Technical Report

Cache-Aware Kernel Tiling: An Approach for System-Level Performance Optimization of GPU-Based Applications

1 Obtaining user-provided information

In the paper, we discussed two user-provided pieces of information, namely *edge weights* and *performance tables*, which capture the performance characteristics of a kernel on a specific platform. An edge weight reflects the cache-sensitivity of a consumer node with respect to the input corresponding to the edge. More specifically, it is the maximum amount of time that can be saved by tiling the kernel and providing the input data via the L2 cache. A performance table provides execution time estimations of a kernel at various grid sizes, presuming that certain kernel parameters are provided via tiling and hence most likely can be accessed through the cache. In this technical report, we explain the process of obtaining the edge weights and constructing the performance tables.

To acquire an edge weight, we need to first measure the overall kernel execution time under the best possible tiling scenario with respect to the corresponding input. Towards this, we isolate the original kernel such that the inputs can be directly provided by the user (instead of by the producer node), split the kernel into smaller sub-kernels, and sum up the duration of all sub-kernels. The weight is then calculated by subtracting the obtained sum from the default kernel duration. To achieve the best tiling scenario for a given kernel and an edge, we have to find the appropriate grid size for the sub-kernels. This grid size is the same among all sub-kernels (except for the last sub-kernel that may have a smaller grid size). The appropriate grid size for the sub-kernels has to satisfy the following conditions: *(i)* sub-kernels achieve high cache performance with respect to the input corresponding to the edge, and *(ii)* sub-kernels are sufficiently large to minimize the adverse effect of low GPU utilization on performance. The overall duration of the kernel corresponding to a given sub-kernel size is simply obtained by multiplying the number of required sub-kernels by single duration of a sub-kernel (here, for simplicity, we assume that the original grid size is divisible by the sub-kernel size). A simple approach to find the sub-kernel grid size is to start off with the minimum grid size (i.e., one block), which provides the best cache performance for a sub-kernel, and gradually increase the size until we find the best performance trade-off between the cache hit rate and the GPU utilization. Meanwhile, to ensure that the data corresponding to the input edge reside

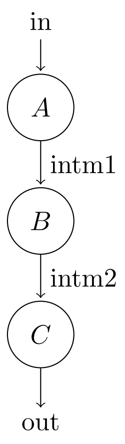

Figure 1: Sample application graph

in the cache at the time of first access, we exploit the fact that all memory operations in GPU must go through the L2 cache. Therefore, to warm up the cache before accessing the data, it is enough to copy the sample input data from the host memory to the GPU memory using the explicit CUDA memory-copy operation `cudaMemcpy`. We also use the information regarding the cache hit rate and the GPU utilization, provided by the NVIDIA profiler tool, to analyze the performance of a sub-kernel at various grid sizes.

For any kernel, we build one or more performance tables. The exact number of the performance tables depends on the number of kernel parameters that may be provided via tiling. For example, a node with two parameters requires four performance tables, corresponding to four different combinations of whether any of the parameters is provided via tiling or not. The procedure to construct the table for each combination is similar to the one described above to acquire the edge weights: For each grid size, we first warm up the cache (using the memory-copy operation) such that the data elements of the input parameters that are supposed to be provided via tiling reside in the cache and others do not. Next, we launch the kernel and measure the kernel duration at the given grid size. We repeat the same procedure with different grid sizes. However, to avoid repeating the same process for every possible grid size, we select a few sample grid sizes and use interpolation to estimate the kernel duration for the missing grid sizes.

2 Enforcing a sample schedule

Below, we provide a simplified example to illustrate the source code modification required to enforce the schedule produced by KTILER. The application consists of three consecutive kernels with data dependencies between them, as depicted by the application graph in Figure 1. The parameters `in` and `out` are the input and output data, respectively, while `intm1` and `intm2` carry the intermediate data.

Listing 1 shows the application source code in the default mode. After performing the preliminary steps, the main function on the host invokes the GPU kernels, successively, and reads back the final result (i.e., `out`) from the GPU memory. The kernels in this example are two dimensional. At the beginning of each kernel, each thread computes its global x and y indices using its two-dimensional local thread index as well as the index of the block to which it belongs (lines 5–6).

```
 1 __global__
 2 void A(float * in, float * intm1)
 3 {
 4   //Compute global indices for each thread
 5   int ix = threadIdx.x + blockIdx.x * blockDim.x;
 6   int iy = threadIdx.y + blockIdx.y * blockDim.y;
 7 
 8   //Implementation of kernel A
 9 }
10 
11 __global__
12 void B(float * intm1, float * intm2)
13 {
14    //Define global indices and implement kernel B
15 }
16 
17 __global__
18 void C(float * intm2, float * out)
19 {
20    //Define global indices and implement kernel C
21 }
22 
23 void main()
24 {
25   //Initialization and preparation
26 
27   //Allocate memory on GPU for in, intm1, intm2, and out
28 
29   //Copy the input data from host memory to GPU memory
30 
31   //Invoke the kernels
32   A <<<numBlocksA, threadsPerBlockA>>> (in, intm1);
33   B <<<numBlocksB, threadsPerBlockB>>> (intm1, intm2);
34   C <<<numBlocksC, threadsPerBlockC>>> (intm2, out);
35 
36   //Wait until all kernels are completed
37   cudaDeviceSynchronize();
38 
39   //Read back the result (ie. 'out') from the GPU
40 
41   //Rest of the program
42 }
```

