PARALLEL PROGRAMMING

GPU PROGRAMMING LAB ASSIGNMENT

AUTHORS: Lawrence Adu-Gyamfi (1484610)
Maximilian Grunwald (1519529)

Date : 22/01/2019

Table of Contents

1. Introduction

The purpose of this report is to present the code modifications together with the execution results for 2 applications run on CPU and GPU. The applications are:

a. Solving a linear system of equations using the Jacobi Method
b. Heat Transfer

We present as well, a detailed analysis of the performance of these applications and where applicable, we present improvements performed on these applications and the consequent results.

2. Solving a linear system of equations using the Jacobi Method

This program uses the Jacobi Iterative method to solve a system of linear equations. The code is written in C++ and the final code was provided as part of the assignment problem.

2.1 Source Code – OpenACC Constructs

Below is a snippet of the code of the program highlighting the Open ACC pragmas that have been included in the code to implement parallelism and for execution on a CPU and GPU.

```
#pragma acc data copyin(A[:nsize*nsize],b[:nsize],diag[:nsize]) copy(xnew[:nsize],xold[:nsize])
{
while ((residual > TOLERANCE*TOLERANCE) && (iters < max_iters)) {
  ++iters;
  if (iters & 1 == 1)
  {
    residual = 0.0;
    #pragma acc kernels loop independent
    for (i = 0; i < nsize; ++i) {
      TYPE rsum = (TYPE)0;
      for (j = 0; j < nsize; ++j) {
        rsum += A[i*nsize + j] * xold[j];
      }
      xnew[i] = (b[i] - rsum) / diag[i];
      TYPE dif = xnew[i] - xold[i];
      residual += dif * dif;
    }
  }
  else {
    residual = 0.0;
    #pragma acc kernels loop independent
    for (i = 0; i < nsize; ++i) {
      TYPE rsum = (TYPE)0;
      for (j = 0; j < nsize; ++j) {
        rsum += A[i*nsize + j] * xnew[j];
      }
      xold[i] = (b[i] - rsum) / diag[i];
      TYPE dif = xnew[i] - xold[i];
      residual += dif * dif;
    }
  }
```

Figure 2-1: Base Code Jsolve4.cpp

2.2 CPU Execution Results

The following is the output after compiling with the CPU (for both n=1000 and 4000) indicating that all loops have parallelized as requested in the code with the OpenACC constructs. For the sections of the code where the residuals are summed up the compiler generates the code for reduction as well.

Compiling for CPU
main:
110, Loop is parallelizable
Generating Multicore code
110, #pragma acc loop gang
112, Loop is parallelizable
117, Generating implicit reduction(+:residual)
123, Loop is parallelizable
Generating Multicore code
123, #pragma acc loop gang
125, Loop is parallelizable
130, Generating implicit reduction(+:residual)

2.2.1 (Nsize, Max_Iterations) = (1000,5000)

Below are the perf outputs when the code was executed on CPU with a problem size of 1000 and for 5000 iterations.
The execution time recorded was 0.5 sec, which indicates that probably this size of problem is not good enough to make constructive performance analysis considering the version of the code used is the highly optimized version.
We can confirm again that the program has been parallelized by the number of CPUs utilized for the execution which is close to 4 CPUs.
A look at the IPC for the cores indicates a value of 1.49 which is not the best even if we know it is difficult to achieve values higher than 3 in this case. This can be a sign of a possible performance bottleneck, most likely due to the bandwidth of the slow DRAM. This will be verified even further when we increase the nsize.

```
Converged after 4448 iterations and 0.452831 seconds, residual is 0.000998887
Solution error is 0.000997877

 Performance counter stats for './jCPU4 1000 5000':

      1827.473638      task-clock (msec)         #    3.642 CPUs utilized
                30      context-switches          #    0.016 K/sec
                 2      cpu-migrations            #    0.001 K/sec
             1,262      page-faults               #    0.691 K/sec
     6,109,952,858      cycles                    #    3.343 GHz
     3,321,474,163      stalled-cycles-frontend   #   54.36% frontend cycles idle
   <not supported>      stalled-cycles-backend
     9,098,888,668      instructions              #    1.49  insns per cycle
                                                  #    0.37  stalled cycles per insn
       866,582,182      branches                  #  474.197 M/sec
         4,628,294      branch-misses             #    0.53% of all branches

       0.501764361 seconds time elapsed
```

Figure 2-2: CPU Results (Nsize, Max_Iterations) = (1000,5000)

An extra run was done to check the cache-misses; a snip of this shown below as well. We noticed a high amount of cache misses which is understandable based on the datatype being used to store the values (double) and the size of the matrix(nsize*nsize).

```
Converged after 4448 iterations and 0.455247 seconds, residual is 0.000998887
Solution error is 0.000997877

 Performance counter stats for './jCPU4 1000 10000':

       25,793,229      cache-misses              #   14.038 M/sec
    9,182,717,481      instructions              #    1.48  insns per cycle
    6,211,079,619      cpu-cycles                #    3.380 GHz
      1837.439214      cpu-clock (msec)
      1837.439126      task-clock (msec)         #    3.664 CPUs utilized

      0.501545957 seconds time elapsed
```

Figure 2-3: CPU Results (2) (Nsize, Max_Iterations) = (1000,5000)

2.2.2 (Nsize, Max_Iterations) = (4000,20000)

Below is the result generated after executing the code on the CPU using a problem size of 4000 and running it for 20000 iterations.

We can confirm here as well that the problem has been executed utilising parallel threads or cores in the CPU by the value for the CPUs utilized (3.995).
But what is immediately evident is the time for execution (127.3 sec) which is 254x the results for the execution with nsize=1000. We do know that the program has a complexity of $O(n^2)$ because of the "double for-loop" and so an increase in 4x of problem size should result in at least 16x the execution time.
Also, we can see that the convergence happens after 19212 iterations which is about 4x the number for the problem size of 1000. Yet this is still not consistent with the increase in execution times we noticed.

A look at the results of the computation throughput gives us an idea what could be leading to this unexplained increase in execution time. The IPC seems to have gone down to 0.31 which is way below the average expected and that for the nsize of 1000. This is probably the effect of the increase in problem size resulting in a performance bottleneck from the bandwidth of the DRAM.

```
Converged after 19212 iterations and 127.341 seconds, residual is 0.000999766
Solution error is 0.000999515

 Performance counter stats for './jCPU4 4000 20000':

     510254.089366      task-clock (msec)         #    3.995 CPUs utilized
              827      context-switches          #    0.002 K/sec
                8      cpu-migrations            #    0.000 K/sec
           14,232      page-faults               #    0.028 K/sec
1,882,512,588,294      cycles                    #    3.689 GHz
1,694,744,710,773      stalled-cycles-frontend   #   90.03% frontend cycles idle
  <not supported>      stalled-cycles-backend
  582,724,551,104      instructions              #    0.31  insns per cycle
                                                 #    2.91  stalled cycles per insn
   51,087,063,025      branches                  #  100.121 M/sec
       78,779,961      branch-misses             #    0.15% of all branches

     127.734114972 seconds time elapsed
```

Figure 2-4: CPU Results (1) (Nsize, Max_Iterations) = (4000,20000)

Another run to profile the cache-misses shows an immense increase (620x) in the cache misses, which confirms our earlier assumption for the possible source of performance limitation. This result is shown below as well.

```
Converged after 19212 iterations and 127.291 seconds, residual is 0.000999766
Solution error is 0.000999515

 Performance counter stats for './jCPU4 4000 20000':

    16,063,385,119      cache-misses              #   31.517 M/sec
   571,057,642,708      instructions              #    0.30  insns per cycle
 1,880,441,628,775      cpu-cycles                #    3.689 GHz
     509678.984353      cpu-clock (msec)
     509678.999994      task-clock (msec)         #    3.995 CPUs utilized

     127.580768966 seconds time elapsed
```

Figure 2-5: CPU Results (2) (Nsize, Max_Iterations) = (4000,20000)

2.3 GPU Execution Results

Below is the output after compiling for the GPU.

Compiling for GPU
main:
103, Generating copyin(A[:nsize*nsize],b[:nsize],diag[:nsize])
Generating copy(xold[:nsize],xnew[:nsize])
110, Loop is parallelizable
Accelerator kernel generated
Generating Tesla code
110, #pragma acc loop gang /* blockIdx.x */
112, #pragma acc loop vector(128) /* threadIdx.x */
113, Generating implicit reduction(+:rsum)
117, Generating implicit reduction(+:residual)
112, Loop is parallelizable
123, Loop is parallelizable
Accelerator kernel generated
Generating Tesla code
123, #pragma acc loop gang /* blockIdx.x */
125, #pragma acc loop vector(128) /* threadIdx.x */
126, Generating implicit reduction(+:rsum)
130, Generating implicit reduction(+:residual)
125, Loop is parallelizable

This output indicates that the code for the 2 loops has been parallelized. Also, the compiler generates the code for reduction for the computation of “rsum” and the residuals. Also, we can tell that this computation is performed in SIMD mode because a single gang of 128 workers has been dedicated by the compiler for both reduction tasks.
This is confirmed by the chart below obtained from the nvprof graphical viewer.

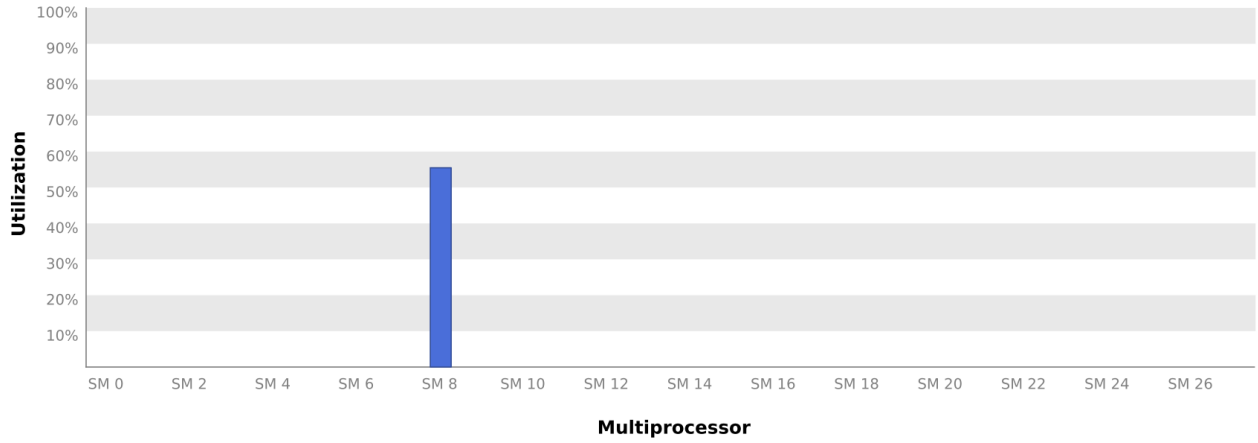

Figure 2-6: Multiprocessor Utilization for Reduction Kernel

We can also see the data movement codes generated by the compiler to copy the matrix and the vectors between the host and device.

2.3.1 (Nsize, Max_Iterations) = (1000,5000)

The profiling results of running the application on the GPU for a problem size of 1000 is shown below and it completes the solution in 1.4 sec which is more than the 0.5 secs produced by the CPU version of the execution of the application.
However, we notice that close to 90% of the execution time is spent on main double loops in the application. About 9% is spent on the operations for the reduction while the remaining is spent on data movement.

```
Converged after 4448 iterations and 1.41702 seconds, residual is 0.000998887
Solution error is 0.000997877
Profiling application: ./jGPU4 1000 5000
==11367== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   44.59%  409.25ms      2224  184.02us  38.657us  459.09us  main_123_gpu
                   44.53%  408.62ms      2224  183.73us  39.265us  485.56us  main_110_gpu
                    4.96%  45.474ms      2224  20.446us  3.9040us  31.745us  main_117_gpu__red
                    4.94%  45.376ms      2224  20.402us  4.0320us  31.682us  main_130_gpu__red
                    0.51%  4.6377ms      4450  1.0420us     768ns  6.2730us  [CUDA memcpy DtoH]
                    0.48%  4.3614ms      4453     979ns     640ns  654.46us  [CUDA memcpy HtoD]
```

Figure 2-7: GPU Results (1) (Nsize, Max_Iterations) = (1000,5000)

The results of a few of the most interesting metrics after running the code on the GPU are discussed below.

```
==11523== Profiling result:
==11523== Metric result:
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 Ti (0)"
    Kernel: main_110_gpu
          5                        l2_read_throughput                                L2 Throughput (Reads)  373.45GB/s  377.23GB/s  375.03GB/s
          5                       l2_write_throughput                               L2 Throughput (Writes)  3.7169GB/s  3.8076GB/s  3.7434GB/s
          5                             inst_executed                             Instructions Executed     1387000     1387798     1387159
          5                             sm_efficiency                         Multiprocessor Activity      83.80%      88.76%      87.34%
          5                                       ipc                                    Executed IPC    0.817888    0.958877    0.922352
          5                                issued_ipc                                      Issued IPC    0.921300    1.573164    1.068320
          5                             l2_utilization                          L2 Cache Utilization     Low (2)     Low (2)     Low (2)
          5          double_precision_fu_utilization          Double-Precision Function Unit Utilization     Low (3)     Mid (4)     Low (3)
          5                   dram_write_transactions                   Device Memory Write Transactions       39555       39663       39612
          5                      dram_read_throughput                      Device Memory Read Throughput  186.06GB/s  188.00GB/s  186.92GB/s
          5                     dram_write_throughput                     Device Memory Write Throughput  29.405GB/s  29.625GB/s  29.491GB/s
          5                          dram_utilization                          Device Memory Utilization     Mid (5)     Mid (5)     Mid (5)
```

Figure 2-8: GPU Results (2) (Nsize, Max_Iterations) = (1000,5000)

We observe that the Multiprocessor activity which signifies the combined efficiency of utilization of the available SMs is close to 90% which means the parallelized application makes good use of the resources of the GPU.

We notice however that the computation throughput is just shy of 1 which is about 18% of the peak available throughput of 6 for the GeForce GTX 1080 Ti.
The L2 cache utilization is low (2) so it is not entirely clear if this is the source of performance bottleneck. We do notice a mid-value of 5 for the device memory utilization which is not a major source of concern as well at this point. So, we can safely assume at this stage that the memory bandwidth is not a performance bottleneck. A look at the computation metrics reveal a mid-value of 4 for double precision function unit utilization as well as low value values for the other function units' utilization. So, we can assume as well that our application is not being limited by the computation resources.

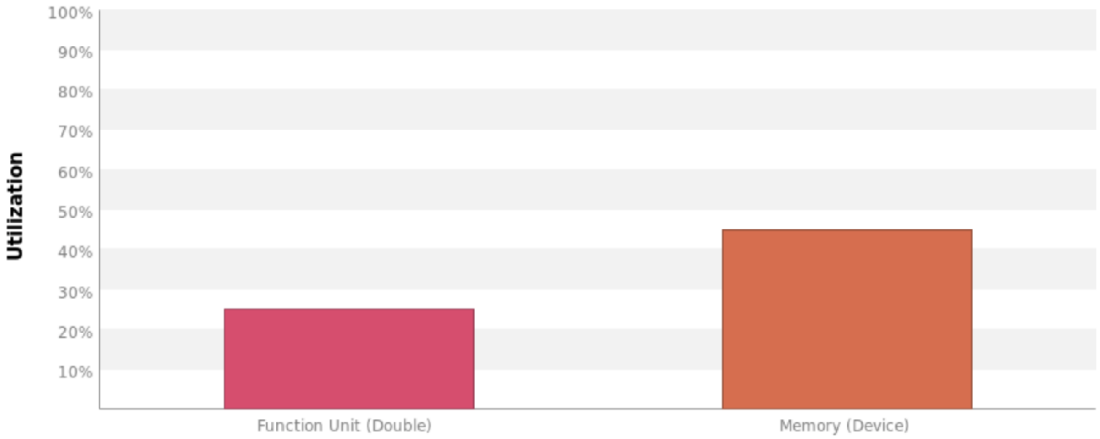

Figure 2-9: Compute and Memory Utilization (Nsize, Max_Iterations) = (1000,5000)

This is most likely due to the size of the problem. This is eventually confirmed by the output of the performance analyser of the nvprof tool.

In the next section, the problem size is increased, and these metrics are evaluated again to confirm if indeed this will have an impact on the performance and expose the real bottlenecks.

2.3.2 (Nsize, Max_Iterations) = (4000,20000)

Next, we run the application for a problem size of 4000 and max iterations of 20000. Below are the results obtained for the execution.

```
Converged after 19212 iterations and 8.6915 seconds, residual is 0.000999766
Solution error is 0.000999515
==11131== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   48.91%  3.81870s      9606  397.53us  333.81us  4.2015ms  main_110_gpu
                   48.69%  3.80116s      9606  395.71us  332.78us  4.1945ms  main_123_gpu
                    0.96%  75.190ms      9606  7.8270us  5.9520us  61.090us  main_117_gpu__red
                    0.95%  73.957ms      9606  7.6980us  5.8880us  60.738us  main_130_gpu__red
                    0.32%  24.775ms     19224  1.2880us     512ns  1.9457ms  [CUDA memcpy HtoD]
                    0.17%  13.289ms     19214     691ns     608ns  5.6640us  [CUDA memcpy DtoH]
```

Figure 2-10: GPU Results (1) (Nsize, Max_Iterations) = (4000,20000)

Firstly, we notice an increase of about 6x in the execution time for the execution with a size of 4000 with convergence happening after 19212 iterations which is consistent with the results of the CPU execution.

We present further analysis of the execution below focusing mainly on the metrics which are different from the run of the application with a size of 1000 (previous case).

```
Profiling application: ./jGPU4 4000 10
==11258== Profiling result:
==11258== Metric result:
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 Ti (0)"
    Kernel: main_110_gpu
          5                        l2_read_throughput                       L2 Throughput (Reads)  626.90GB/s  634.92GB/s  631.00GB/s
          5                       l2_write_throughput                      L2 Throughput (Writes)  1.5669GB/s  1.5854GB/s  1.5794GB/s
          5                             inst_executed                        Instructions Executed    14968000    14970520    14968504
          5                             sm_efficiency                   Multiprocessor Activity      96.96%      98.28%      97.61%
          5                                       ipc                              Executed IPC    0.964229    1.969510    1.167775
          5                                issued_ipc                               Issued IPC    0.961134    1.974401    1.169414
          5                          l2_utilization                       L2 Cache Utilization     Low (3)     Mid (4)     Low (3)
          5          double_precision_fu_utilization  Double-Precision Function Unit Utilization     Low (3)    High (7)     Mid (4)
          5                    dram_read_throughput                 Device Memory Read Throughput  313.69GB/s  317.42GB/s  315.64GB/s
          5                   dram_write_throughput                Device Memory Write Throughput  3.2347GB/s  3.2740GB/s  3.2549GB/s
          5                         dram_utilization                     Device Memory Utilization    High (8)    High (8)    High (8)
```

Figure 2-11: GPU Results (2) (Nsize, Max_Iterations) = (4000,20000)

Interestingly the multiprocessor capacity is being utilized largely by the application up to about 98% which is even better.

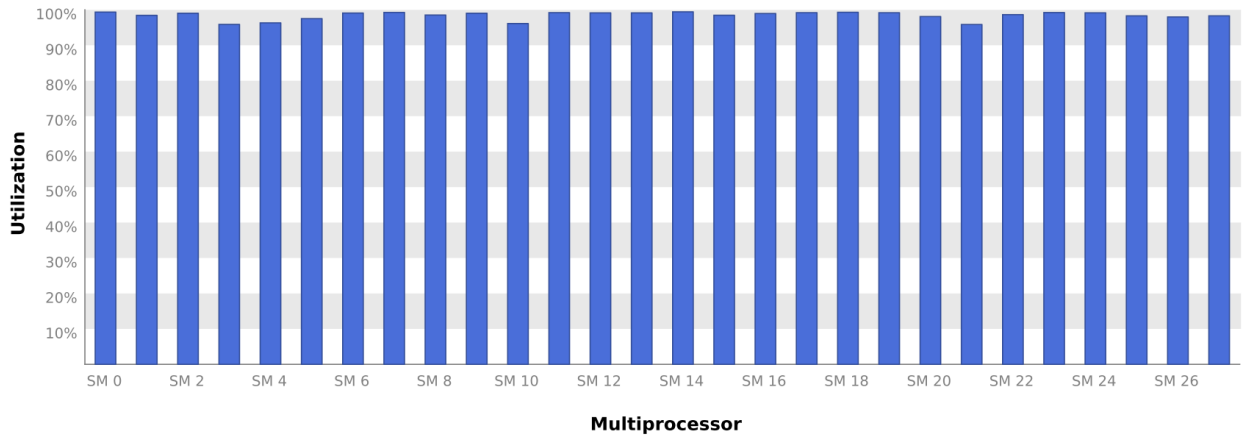

Figure 2-12: Multiprocessor Utilization (Nsize, Max_Iterations) = (4000,20000)

We do notice as well an improvement in the computation throughput of the multiprocessors to almost 2, which is about 30% of the peak capacity of the GPU.

A very interesting observation in the performance results is an immense increase in the device memory utilization and the L2 throughput. We are using about 30-40% of the peak bandwidth of L2 cache memory as well as a high value of 80% for the device memory throughput.
This is confirmed by the chart below which shows that the application is using close 80% of the peak bandwidth of the device memory.

At this stage we can conclude that the performance of our application is limited by the memory bandwidth of the device.

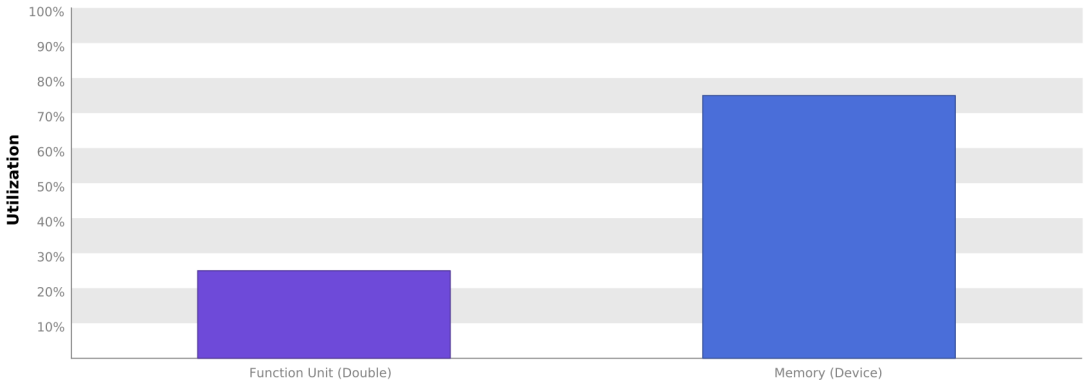

Figure 2-13: Compute and Memory Utilization (Nsize, Max_Iterations) = (4000,20000)

3. HEAT TRANSFER

3.1 CODE MODIFICATION FOR GPU ACCELERATION

Below is a snippet of the code of the application modified to include some OpenACC constructs required for compiling and executing the program in parallel on CPU and GPU. The full code is attached separately as part of submission of this report.

```
L2 = (1.0f-2.0f*L);
#pragma acc data copy(U1[:N+2], U2[:N+2])
{
for (t=1; t<=T; t++)  // loop on time
  if ((t&1) == 1) // t is odd
  {
    #pragma acc kernels loop independent
    for (x=1; x<=N; x++)  // loop on 1D bar
      U2[x] = L2*U1[x] + L*U1[x+1] + L*U1[x-1];
}
  else            // t is even
  {
    #pragma acc kernels loop independent
    for (x=1; x<=N; x++)  // loop on 1D bar
      U1[x] = L2*U2[x] + L*U2[x+1] + L*U2[x-1];
}
}
```

Figure 3-1: GPU-Accelerated Code for Heat Transfer Application

3.2 CPU EXECUTION Result

Below is the output after compiling the GPU-accelerated version of the application for the CPU. We notice the compiler acknowledges the OpenACC constructs requesting for parallelizing the nested loops and generated the required code for this parallelization. This is presented once as it is the similar for all problem cases run.

