# PACC: An Extension of OpenACC for Pipelined Processing of Large Data on a GPU

Tomochika Kato, Fumihiko Ino, and Kenichi Hagihara
Graduate School of Information Science and Technology
Osaka University
1–5 Yamadaoka, Suita, Osaka 565–0871
Email: {t-kato, ino, hagihara}@ist.osaka-u.ac.jp

***Abstract*—We present a suite of directives, named pipelined accelerator (PACC), and its implementation for accelerating large-scale computation on a graphics processing unit (GPU). PACC extends OpenACC to achieve division of large data that cannot be entirely stored in device memory. Given a program with PACC directives, our PACC translator rewrites the program into an OpenACC program such that data is divided into multiple chunks for accelerated execution. Furthermore, the generated program processes chunks in a pipeline so that data transfer between the CPU and GPU can overlap with computation on the GPU. Some preliminary results are also presented to show the impact of PACC in terms of the program execution time and the maximum data size that can be processed successfully.**

## I. INTRODUCTION

Directive-based programming is an attractive approach for reducing development efforts needed to implement parallel code on an accelerator device such as the graphics processing unit (GPU) and Xeon Phi. For example, OpenACC provides a collection of directives that are useful to specify segments of serial code to be offloaded to an accelerator device. Consequently, accelerated code can be easily obtained by adding directives to the serial code with parametric optimization. Furthermore, such additional directives can be ignored by a compiler option. This means that the application code can be easily rolled back to the original state, which does not include accelerator-specific code. From a long-range point of view, this type of code flexibility is essential to achieve *performance portability*, which ensures high performance on different platforms.

Although OpenACC significantly lowers the barrier to accelerated computing, it assumes that the entire data is stored in device memory. Therefore, rewriting of OpenACC code is required to process large data that cannot fit into the limited capacity of device memory, which is an order of magnitude smaller than that of main memory. Without this code modification, the program execution can result in a failure due to device memory exhaustion. Figure 1 shows an example of OpenACC code modification. The original code of Fig. 1(a) fails to run if the array size `n` is too large to fit in device memory. To avoid this, the array data `s` and `d` have to be divided into smaller portions, called *chunks*, which are then processed on the accelerator device in a sequence. However, as shown in Fig. 1(b), this adaptation changes the loop structure in the code, diminishing the benefit of directives. That is, the modified code loses the performance portability because it can fail to efficiently run on other machines with a large-capacity memory.

```
1  #pragma acc parallel loop copyin(s[0:n]) copyout(d[1:n-2])
2  for (i=1; i<n-1; i++) {
3      d[i] = s[i-1] + s[i+1];
4  }
```

(a)

```
1  for (j=1; j<n/csize; j++) {
2  #pragma acc parallel loop copyin(s[csize*j:csize]) \
3      copyout(d[csize*j+1:csize-2])
4      for (i=1; i<csize-2; i++) {
5          d[csize*j+i] = s[csize*j+i-1] + s[csize*j+i+1];
6    }
7  }
```

(b)

Fig. 1. Examples of OpenACC code. (a) A naive code that can result in an execution failure due to device memory exhaustion. (b) A rewritten code that avoids device memory exhaustion by data division.

In this work, we present an extension of OpenACC, named pipelined accelerator (PACC), which is capable of pipelined execution of large-scale stencil computation on a GPU. We also present a source-to-source translator that generates a pipelined OpenACC code from a PACC code. According to PACC directives, our translator automatically divides large data into smaller chunks, which are then processed in a pipelined manner. This pipelined execution is efficiently realized by overlapping computation with data transfer between the CPU and GPU.

## II. PACC: PIPELINED ACCELERATOR

PACC directives allow application developers to specify a data division scheme and an execution configuration of the pipeline. Figure 2 shows an example of PACC code that includes four PACC directives. Similar to OpenACC directives, a PACC directive starts with `#pragma pacc`. The `pipeline` construct at line 1 defines the code region to be processed in a pipeline. This construct is similar to the `data` construct of OpenACC and can have clauses such as `targetin` and `targetout`. These clauses differ from the `copyin` and `copyout` clauses of OpenACC in terms of specifying a data division scheme. For example, the argument `s(l,c,r)` of a `targetin` clause specifies that the array `s` to be copied to the device memory must be divided into chunks of `c` elements with a halo of left size `l` and right size `r`. Note that the chunk size `c=1` at line 1 represents the size with respect to

```
1   #pragma pacc pipeline targetin(s(0,1,0)[1:N-2][0:N]) \
2       targetout(d(0,1,0)[1:N-2][1:N-2])
3   {
4   #pragma pacc update device(s[1:N-2][0:N])
5   #pragma pacc parallel loop dim(2)
6       for (j=1; j<N-1; j++) {
7           for (i=1; i<N-1; i++) {
8               d[j][i] = s[j][i-1] + s[j][i+1];
9           }
10      }
11  #pragma pacc update host(d[1:N-2][0:N])
12  }
```

Fig. 2. An example of PACC code.

the second dimension j. In this example, each chunk consists of N elements, because the array is not divided with respect to the first dimension i.