Listing 1: Default program

Listing 2 shows the modified source code that enforces the tiling schedule. Here, the main function first reads the schedule from a text file. The schedule contains the information regarding the exact tiling of each cluster. In this example, there is only one cluster, which includes all three kernels. Within a cluster, tiling is defined as a vector of sub-kernel sequences. Each sequence is a vector of data structures, where each data structure carries the information required to launch a sub-kernel corresponding to one of the member nodes in the cluster. Specifically, a data structure contains a subset of original kernel block ids and the number of blocks that must be processed by the corresponding sub-kernel. At runtime, every block of a sub-kernel uses these information to obtain the coordinates of the unique data segment that it must process. Lines 44–46 store all the block ids in the schedule in a one-dimensional array on the host memory, which is then copied to a one-dimensional array in the GPU memory. Note that during the scheduling phase, multi-dimensional block ids are converted into one-dimensional format. The purpose is mainly to minimize the number of memory accesses made by the threads to read the original block ids at the beginning of each kernel (see lines 4–5). Next, the sub-kernels are invoked in the exact order specified by the schedule through a for-loop (line 51). The loop goes through the sub-kernel sequences within the tiling cluster. The body of the loop invokes the three sub-kernels associated with the member kernels. In our example, the sub-kernel of node `A` is launched first. Line 53 obtains the grid size of the sub-kernel from the first element of the current sub-kernel sequence. The sub-kernel is then launched, in line 54, by specifying the grid size and the block size (note that the block size remains unchanged). In addition to the two input and output parameters, the sub-kernel takes two extra parameters. Parameter `d_bids` points to the first original block id of the current sub-kernel, and `GX_A` holds the original grid size of kernel `A` along the x axis. Upon invocation, threads in a block use their local block id within the sub-kernel to obtain the global block id within the original kernel. As mentioned earlier, the obtained global block id is one dimensional. Therefore, in lines 4–5, each thread uses the original grid size in the x dimension to obtain the original two-dimensional block id. Next, in line 55, the host increments `d_bids` by the grid size of sub-kernel of node `A` so that it points to the first original block id of the sub-kernel of node `B`. This procedure continues all sub-kernels are processed.

```
1  __global__
2  void A(float * in, float * intm1, const int * d_bids, const int gx)
3  {
4    int blockIdx_x = d_bids[blockIdx.x] % gx;
5    int blockIdx_y = d_bids[blockIdx.x] / gx;
6
7    //Define global indices for each thread
8    int ix = threadIdx.x + blockIdx_x * blockDim.x;
9    int iy = threadIdx.y + blockIdx_y * blockDim.y;
10
11   //Implementation of kernel A
12 }
13
14 __global__
15 void B(float * intm1, float * intm2, const int * d_bids, const int gx)
16 {
17    //Modified similar to kernel A
18 }
19
20 __global__
21 void C(float * intm2, float * out, const int * d_bids, const int gx)
22 {
23    //Modified similar to kernel A
24 }
25
```

```
void main()
{
  //Initialization and preparation

  //Allocate memory on GPU for in, intm1, intm2, and out

  //Copy the input data from host memory to GPU memory

  //Read the schedule from a text file
  readSch(sch, loc);

  //One dimensional vector to hold all block IDs on host (CPU)
  vector<int> h_bids;

  //One dimensional array to hold all block IDs on GPU
  int * d_bids;

  //Copy all block IDs to bids
  for (auto sk_seq = sch->begin(); sk_seq != sch->end(); sk_seq++)
    for (auto sk = sk_seq->begin(); sk != sk_seq->end(); sk++)
      h_bids.insert(h_bids.end(), sk->bids.begin(), sk->bids.end());

  //Copy h_bids to d_bids

  //Invoke the sub-kernels in a for-loop
  for (auto sk_seq = sch->begin(); sk_seq != sch->end(); sk_seq++)
  {
    numBlocksA = sk_seq->at(0).gridSize;
    A <<<numBlocksA, threadsPerBlockA>>> (in, intm1, d_bids, GX_A);
    d_bids += numBlocksA;

    numBlocksB = sk_seq->at(1).gridSize;
    B <<<numBlocksB, threadsPerBlockB>>> (intm1, intm2, d_bids, GX_B);
    d_bids += numBlocksB;

    numBlocksC = sk_seq->at(2).gridSize;
    C <<<numBlocksC, threadsPerBlockC>>> (intm2, out, d_bids, GX_C);
    d_bids += numBlocksC;
  }

  //Wait until all kernels are completed
  cudaDeviceSynchronize();

  //Read back the result (ie. 'out') from the GPU

  //Rest of the program
}
```

Listing 2: Transformed program