Compiling for CPU
main:
61, Loop is parallelizable
Generating Multicore code
61, #pragma acc loop gang
67, Loop is parallelizable
Generating Multicore code
67, #pragma acc loop gang

Below are the results for running the application on the CPU for different problem sizes while maintaining the number of time steps at 10000. The problem sizes have been increased in multiples of to from 10^6 to 10^8.

```
Running and profiling on CPU
Stencil computation of 10000 steps on 1-D vector of 1000000 elements with L=1.2339999666e-03

CheckSum = 5.5842300415e+01

 Performance counter stats for './hCPU 10000 1000000':

       9922.551369      task-clock (msec)         #    3.947 CPUs utilized
                51      context-switches          #    0.005 K/sec
                 1      cpu-migrations            #    0.000 K/sec
             1,628      page-faults               #    0.164 K/sec
    35,795,312,875      cycles                    #    3.607 GHz
    18,006,254,008      stalled-cycles-frontend   #   50.30% frontend cycles idle
  <not supported>       stalled-cycles-backend
    71,170,310,294      instructions              #    1.99  insns per cycle
                                                  #    0.25  stalled cycles per insn
     2,037,465,816      branches                  #  205.337 M/sec
           267,880      branch-misses             #    0.01% of all branches

       2.513638981 seconds time elapsed
```

Figure 3-2: CPU Results Vector Size = 10^6

```
Running and profiling on CPU
Stencil computation of 10000 steps on 1-D vector of 10000000 elements with L=1.2339999666e-03

CheckSum = 5.5842300415e+01

 Performance counter stats for './hCPU 10000 10000000':

     280883.092523      task-clock (msec)         #    3.993 CPUs utilized
              488      context-switches          #    0.002 K/sec
                6      cpu-migrations            #    0.000 K/sec
            1,832      page-faults               #    0.007 K/sec
1,035,478,985,160      cycles                    #    3.687 GHz
  858,874,292,606      stalled-cycles-frontend   #   82.94% frontend cycles idle
  <not supported>      stalled-cycles-backend
  704,639,199,991      instructions              #    0.68  insns per cycle
                                                 #    1.22  stalled cycles per insn
   18,071,361,817      branches                  #   64.338 M/sec
        1,464,920      branch-misses             #    0.01% of all branches

      70.351540159 seconds time elapsed
```

Figure 3-3: CPU Results Vector Size = 10^7

```
Running and profiling on CPU
Stencil computation of 10000 steps on 1-D vector of 100000000 elements with L=1.2339999666e-03

CheckSum = 5.5842300415e+01

 Performance counter stats for './hCPU 10000 100000000':

    2815112.315680      task-clock (msec)         #    4.001 CPUs utilized
            4,268      context-switches          #    0.002 K/sec
                8      cpu-migrations            #    0.000 K/sec
            2,343      page-faults               #    0.001 K/sec
10,278,529,040,223      cycles                    #    3.651 GHz
 8,536,479,622,155      stalled-cycles-frontend   #   83.05% frontend cycles idle
   <not supported>      stalled-cycles-backend
 6,923,446,639,490      instructions              #    0.67  insns per cycle
                                                  #    1.23  stalled cycles per insn
   139,784,361,797      branches                  #   49.655 M/sec
        14,590,013      branch-misses             #    0.01% of all branches

     703.655304177 seconds time elapsed
```

Figure 3-4: CPU Results Vector Size = 10^8

We further confirm from the values of the CPUs utilized in outputs that the application has indeed been executed in parallel which is close to 4 (The available cores on the CPU execution).
We do notice as well an increase in multiples of 10 for the number of instructions generated between the different runs, and this is consistent with the problem sizes.
It is immediately clear the difference in the execution times of each runs of the application. The major inconsistency is between the execution for 10^6 and 10^7 problem sizes. We expected a linear increase in the time of execution as per the problem size. However, we see a jump from 2 sec to about 70 sec. It is more consistent however for the next problem size where the time of execution increased by a multiple of 10 to 700 sec.
We do notice though that the instructions throughput reduced drastically from about 2 IPCs to 0.67 after the increase in the problem size (from 10^6 to 10^7). This is what causes the main inconsistency for the execution times for the 10^6 and 10^7 problems. We notice however for the 10^7 and 10^8 problems which have about the same IPC, the time change is consistent with the change in problem size.
This is an indication that the possible bottleneck in the performance of our application might be the memory bandwidth. With the increase in the problem sizes there is probably more read and writes in the slow DRAM of the CPU which in turn increases the latency resulting in the drop in the computation throughput.

3.3 GPU Execution

We present below the compiler output for compiling the code of our application for the GPU. This is presented once since it is the same for all problem sizes.
We do notice the data movements between the host and the device as well as code generation for running the application in parallel on the GPU. For the 2 major **"for loops"** where the majority of our computation will take place, we notice the compiler generates code to launch multiple multiprocessors (SMs) and proceeds to make the computation with SIMD operations. This, as we will confirm from the analysis of the execution should lead to maximum utilization of the multiprocessors in the GPU.

Compiling for GPU
main:
55, Generating copy(U2[:N+2],U1[:N+2])
61, Loop is parallelizable
Accelerator kernel generated
Generating Tesla code
61, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
67, Loop is parallelizable
Accelerator kernel generated
Generating Tesla code
67, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */

Below are the results after execution of the application on the GPU for three (3) separate problems, as done in the case of the CPU. (10^6, 10^7 and 10^8).