The remaining constructs in Fig. 2 are similar to those of OpenACC, but the parallel construct at line 5 has an extended clause dim, which associates the loop control variable with the dimension of the array data to be divided. For example, the clause dim(2) at line 5 indicates that the loop control variable j at line 6 specifies the second dimension of the array data.

Our PACC translator is implemented using the ROSE compiler infrastructure [1], which provides C and C++ frontend to generate an abstract syntax tree (AST) of the given code. The generated AST is then traversed to detect the vertices with the pacc attribute. The detected vertices are marked explicitly for modification, so that our rewrite rules are applied to them in the next traversal. Finally, the modified AST is given to the ROSE code generator to obtain the pipelined OpenACC code.

## III. Experimental Results

We applied our translator to three OpenACC-based applications [2], [3]: Black Scholes (BS), Sobel filter (SF) and 27-point stencil (27S). Using these codes, we implemented PACC and OpenMP versions to compare their performance and the maximum data size that can be processed on a GPU. Our experimental machine had a Core i7 3930K processor, 32 GB main memory, and a Tesla K20 GPU with 5 GB device memory. We used PGI Compiler 14.6, CUDA 6.0, and Windows 7 64-bit.

Figure 3 shows timing results, comparing the performance of the PACC code with those of OpenMP and OpenACC codes. For BS, SF, and 27S, we increased the number of options, the resolution of images, and the size of dimensions, respectively. For all applications, the OpenACC version failed to process data larger than approximately 4.9 GB. In contrast, PACC successfully processed large data up to 25.8 GB, which was close to the maximum data size of 29.3 GB that can be processed by OpenMP. The gap between these data sizes is due to the buffer needed to store a chunk in main memory. This buffer is required to separate the chunk from the original large array, because OpenACC 1.0 assumes the same variable name between device data and host data. This assumption is eliminated by OpenACC 2.0, which provides a mapping function named acc_map_data. By using this new capability, we found that PACC successfully processed the data size of up to 29.3 GB. However, this capability disabled pinned memory, failing to achieve pipelined execution. Consequently, the performance was slightly lower than the non-pipelined version in Fig. 3.

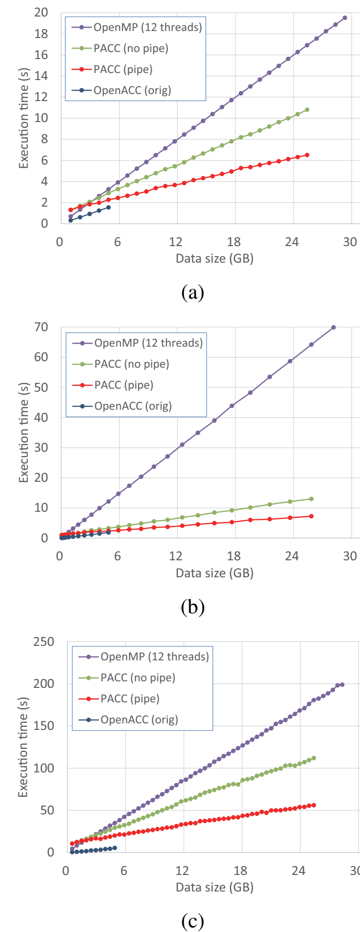


Fig. 3. Timing results using (a) BS, (b) SF, and (c) 27S.

With respect to the performance, PACC was 2.6, 8.7, and 3.2 times faster than OpenMP for BS, SF, and 27S, respectively. However, the highest performance was obtained with OpenACC for small data less than 4.9 GB. For such small data, PACC was slower than OpenACC, because of the overhead of device memory allocation. Our translator currently uses a chunk size of 4.9 GB, which is close to the capacity of device memory. Owing to this large chunk size, it took approximately 500 ms to allocate a buffer in device memory. In addition, the original array data must be copied to the buffer in host memory. This additional overhead occupied 45% of the execution time, but our software pipeline successfully overlapped the overhead with data transfer between the CPU and GPU.

## References


[1] http://rosecompiler.org/

[2] http://www.caps-entreprise.com/resources-and-support/openacc-examples/

[3] http://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite/