Directive-Based Pipelining Extension for OpenMP

Xuewen Cui
Virginia Tech
Blacksburg, VA 24060
xuewenc@vt.edu

Thomas R. W. Scogland
Lawrence Livermore
National Laboratory
Livermore, CA 94550 USA
scogland1@llnl.gov

Bronis R. de Supinski
Lawrence Livermore
National Laboratory
Livermore, CA 94550 USA
bronis@llnl.gov

Wu-chun Feng
Virginia Tech
Blacksburg, VA 24060
feng@cs.vt.edu

***Abstract*—Heterogeneity continues to increase in computing applications, with the rise of accelerators such as GPUs, FPGAs, APUs, and other co-processors. They have also become common in state-of-the-art supercomputers on the TOP500 list. Programming models, such as CUDA, OpenMP, OpenACC and OpenCL are designed to offload compute intensive workloads to co-processors efficiently. However, the naive offload model, synchronously copying and executing, in sequence is inefficient while pipelining these activities reduces programmability.**

We propose a directive-based pipelining extension for OpenMP. Our extension offers a simple interface to overlap data transfer and kernel computation with an auto-tuning scheduler. We achieve performance improvements between 40% and 60% for a Lattice QCD application.

I. Introduction

Heterogeneous systems, particularly those containing GPUs, are becoming prominent on the list of the TOP500 supercomputers. Many new purpose-built models have been created for accelerators, but rather than grapple with unfamiliar programming models scientists usually prefer to keep their existing verified C or FORTRAN code. The OpenMP 4.0 [1], [2] device constructs and OpenACC [3] were designed to allow the straightforward continued use of those trusted codes. Without extensively rewriting their code, users can add directives to offload their computation to the accelerator.

These models all use a similar offload procedure. Users copy their data to the accelerator, launch their computation and copy their output back to the host. This naive offload model is straightforward but, run serially, data transfers can consume a large portion of the execution time. Moreover, scientific applications often use huge data arrays or matrices that do not fit in the GPU's on-chip memory.

To address these issues we propose an OpenMP extension to overlap data transfers and computation automatically through pipelining. Our extension offers a simple interface to pipeline a parallel loop with an auto-tuning scheduler. We find performance improvements between 40% and 60% on an NVIDIA Tesla K40m GPU.

II. Design

Our goal of automated pipelining presents two main challenges. First, we must overlap data transfers and kernel computation correctly. Second, we must calculate a chunk size, number of loop iterations, that will allow all necessary inputs and outputs of a chunk to fit in memory on the target device.

```
#pragma target teams distribute parallel for\
  pipeline(<scheduler>(<chunk_size>,<num_stream>))\
  pipeline_copy(in/out)(<var>(size)[<cond>:<num>])\
pipeline_shadow(in/out)(<var>(<offsets…>:<cond_roll>))\
  pipeline_mem_limit(<mem_size>)
```

pipeline() inputs	
<scheduler>	Scheduler to use for this region(static, adaptive)
<chunk_size>	Sub-task chunk size
<num_stream>	Stream number to launch on GPU
pipeline_copy() and pipeline_shadow() inputs	
<in/out>	Input data or output data
<var>	Variable(array) to copy
<size>	Size of each "item" in the array/matrix
<cond>	Whether this dimension is the outer split loop
<num>	Number of items in this dimension
<offsets…>	Shadow element offsets
<cond_roll>	If the shadow should wrap around
<mem_size>	Maximum memory usage

Fig. 1. Our proposed extension

We address these challenges by dividing the loop into several smaller chunks, then launch each on a different GPU stream. As soon as the data transfers for the first chunk finish, its kernel launches. Each chunk's transfers and computation are enqueued separately, and thus may run in parallel. Control over the chunk size also allows us to avoid exceeding available device memory without complex coding, and to auto-tune the chunk size and the number of streams for better performance.

Figure 1 presents our extension. The `pipeline()` clause specifies which scheduler to use. Users can choose between *static* and *adaptive*. The *static* scheduler requires the user to specify `chunk_size` and `num_stream`. With the *adaptive* scheduler, the user can provide hints for the initial chunk size and the maximum number of streams, which are auto-tuned at run-time. The `pipeline_copy()` clause defines the input and output arrays in terms of their size, number of elements per dimension, and association with the loop iterations being scheduled by the construct. We must consider data dependencies, so we integrate this information into the `pipeline_shadow()` clause by specifying which inputs are necessary to produce the output of a given iterator value. We also limit the memory usage by the `pipeline_mem_limit()` clause. Figure 2 shows an example for a simple stencil computation.

III. Evaluation and Analysis

We use a Lattice QCD code as a case study for our proposed extension, and test it with two GPUs, a Tesla K40m and a

```
#pragma target teams distribute parallel for\
  pipeline(static(1,3))\
  pipeline_copyin(A0[0:nx-1][0:ny-1][true:nz-1])\
  pipeline_copyout(Anext[0:nx-1][0:ny-1][true:nz-1])\
  pipeline_shadowin(A0(-1,0,1:true))\
  pipeline_shadowout(Anext(0:0))\
  pipeline_mem_limit(MB_256)
for(k=1;k<nz-1;k++) {
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[Index3D (i, j, k)] =
         (A0[Index3D (i, j, k + 1)] +
          A0[Index3D (i, j, k - 1)] +
          A0[Index3D (i, j + 1, k)] +
          A0[Index3D (i, j - 1, k)] +
          A0[Index3D (i + 1, j, k)] +
          A0[Index3D (i - 1, j, k)])*c1
        - A0[Index3D (i, j, k)]*c0;
    } } }
```

Fig. 2. A stencil benchmark example

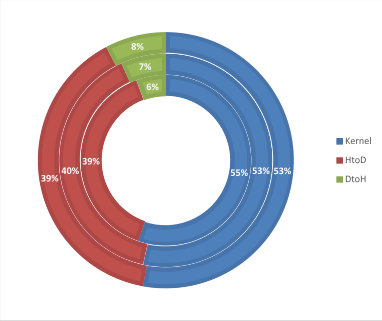

Fig. 3. Time consumption distribution

C2070. The main kernel is implemented in OpenACC and manually split into chunks, launch with a specific number of GPU streams. Each result is an average of six tests.

We have three data sets: the number of inner-loop iterations is 16^4, 24^4 and 32^4. The pie chart in Figure 3 shows the time of the kernel and data transfer for each dataset, 16, 24, 32 from the inside out. We observe that the data transfer takes a large portion of execution time — nearly 50%.

We evaluate the speedup of our pipelined OpenACC version over naive OpenACC. We observe that pipelining achieves up to 1.6× speedup. As the problem size grows, the speedup increases, indicating that larger cases may approach the theoretical upper bound of 2×.

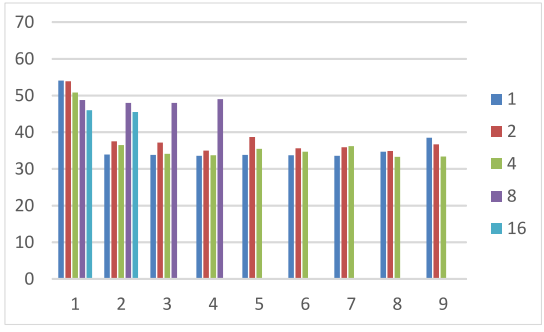

Fig. 4. Time in seconds to run the large test case with different chunk sizes (color) and number of streams (x axis) on K40m

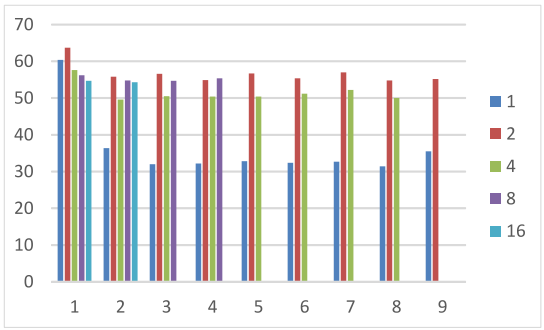

Fig. 5. Time in seconds to run the large test case with different chunk sizes (color) and number of streams (x axis) on C2070

We also vary `chunk_size` and `num_stream`. Figure 4 shows the results for the large test case on the K40m. Using two streams usually performs significantly better than one due to overlapping of data transfers and computation. However, more than four streams offers no further benefits because more streams incur more API and scheduling overhead, while only slightly increasing potential overlap. Increasing `chunk_size` can reduce the API calls and kernel launch overhead, but complicates load balancing. As we can see with `chunk_size` 8 and 16 for the large test case, performance can degrade when the size is too large. On Fermi, increasing the `chunk_size` always reduces performance. In this application, the Kepler GPU needs two streams to reach its best performance while the Fermi GPU needs three or four. The Fermi GPU C2070 outperforms the K40m given enough streams. Further figures can be found in the poster.

IV. Future Work

We will investigate more benchmarks and use-cases for our extension. We will also provide support for non-contiguous data regions. We are also considering a source-to-source translator for our extension based on our previous work [4], [5]. Finally, we will further investigate the behavior of our design with different parameters (e.g., chunk size and number of streams) and integrate a performance model into our auto-tuning scheduler.

References

[1] OpenMP ARB, "OpenMP application program interface version 4.0," 2013.

[2] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the OpenMP accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[3] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC–first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[4] T. R. Scogland, W.-c. Feng, B. Rountree, and B. R. de Supinski, "CoreTSAR: Core Task-Size Adapting Runtime," 2014.

[5] T. R. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski, "Heterogeneous Task Scheduling for Accelerated OpenMP," in *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 144–155.