The main observations are discussed below.

```
Running on GPU
Stencil computation of 10000 steps on 1-D vector of 1000000 elements with L=1.2339999666e-03

CheckSum = 5.5842308044e+01

real    0m2.289s
user    0m1.022s
sys     0m1.146s

Running and profiling on GPU
==16991== NVPROF is profiling process 16991, command: ./hGPU 10000 1000000
Stencil computation of 10000 steps on 1-D vector of 1000000 elements with L=1.2339999666e-03

CheckSum = 5.5842308044e+01
==16991== Profiling application: ./hGPU 10000 1000000
==16991== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.96%  569.19ms      5000  113.84us  66.147us  352.17us  main_61_gpu
                   49.88%  568.26ms      5000  113.65us  66.114us  361.49us  main_67_gpu
                    0.11%  1.1985ms         2  599.24us  598.68us  599.80us  [CUDA memcpy HtoD]
                    0.06%  631.54us         2  315.77us  315.66us  315.88us  [CUDA memcpy DtoH]
```

Figure 3-5: GPU Results Vector Size = 10^6 – (1)

```
Running on GPU
Stencil computation of 10000 steps on 1-D vector of 10000000 elements with L=1.2339999666e-03

CheckSum = 5.5842308044e+01

real    0m4.552s
user    0m2.723s
sys     0m1.628s

Running and profiling on GPU
==17953== NVPROF is profiling process 17953, command: ./hGPU 10000 10000000
==17953== Profiling application: ./hGPU 10000 10000000
Stencil computation of 10000 steps on 1-D vector of 10000000 elements with L=1.2339999666e-03

CheckSum = 5.5842308044e+01
==17953== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.78%  1.55514s      5000  311.03us  210.06us  1.6460ms  main_61_gpu
                   49.77%  1.55465s      5000  310.93us  210.22us  1.6448ms  main_67_gpu
                    0.25%  7.8544ms         6  1.3091ms  708.48us  1.8500ms  [CUDA memcpy HtoD]
                    0.20%  6.2866ms         6  1.0478ms  506.71us  1.3186ms  [CUDA memcpy DtoH]
```

Figure 3-6: GPU Results Vector Size = 10^7 – (1)

```
Running on GPU
Stencil computation of 10000 steps on 1-D vector of 100000000 elements with L=1.2339999666e-03

CheckSum = 5.5842308044e+01

real    0m25.005s
user    0m18.888s
sys     0m5.794s

Running and profiling on GPU
==19091== NVPROF is profiling process 19091, command: ./hGPU 10000 100000000
Stencil computation of 10000 steps on 1-D vector of 100000000 elements with L=1.2339999666e-03

CheckSum = 5.5842308044e+01
==19091== Profiling application: ./hGPU 10000 100000000
==19091== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.78%  11.4459s      5000  2.2892ms  2.1919ms  10.675ms  main_61_gpu
                   49.55%  11.3927s      5000  2.2785ms  2.1805ms  10.665ms  main_67_gpu
                    0.40%  91.775ms        48  1.9120ms  1.3509ms  2.0431ms  [CUDA memcpy HtoD]
                    0.27%  62.911ms        48  1.3106ms  1.1103ms  1.3335ms  [CUDA memcpy DtoH]
```

Figure 3-7: GPU Results Vector Size = 10^8 – (1)

We confirm from the results that majority of the GPU time (over 99%) is spent in the main kernels (for loops) of the application. The most interesting detail captured by these results is the evolution of the execution time as the problem size increases especially in these main kernels.
We observe an increase of about 3x in the time spent running these kernels between the 10^6 and 10^7 problem size. However, between the 10^7 and 10^8 increase we notice close to about 9x the time required for execution.
We originally have an idea from the CPU execution that the performance of our application could be limited by the memory bandwidth. We confirm this as well from the profiling of the execution of the application shown below.

```
Profiling application: ./hGPU 20 1000000
==17087== Profiling result:
==17087== Metric result:
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 Ti (0)"
    Kernel: main_61_gpu
          10                        l2_read_throughput                       L2 Throughput (Reads)  532.99GB/s  537.80GB/s  534.91GB/s
          10                       l2_write_throughput                      L2 Throughput (Writes)  190.28GB/s  191.61GB/s  190.87GB/s
          10                             inst_executed                        Instructions Executed     1218782     1218782     1218782
          10                             sm_efficiency                   Multiprocessor Activity      88.38%      90.73%      89.17%
          10                                       ipc                              Executed IPC    1.280329    2.640228    1.447213
          10                            l2_utilization                      L2 Cache Utilization     Mid (4)     High (7)     Mid (4)
          10                     dram_read_throughput                Device Memory Read Throughput  152.71GB/s  155.95GB/s  153.28GB/s
          10                    dram_write_throughput               Device Memory Write Throughput  151.36GB/s  152.55GB/s  151.80GB/s
          10                         dram_utilization                    Device Memory Utilization    High (7)    High (7)    High (7)
```

Figure 3-8: GPU Results Vector Size = 10^6 – (2)

```
==18054== Profiling application: ./hGPU 20 10000000
Stencil computation of 20 steps on 1-D vector of 10000000 elements with L=1.2339999666e-03

CheckSum = 2.1243949890e+01
==18054== Profiling result:
==18054== Metric result:
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 Ti (0)"
    Kernel: main_61_gpu
          10                        l2_read_throughput                       L2 Throughput (Reads)  608.25GB/s  611.20GB/s  609.40GB/s
          10                       l2_write_throughput                      L2 Throughput (Writes)  217.14GB/s  217.87GB/s  217.54GB/s
          10                             inst_executed                        Instructions Executed    13478860    13478860    13478860
          10                             sm_efficiency                   Multiprocessor Activity      98.66%      99.17%      98.74%
          10                                       ipc                              Executed IPC    1.525457    3.017756    1.677850
          10                            l2_utilization                      L2 Cache Utilization     Mid (4)     Mid (4)     Mid (4)
          10                     dram_read_throughput                Device Memory Read Throughput  173.90GB/s  174.50GB/s  174.22GB/s
          10                    dram_write_throughput               Device Memory Write Throughput  173.65GB/s  174.24GB/s  173.99GB/s
          10                         dram_utilization                    Device Memory Utilization    High (8)    High (8)    High (8)
```

