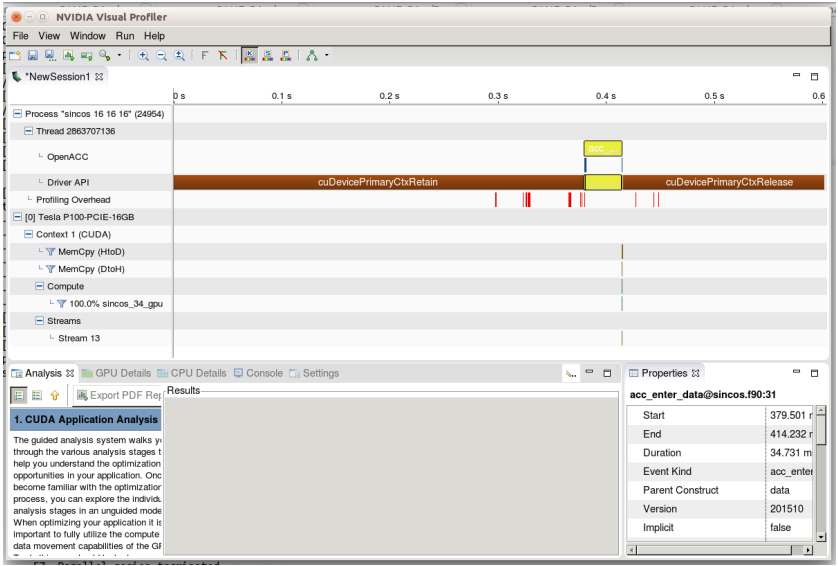
NVIDIA command-line profiler (nvprof) flags

- `-openacc-profiling <on|off>`
  Turn on/off OpenACC profiling. Note: OpenACC profiling is only supported on x86_64 and Power8 Linux. Default is on.
- `-print-openacc-summary`
  Print a summary of all recorded OpenACC activities.
- `-print-openacc-trace`
  Print a detailed trace of all recorded OpenACC activities, including each activity's timestamp and duration.
- `-print-openacc-constructs`
  Include the name of the OpenACC parent construct that caused an OpenACC activity to be emitted. Note that for applications using PGI OpenACC runtime before 16.1, this value will always be unknown.
- `-openacc-summary-mode <exclusive|inclusive>`
  Specify how activity durations are presented in the OpenACC summary. Allowed values: "exclusive" - exclusive durations (default). "inclusive" inclusive durations. See OpenACC Summary Modes for more information.

NVIDIA command-line profiler (nvprof)

```
==8827== Profiling application: ./dependency1
==8827== Profiling result:
Time(%)      Time  Calls       Avg       Min       Max  Name
 62.70%  11.778us      2  5.8890us  5.5690us  6.2090us  [CUDA memcpy HtoD]
 23.51%  4.4160us      1  4.4160us  4.4160us  4.4160us  [CUDA memcpy DtoH]
 13.80%  2.5920us      1  2.5920us  2.5920us  2.5920us  _Z6vecaddPfS_S_i_2_gpu
==8827== API calls:
Time(%)      Time  Calls       Avg       Min       Max  Name
85.84%   386.32ms      1 386.32ms  386.32ms  386.32ms  cuCtxCreate
13.56%   61.040ms      1 61.040ms  61.040ms  61.040ms  cuMemHostAlloc
0.24%    1.0889ms      1 1.0889ms  1.0889ms  1.0889ms  cuMemAllocHost
0.21%    941.06us      4 235.26us  9.8680us  479.28us  cuMemAlloc
0.05%    239.55us      1 239.55us  239.55us  239.55us  cuModuleLoadData
0.02%    78.488us      9 8.7200us  3.0380us  30.140us  cuEventRecord
0.02%    72.419us      2 36.209us  20.628us  51.791us  cuMemcpyHtoDAsync
0.01%    67.319us      1 67.319us  67.319us  67.319us  cuStreamCreate
0.01%    65.466us      1 65.466us  65.466us  65.466us  cuLaunchKernel
0.01%    42.055us      5 8.4110us  1.5540us  17.727us  cuEventSynchronize
0.00%    21.269us      1 21.269us  21.269us  21.269us  cuMemcpyDtoHAsync
0.00%    14.038us      4 3.5090us  1.0750us  8.6130us  cuEventCreate
0.00%    11.670us      8 1.4580us     526ns  4.5340us  cuDeviceGet
0.00%    10.901us      4 2.7250us  2.1980us  3.8630us  cuEventElapsedTime
0.00%    10.766us      2 5.3830us  1.0050us  9.7610us  cuDeviceGetCount
0.00%    10.668us      1 10.668us  10.668us  10.668us  cuModuleGetFunction
0.00%    10.495us      3 3.4980us  2.0350us  6.0010us  cuStreamSynchronize
0.00%    5.1980us      4 1.2990us     367ns  3.3090us  cuDeviceComputeCapability
0.00%    2.4950us      1 2.4950us  2.4950us  2.4950us  cuDriverGetVersion
0.00%    2.1270us      2 1.0630us     587ns  1.5400us  cuCtxGetCurrent
0.00%    1.1050us      1 1.1050us  1.1050us  1.1050us  cuCtxSetCurrent
0.00%      931ns      1     931ns     931ns     931ns  cuCtxAttach
```

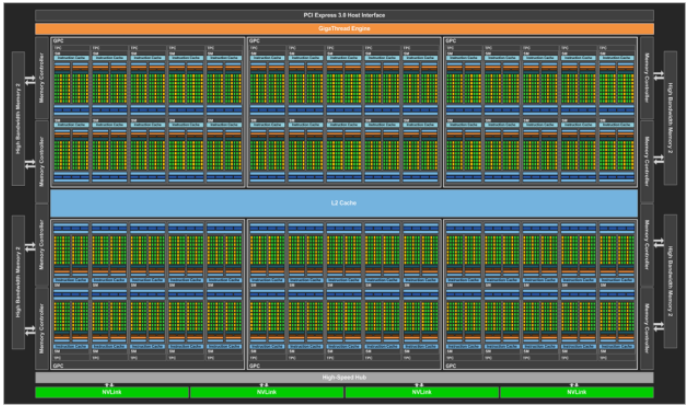
NVIDIA Visual Profiler (nvvp/pgprof)



Pascal Architecture

Pascal SMM Specifications:

- 56 SMs
- 4096-bit HBM2 memory interface
- 4096 KB L2 cache
- NVLink support
- 16-nm FinFET
- 15,3 billion transistors
- 5,3 TFLOPS of double precision
- 10,6 TFLOPS of single precision
- 21,2 TFLOPS of half-precision



Pascal Architecture

Pascal SMM Specifications:

- Up to 3840 CUDA cores
- 14336 KB register file size/GPU
- Up to 64 KB of shared memory

Volta Architecture

Volta SMM Specifications:

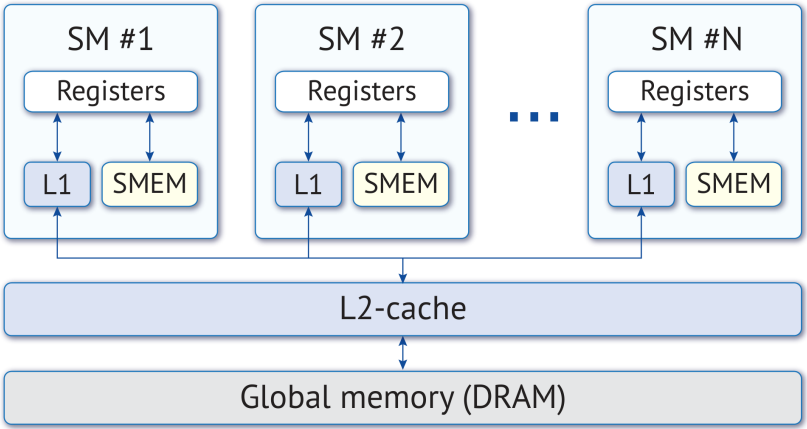
- 84 SMs
- 4096-bit HBM2 memory interface
- 6144 KB L2 cache
- NVLink support
- 12nm FFN
- 21.1 billion transistors
- 7,5 TFLOPS of double precision
- 15 TFLOPS of single precision
- 30 TFLOPS of half-precision

Volta Architecture

Volta SMM Specifications:

- Up to 5376 CUDA cores
- 14336 KB register file size/GPU
- Up to 128 KB of shared memory

GPU memory hierarchy



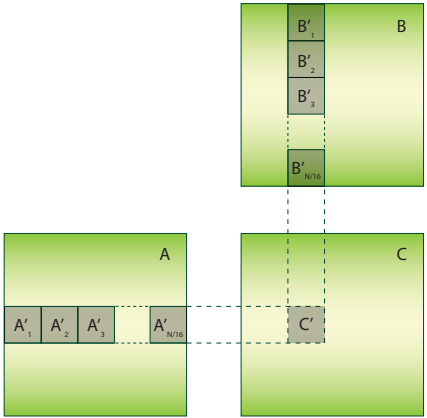
OpenACC `tile` clause

`tile( size-expr-list )`

- Gives a way to express locality within a loop nest
- Guides the compiler to additional optimization options, e.g. intra-tile data caching/reuse

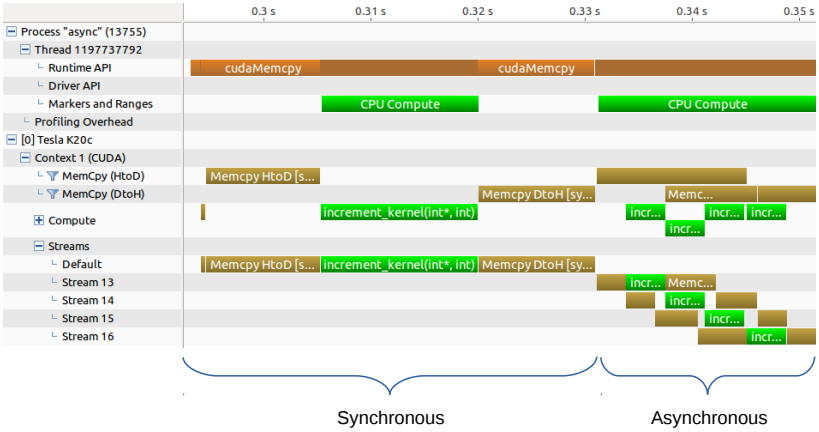
```
#pragma acc parallel loop private(i,j) tile(8,8)
for(i = 0; i < rows; i++)
{
    for(j = 0; j < cols; j++)
    {
        out[i*rows + j] = in[j*cols + i];
    }
}
```

By adding `tile(8,8)`, compiler automatically introduces two additional loops that work on an 8×8 chunk of matrix.



Asynchrony

- Independent calculations
- Independent data transfers
- Overlapping data transfers and calculations



OpenACC `wait` directive

The `wait` directive causes the local thread to wait for completion of asynchronous operations, such as an accelerator parallel or kernels region or an update directive, or causes one device activity queue to synchronize with one or more other activity queues. The wait argument, if present, must be one or more async- arguments.

- Fortran:

```
!$acc wait [(int-expr-list)] clause-list
```

- C:

```
#pragma acc wait [(int-expr-list)] clause-list
```

OpenACC async clause

```
async [( int-expr )]
```

- The async clause may appear on:

parallel	kernels
enter data	exit data
update	wait

- In all cases, the async clause is optional:
  - when there is no async clause, the local thread will wait until the compute construct or data operations are complete before executing any of the code that follows, or, on the wait directive, until all operations on the appropriate asynchronous activity queues are complete.
  - When there is an async clause, the parallel or kernels region or data operations may be processed asynchronously while the local thread continues with the code following the construct or directive.

Asynchrony

```
int n=100;
float a[100], b[100], c[100], d[100], e[100];
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
    b[i]=(float)rand()/RAND_MAX;
}
#pragma acc enter data copyin (a,b) create(c)
#pragma acc parallel loop independent present (a,b) async(0)
for (int i=0; i<n; i++)
{
    c[i]=a[i]+b[i];
}
#pragma acc exit data copyout (c) async(0)
```

Asynchrony

```
for (int i=0; i<n; i++)
{
    d[i] = (float)rand() / RAND_MAX;
}
#pragma acc enter data copyin (d) create(e)
#pragma acc parallel loop independent present (a,b,d) async(1)
for (int i=0; i<n; i++)
{
    e[i]=a[i]-b[i]+d[i];
}
#pragma acc exit data copyout (e) async(1)
#pragma acc wait (0,1)
```

Asynchrony

```
$ pgc++ async_wait.cpp -acc -Minfo=accel -ta=nvidia,time -o async_wait
async_wait.cpp:
main:
13, Generating enter data create(c[:],b[:],a[:])
Generating present(a[:],b[:])
Accelerator kernel generated
Generating Tesla code
16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
13, Generating copyout(c[:])
19, Generating exit data copyout(c[:])
24, Generating enter data create(e[:],d[:])
Generating present(a[:],b[:],d[:])
Accelerator kernel generated
Generating Tesla code
28, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
24, Generating copyout(e[:])
31, Generating exit data copyout(e[:])
```

Asynchrony

```
$ ./async_wait
1.23457=0.840188+0.394383 1.42524=0.840188-0.394383+0.979434
1.58154=0.783099+0.79844 0.72847=0.783099-0.79844+0.743811
1.1092=0.911647+0.197551 1.61746=0.911647-0.197551+0.903366
1.10345=0.335223+0.76823 0.550589=0.335223-0.76823+0.983596
0.831745=0.277775+0.55397 0.390685=0.277775-0.55397+0.66688
1.10627=0.477397+0.628871 0.345785=0.477397-0.628871+0.497259
0.878185=0.364784+0.513401 0.0153516=0.364784-0.513401+0.163968
1.86842=0.95223+0.916195 0.866046=0.95223-0.916195+0.830012
1.35301=0.635712+0.717297 0.807364=0.635712-0.717297+0.888949
0.748571=0.141603+0.606969 -0.388372=0.141603-0.606969+0.0769947

Accelerator Kernel Timing data
    Timing may be affected by asynchronous behavior
    set PGI_ACC_SYNCHRONOUS to 1 to disable async() clauses
/scratch/aivahnenko/openacc/openacc2_0/async_wait/async_wait.cpp
```

Asynchrony

```
$ ./async_wait
1.23457=0.840188+0.394383 1.42524=0.840188-0.394383+0.979434
1.58154=0.783099+0.79844 0.72847=0.783099-0.79844+0.743811
1.1092=0.911647+0.197551 1.61746=0.911647-0.197551+0.903366
1.10345=0.335223+0.76823 0.550589=0.335223-0.76823+0.983596
0.831745=0.277775+0.55397 0.390685=0.277775-0.55397+0.66688
1.10627=0.477397+0.628871 0.345785=0.477397-0.628871+0.497259
0.878185=0.364784+0.513401 0.0153516=0.364784-0.513401+0.163968
1.86842=0.95223+0.916195 0.866046=0.95223-0.916195+0.830012
1.35301=0.635712+0.717297 0.807364=0.635712-0.717297+0.888949
0.748571=0.141603+0.606969 -0.388372=0.141603-0.606969+0.0769947

Accelerator Kernel Timing data
    Timing may be affected by asynchronous behavior
    set PGI_ACC_SYNCHRONOUS to 1 to disable async() clauses
/scratch/aivahnenko/openacc/openacc2_0/async_wait/async_wait.cpp
```

Atomic directive

- C:
  ```
  #pragma acc atomic [ atomic-clause ]
  ```

An atomic construct ensures that a specific storage location is accessed and/or updated atomically, preventing simultaneous reading and writing by gangs, workers and vector threads that could result in indeterminate values.

Atomic directive

- C:

  `#pragma acc atomic [ atomic-clause ]`

  `atomic-clause` is one of:

`read`

```
v = x;
```

`write`

```
x = expr;
```

`update`

```
x++; ++x;
x binop= expr;
x = expr binop x;
x--; --x;
x = x binop expr;
```

`capture`

```
v = x++; v = x--;
v = ++x; v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

Atomic directive: restrictions

The following restriction applies to the atomic construct:

- All atomic accesses to the storage locations designated by x throughout the program are required to have the same type and type parameters
- Storage locations designated by x must be less than or equal in size to the largest available native atomic operator width

Atomic directive: reduction example

```
int n=1000;
float a[1000];
float b=0;
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
}

#pragma acc parallel copy (b,a)
for (int i=0; i<n; i++)
{
    #pragma acc atomic update
    b+=a[i];
}
```

Atomic directive: reduction example

```
int n=1000;
float a[1000];
float b=0;
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
}

#pragma acc parallel copy (b,a)
for (int i=0; i<n; i++)
{
    #pragma acc atomic update
    b+=a[i];
}
```

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

Advanced code optimization practices — by examples

Dr. Aleksei Ivakhnenko

March 11, 2018

Tips and tricks

- Eliminate Pointer Arithmetic
- Privatize arrays
- Make while loops parallelizable
- Rectangles are better than triangles
- Restructure linearized arrays with computed indices
- Privatize live-out scalars
- Watch for runtime device errors

Eliminate Pointer Arithmetic

OpenACC can not parallelize loops with pointer arithmetic:

Wrong:

```
void my_memcpy(float* restrict A,
    float * restrict B, int count)
{
    float* ptrA = A;
    float* ptrB = B;
    #pragma acc parallel
    for (int i=0; i<count;++i)
    {
        *ptrA++ = *prtB++;
    }
    return;
}
```

Correct:

```
void my_memcpy(float *restrict A,
    float * restrict B, int count)
{
    #pragma acc parallel
    {
        for (int i=0; i<count;++i)
        {
            A[i] = B[i];
        }
    }
    return;
}
```

Eliminate Pointer Arithmetic

OpenACC can not parallelize loops with pointer arithmetic:

Wrong:

```
pgcc -acc -Minfo=accel -Mcuda=ptxinfo -ta=tesla:cc35 pointers↩
      .c -o pointers
my_memcpy:
      8, Loop carried scalar dependence for ptrA at line 9
         Loop carried scalar dependence for ptrB at line 9
         Accelerator kernel generated
          8, #pragma acc loop seq
      8, Accelerator scalar kernel generated
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function 'my_memcpy_8_gpu' ↩
     for 'sm_35'
ptxas info    : Function properties for my_memcpy_8_gpu
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill ↩
         loads
ptxas info    : Used 3 registers, 324 bytes cmem[0]
```

Correct:

```
pgcc -acc -Minfo=accel -Mcuda=ptxinfo -ta=tesla:cc35 pointers_acc.c -o↩
       pointers_acc
my_memcpy:
      5, Generating copyin(A[:count])
         Generating copyout(B[:count])
      7, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          7, #pragma acc loop gang, vector(128) /* blockIdx.x ↩
                threadIdx.x */
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function 'my_memcpy_7_gpu' for 'sm_35'
ptxas info    : Function properties for my_memcpy_7_gpu
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 10 registers, 344 bytes cmem[0]
```

Privatize Arrays

OpenACC requires to privatize arrays for the correct parallelization:

Wrong:

```
#pragma acc parallel
{
    for (int i=0; i<N;++i) {

        for (int j=0; j<M;++j) {
            for (int ii=0; ii<10;++ii) {
                tmp[ii] = ii;
            }
            sum=0;
            for (int ii=0; ii<10;++ii) {
                sum+=tmp[ii];
            }
            A[i][j] = sum;
        }
    }
}
```

Correct:

```
#pragma acc parallel
{
    for (int i=0; i<N;++i) {
        #pragma acc loop private (tmp[0:10])
        for (int j=0; j<M;++j) {
            for (int ii=0; ii<10;++ii) {
                tmp[ii] = ii;
            }
            sum=0;
            for (int ii=0; ii<10;++ii) {
                sum+=tmp[ii];
            }
            A[i][j] = sum;
        }
    }
}
```

Privatize Arrays

OpenACC requires to privatize arrays for the correct parallelization:

Wrong:

```
pgcc -acc -Minfo=accel -Mcuda=ptxinfo -ta=tesla:cc35 arrays.c -o arrays
main:
     13, Generating implicit copyout(A[:][:],tmp[:10])
     15, Parallelization would require privatization of array tmp[:i1+i2+10]
     16, Parallelization would require privatization of array tmp[:i1+i2+10]
         Accelerator kernel generated
...
```

Correct:

```
pgcc -acc -Minfo=accel -Mcuda=ptxinfo -ta=tesla:cc35 arrays_correct.c -o arrays_correct
main:
     14, Generating implicit copyout(A[:][:])
     16, Loop is parallelizable
     19, Loop is parallelizable
         Accelerator kernel generated
...
```

Make While Loops Parallelizable

OpenACC doesn't support while loops parallelization:

Wrong:

```
#pragma acc parallel
{
    while (i < N && found == -1)
    {
        if (A[i] >= 102.0f)
            found = i;
        ++i;
    }
}
```

Correct:

```
#pragma acc parallel
{
    for (i = 0; i < N; ++i) {
        if (A[i] >= 102.0f)
            found[i] = i;
        else
            found[i] = -1;
    }
}
i = 0;
while (i < N && found[i] < 0) {
    ++i;
}
```

Make While Loops Parallelizable

OpenACC doesn't support while loops parallelization:

Wrong:

```
$ pgcc -Minfo=accel -ta=nvidia:4.0 while.c
my_while:
...
    Accelerator restriction: size of the GPU copy of an array depends on values computed in this loop
    Accelerator restriction: loop has multiple exits
    Accelerator restriction: loop contains multi-level induction variable
...
```

Correct:

```
pgcc -acc -Minfo=accel -Mcuda=ptxinfo -ta=tesla:cc35 arrays_correct.c -o arrays_correct
my_while:
...
    9, Loop is parallelizable
       Accelerator kernel generated
    9, #pragma acc for parallel, vector(256) /* blockIdx.x threadIdx.x */
...
```

Rectangles Are Better Than Triangles

Do not use triangular loops. Here the lower triangle of data will be replaced with garbage from the GPU:

Wrong:

```
#pragma acc parallel copyout(A[0:N][0:M])
{
    for (int i=0; i<N;++i) {
        for (int j=i; j<M;++j) {
            A[i][j] = i+j;
        }
    }
}
```

Correct:

```
#pragma acc parallel copy(A[0:N][0:M])
{
    for (int i=0; i<N;++i) {
        for (int j=0; j<M;++j) {
            A[i][j] = i+j;
        }
    }
}
```

Restructure Linearized Arrays With Computed Indices

Complex index dependencies prevent loop parallelization:

Wrong:

```
#pragma acc parallel copyout(A[0:N*M])
{
    for (int i=0; i<N;++i) {
        for (int j=0; j<M;++j) {
            idx = (i*N)+j;
            A[idx] = B[i][j];
        }
    }
}
```

Correct:

```
#pragma acc parallel copyout(A[0:N][0:M])
{
    for (int i=0; i<N;++i) {
        for (int j=0; j<M;++j) {

            A[i][j] = B[i][j];
        }
    }
}
```

Restructure Linearized Arrays With Computed Indices

Complex index dependencies prevent loop parallelization:

Wrong:

```
$ pgcc -Minfo=accel -ta=nvidia:4.0 restruct.c
main:
...
    15, Parallelization would require privatization of array 'A[:M*N-1]'
    16, Parallelization would require privatization of array 'A[:M*N-1]'
        Accelerator kernel generated
    15, #pragma acc for seq
    16, #pragma acc for seq
        Non-stride-1 accesses for array 'B'
...
```

Correct:

```
$ pgcc -Minfo=accel -ta=nvidia:4.0 restruct.c
main:
...
    15, Loop is parallelizable
    16, Loop is parallelizable
        Accelerator kernel generated
    15, #pragma acc for parallel, vector(16) /* blockIdx.y threadIdx.y */
    16, #pragma acc for parallel, vector(16) /* blockIdx.x threadIdx.x */
...
```

Privatize Live-out Scalars

Live-out scalars prevent parallelization, do not use them or restructure the code. The second option gives wrong value in idx output:

Wrong:

```
#pragma acc parallel
{

    for (int i=0; i<N;++i) {
        for (int j=0; j<M;++j) {
            idx = i+j;
            A[i][j] = idx;
        }
    }
}
printf("%d %d %d\n",idx, A[1][1], A[2][1]);
```

Correct (*printf* is still wrong):

```
#pragma acc parallel
{
    #pragma acc loop private(idx)
    for (int i=0; i<N;++i) {
        for (int j=0; j<M;++j) {
            idx = i+j;
            A[i][j] = idx;
        }
    }
}
printf("%d %d %d\n",idx, A[1][1], A[2][1]);
```

Privatize Live-out Scalars

Live-out scalars prevent parallelization, do not use them or restructure the code. The second option gives wrong value in idx output:

Wrong:

```
$ pgcc -Minfo=accel -ta=nvidia:4.0 live-out.c
main:
...
    16, Accelerator restriction: induction variable live-out from loop: idx
    17, Accelerator restriction: induction variable live-out from loop: idx
...
```

Correct:

```
$ pgcc -Minfo=accel -ta=nvidia:4.0 live-out.c
main:
...
    15, #pragma acc for parallel, vector(16) /* blockIdx.y threadIdx.y */
    16, #pragma acc for parallel, vector(16) /* blockIdx.x threadIdx.x */
...
```

Watch out for Runtime Device Errors

Even if the code has been successfully compiled, there still could be runtime errors:

Call to cuMemcpyDtoH returned error 700:
Launch failed

```
#pragma acc parallel copyin(B[0:N][0:M])
{
    for (int i=0; i<N;++i) {
        for (int j=0; j<M;++j) {
            A[i][j] = B[i][j+1];
        }
    }
}
```

Call to cuMemcpy2D returned error 1:
Invalid value

```
#pragma acc parallel copyin(B[0:N][0:M+1])
{
    for (int i=0; i<N;++i) {
        for (int j=0; j<M;++j) {
            A[i][j] = B[i][j];
        }
    }
}
```