Figure 3-9: GPU Results Vector Size = 10^7 – (2)

```
==19170== Profiling application: ./hGPU 10 100000000
Stencil computation of 10 steps on 1-D vector of 100000000 elements with L=1.2339999666e-03

CheckSum = 2.1122716904e+01
==19170== Profiling result:
==19170== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 Ti (0)"
    Kernel: main_61_gpu
          5                        l2_read_throughput                       L2 Throughput (Reads)  592.79GB/s  592.95GB/s  592.88GB/s
          5                       l2_write_throughput                      L2 Throughput (Writes)  211.65GB/s  211.71GB/s  211.69GB/s
          5                             inst_executed                        Instructions Executed    99137560    99137560    99137560
          5                           sm_efficiency                   Multiprocessor Activity      99.86%      99.88%      99.87%
          5                                       ipc                              Executed IPC    0.857493    0.858132    0.857770
          5                            l2_utilization                       L2 Cache Utilization    Low (3)     Low (3)     Low (3)
          5                     dram_read_throughput                Device Memory Read Throughput  169.53GB/s  169.58GB/s  169.56GB/s
          5                    dram_write_throughput               Device Memory Write Throughput  169.36GB/s  169.40GB/s  169.39GB/s
          5                         dram_utilization                    Device Memory Utilization    High (8)    High (8)    High (8)
```

Figure 3-10: GPU Results Vector Size = 10^8 – (2)

From the results we observe that the multiprocessor resources provided by the GPU are being used almost to the peak from about 90% on the 10^6 run to almost 100% for both the 10^7 and 10^8 executions. This is confirmed by the charts below as well.

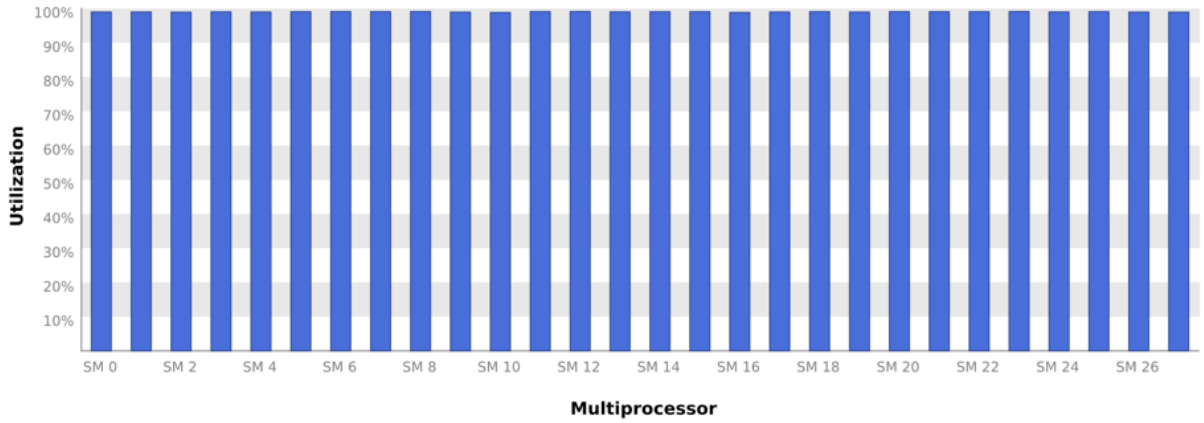

Figure 3-11: Multiprocessor Utilization

However, it is immediately evident between the 3 runs what could possibly be the bottleneck for the performance of our application.
We observe a high level of utilization of the device memory utilization which leads us to conclude that our application is limited by the bandwidth of the device memory. We notice as well, the impact of this on the computation throughput of the multiprocessors. These conclusions are further highlighted by the following charts generated from the graphical analysis tool.

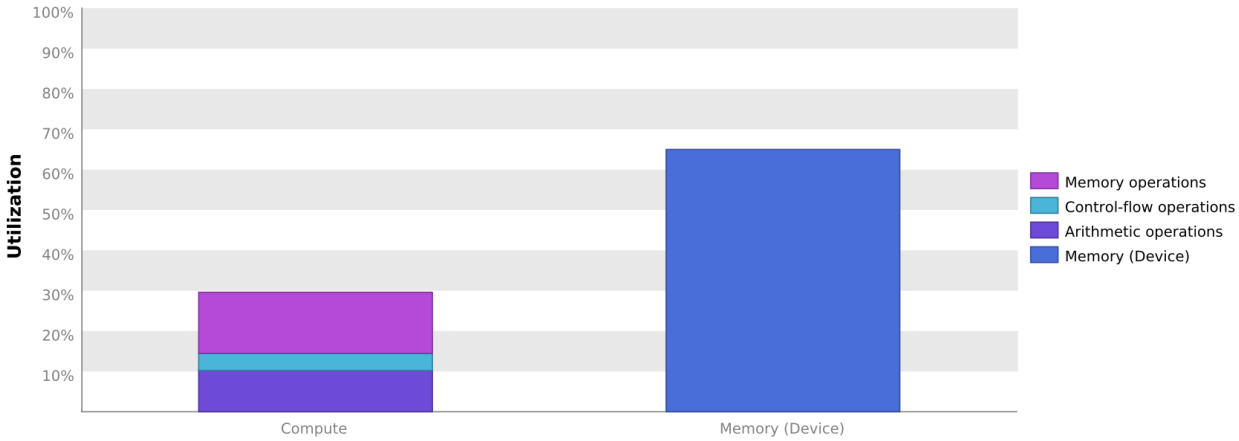

Figure 3-12: Compute and Memory Utilization

3.4 Time Blocking

As an optional requirement from the assignment, the code has been modified to implement the concept called time blocking. The main idea of this implementation is to minimize the data read and writes between the device memory and L2 cache during subsequent iterations.

The full code for implementing this is attached as part of our submission for this assignment.

The output after compilation of the code is shown below highlighting sections of the code which have been parallelized and vectorized accordingly by the compiler.

```
Compiling for CPU
main:
     72, Loop is parallelizable
         Generating Multicore code
         72, #pragma acc loop gang
    103, Loop is parallelizable
         Generating Multicore code
        103, #pragma acc loop gang

Compiling for GPU
main:
     56, Generating copy(U2[:N+2],U1[:N+2])
     61, Accelerator serial kernel generated
         Accelerator kernel generated
         Generating Tesla code
     72, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         72, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     81, Accelerator serial kernel generated
         Accelerator kernel generated
         Generating Tesla code
     94, Accelerator serial kernel generated
         Accelerator kernel generated
         Generating Tesla code
    103, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        103, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    112, Accelerator serial kernel generated
         Accelerator kernel generated
         Generating Tesla code
```

Figure 3-13: Compiler output – Time blocking code

3.4.1 - CPU EXECUTION Result

The results for executing the time blocking version for problem size of 10^8 is shown below:

We do notice a reduction in the execution time by 2x and the computation throughput has increased by over 3x. (from 0.68 to 2.11) which is a tremendous improvement. So even if there are a lot more instructions being executed for the time blocking version (~1.5x), we can assume there is a reduced latency as we are reducing main memory accesses which in turn increases our computation throughput.

```
Running and profiling on CPU
Stencil computation of 10000 steps on 1-D vector of 100000000 elements with L=1.2339999666e-03

CheckSum = 5.5842277527e+01

 Performance counter stats for './hCPU 10000 100000000':

    1470535.903619      task-clock (msec)         #    3.988 CPUs utilized
              2230      context-switches          #    0.002 K/sec
                 9      cpu-migrations            #    0.000 K/sec
              2342      page-faults               #    0.002 K/sec
     5287773265177      cycles                    #    3.596 GHz
     2521868940501      stalled-cycles-frontend   #   47.69% frontend cycles idle
   <not supported>      stalled-cycles-backend
    11279371146161      instructions              #    2.13  insns per cycle
                                                  #    0.22  stalled cycles per insn
      259057191872      branches                  #  176.165 M/sec
           6400271      branch-misses             #    0.00% of all branches

     368.695720395 seconds time elapsed
```

Figure 3-14: CPU Execution Results – Time blocking

3.4.2 - GPU Execution Results

Below are the results after execution of the time blocking version of the code on the GPU.
We observed that the execution time is reduced by over 1.6x while the computation throughput of the multiprocessor is increased by close to 1.3x.
We notice, as well, a reduction of the DRAM utilization to about 70% which is still quite high. However, an interesting point is the extreme increase in the L2 cache utilization precisely the number of reads, which far out-weights the number of writes. This is most likely due to the effect of the time blocking concept where data blocks read from the main memory are kept longer in the cache memory and used for further computations.

```
Running on GPU
Stencil computation of 10000 steps on 1-D vector of 100000000 elements with L=1.2339999666e-03

CheckSum = 5.5842250824e+01

real    0m15.237s
user    0m10.859s
sys     0m3.645s

Running and profiling on GPU
==26210== NVPROF is profiling process 26210, command: ./hGPU 10000 100000000
==26210== Stencil computation of 10000 steps on 1-D vector of 100000000 elements with L=1.2339999666e-03

CheckSum = 5.5842250824e+01
Profiling application: ./hGPU 10000 100000000
==26210== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.56%  6.48541s      2500  2.5942ms  2.3526ms  17.526ms  main_72_gpu
                   49.13%  6.42886s      2500  2.5715ms  2.3331ms  17.428ms  main_103_gpu
                    0.72%  94.854ms        48  1.9761ms  1.3499ms  2.1667ms  [CUDA memcpy HtoD]
                    0.48%  62.876ms        48  1.3099ms  1.1103ms  1.3213ms  [CUDA memcpy DtoH]
                    0.03%  4.0410ms      2500  1.6160us  1.2160us  12.672us  main_81_gpu
                    0.03%  3.5820ms      2500  1.4320us  1.1200us  11.777us  main_94_gpu
                    0.03%  3.5018ms      2500  1.4000us  1.2160us  11.904us  main_112_gpu
                    0.02%  2.9915ms      2500  1.1960us     992ns  13.921us  main_61_gpu
```

Figure 3-15: GPU Execution Results (1) – Time blocking

```
CheckSum = 2.1243949890e+01
==26353== Profiling result:
==26353== Metric result:
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 Ti (0)"
    Kernel: main_72_gpu
          5                        l2_read_throughput                          L2 Throughput (Reads)  940.08GB/s  947.68GB/s  946.05GB/s
          5                       l2_write_throughput                         L2 Throughput (Writes)  195.74GB/s  197.39GB/s  197.04GB/s
          5                             inst_executed                           Instructions Executed   127262560   127262560   127262560
          5                             sm_efficiency                      Multiprocessor Activity      99.87%      99.89%      99.88%
          5                                       ipc                                  Executed IPC    1.026370    1.027443    1.026992
          5                          l2_utilization                          L2 Cache Utilization     Mid (5)     Mid (5)     Mid (5)
          5                    dram_read_throughput                  Device Memory Read Throughput  156.85GB/s  158.15GB/s  157.86GB/s
          5                   dram_write_throughput                 Device Memory Write Throughput  156.80GB/s  158.14GB/s  157.84GB/s
          5                       dram_utilization                      Device Memory Utilization    High (7)    High (8)    High (7)
```

Figure 3-16: GPU Execution Results (2) – Time blocking

4. Conclusion

As part of this exercise we have presented code modification made using OpenACC constructs and clauses in order to implement parallelism and vectorization in the solution of the two different problems.
We have presented detailed analysis of the performance results of the execution of the two different programs for different problem sizes and scenarios.
We have demonstrated that if the problem size is not big this can hide the real bottlenecks of the performance of the application. This was evident in the case of the solution of the linear system of equations where the issue of the limitation of performance by the memory bandwidth was exposed only after increasing the problem size from 1000 to 4000.
Also, we have highlighted that the issue of performance limitation as a result of the memory resources can be identified when the program is run on the CPU by checking the evolution of the cache misses.
An effective metric used to identify this issue again is the change in the computation throughout when the problem size is increased. We noticed that the IPC is largely impacted when the application is limited by memory bandwidth which is understandable as the latency tends to increase consequently.
We have demonstrated how to confirm that our application is making good use of the computation resources provided by the CPU and GPU using the respective metrics from the profiler outputs.
To address the challenge of memory bandwidth being a bottleneck, we have attempted to implement time blocking to limit the number main memory access. This implementation through increased the amount of computations, however it led to a reduction in latency and eventual increase in computation throughput. Also this resulted in effective use of data in the L2-cache and less access of the slow main memory. This eventually led to a reduction in the execution time.

4.1 References

1. Lecture Notes (Programming Massively-Parallel Architectures (GPUs and accelerators) – *Juan Carlos Moure*
2. OpenACC API 2.5 (Reference Guide)