Automatic Horizontal Fusion for GPU Kernels

Ao Li[§]
Carnegie Mellon University
USA
aoli@andrew.cmu.edu

Bojian Zheng
University of Toronto
Canada
bojian@cs.toronto.edu

Gennady Pekhimenko
University of Toronto
Canada
pekhimenko@cs.toronto.edu

Fan Long
University of Toronto
Canada
fanl@cs.toronto.edu

***Abstract*—We present automatic horizontal fusion, a novel optimization technique that complements the standard kernel fusion techniques for GPU programs. Unlike the standard fusion, whose goal is to eliminate intermediate data round trips, our horizontal fusion technique aims to increase the thread-level parallelism to hide instruction latencies. We also present HFUSE, a new source to source CUDA compiler that implements automatic horizontal fusion. Our experimental results show that the horizontal fusion can speed up the running time by 2.5%-60.8%. Our results reveal that the horizontal fusion is especially beneficial for fusing kernels with instructions that require different kinds of GPU resources (e.g., a memory-intensive kernel and a compute-intensive kernel).**

***Index Terms*—GPGPU, Code Generation, Performance, Optimization**

I. Introduction

Graphics Processing Units (GPUs) are widely used to speed up deep learning tasks, scientific computation, and even cryptocurrency mining. Each GPU comes with dozens to hundreds of processing cores enabling thousands of threads running in parallel to achieve much higher computational throughput than a normal CPU [1]. Despite of the rapid advancement of the GPU hardware, the applications running on GPUs are always hunger for more performance. For example, training state-of-the-art deep learning models like ResNet-50 can take 2 hours on 8 Tesla V100 GPUs [2].

To speed up computational tasks running on GPUs, especially for deep learning, people have developed many optimization techniques at the software level [3–8]. Among these techniques, *Kernel fusion* is a popular and effective one [9–14] and it is adopted by almost all deep learning frameworks [4–8, 15, 16]. In GPU programs, a large computational task (e.g., training a neural network) is broken down into multiple *kernels*, each of which corresponds to a small parallelizable sub-task that will be dispatched to GPUs to execute. The idea of kernel fusion is to combine two or more kernels into one large but equivalent kernel to potentially improve the overall performance.

The standard kernel fusion technique in the deep learning frameworks combines kernels *vertically*. The fused kernel will have the same number of threads as the two original kernels. Each thread of the fused kernel sequentially combines the instructions of the corresponding threads of the original kernels [4–7, 16]. The potential performance advantage is from reducing expensive data round trips to the GPU memory — without the fusion the first original kernel needs to write its output to the memory for the second kernel to read.[1] Therefore the standard fusion application is typically limited to neighboring kernels in the data dependency graph, i.e., the output of one kernel is the input of another.

Horizontal Fusion: We present a novel optimization technique, *automatic horizontal fusion*. Unlike the standard fusion that aims to eliminate intermediate data round trip, our horizontal kernel fusion enables the fused kernel to better utilize GPU resources and to better hide instruction latencies. The horizontal fusion complements the standard vertical fusion in its application scenarios — horizontally fusing two kernels is beneficial if the two kernels contain instructions that require different types of GPU resources (e.g., a memory-intensive kernel and a compute-intensive kernel).

We also present HFUSE, a source to source CUDA compiler that implements our automatic horizontal fusion technique. Given the CUDA source code of two kernels, HFUSE automatically produces the horizontally fused kernel that is functionally equivalent to the two but runs potentially faster. In the horizontally fused kernel, the threads are partitioned into two intervals based on their thread ids. Each interval corresponds to threads for the computation of one original kernel. The fused kernel combines the instructions of the original kernels with branch statements. The branch conditions checks the current thread id to dispatch the execution to the path of the corresponding kernel. Because threads of two original kernels coexist in parallel during the execution, the horizontal fusion exploits the thread-level parallelism. It enables the thread scheduling hardware (e.g., warp schedulers in NVIDIA GPUs) to automatically interleave instructions from different kernels to hide instruction latencies.

One challenge of implementing the automatic horizontal fusion is to handle synchronization barriers. A typical CUDA barrier stalls the execution of all threads in a thread block of a kernel until all of the threads reach the barrier. Because a fused kernel contains threads derived from both of the original kernels, such barriers from one of the original kernels will impact the thread execution of another. To address this challenge, HFUSE combines inline PTX assembly instructions with instrumented branch conditions to implement special barriers for the thread sets that correspond to original kernels.

[§]The work was done when Ao was a Master's student at the University of Toronto.

[1]For tiny kernels, both vertical and horizontal kernel fusion also reduces kernel launch overhead.

978-1-6654-0584-3/22 © 2022 IEEE

Another challenge HFUSE faces is to identify the best way to partition the thread space of a fused kernel, which is shared by the instructions of the two original kernels. Because the partition scheme determines how the execution of the original kernels co-exists in GPU, it may significantly impact the performance of the fused kernel. To address this challenge, HFUSE operates with an automatic profiling technique. Given the expected input sizes of two original kernels, HFUSE will automatically search the best thread space partition.

Experimental Results: We evaluate HFUSE with 5 deep learning computational kernels extracted from PyTorch [16] and 4 cryptography computational kernels collected from open source cryptocurrency mining programs [17, 18]. In total, we apply HFUSE to fuse 16 pairs of kernels. We compare the running time of the HFUSE fused kernel with the native kernels launched in parallel and the kernels fused in the standard vertical way. Our results show that the HFUSE fused kernels run up to 60.8% faster than the native kernels; for 7 out of the 16 pairs on 1080Ti GPU and 6 out of the 16 pairs on V100 GPU, the HFUSE fused kernels outperform both the vertically fused kernels and the native kernels.

Our results reveal that the speed up of the horizontally fused kernels comes from interleaving different kernel computations to hide instruction latencies. Although modern GPU architecture provides hardware solutions to launch multiple kernels on one GPU in parallel, these solutions typically only interleave kernel computations at a coarse granularity. For example, NVIDIA GPUs can schedule parallel kernels to different Stream Multi-Processors (SMP), but they never schedule thread blocks of two different kernels to one SMP at the same time. Each SMP will only execute instructions from one kernel at any time and therefore will only be able to interleave instructions inside one kernel. In contrast, HFUSE enables the fused kernel to interleave instructions from different kernels to execute on the SMP. It can therefore allow GPU to better utilize hardware resources in each SMP to to potentially hide instruction latencies.

Our results also reveal the trade-off between the thread-level and the block-level parallelism for kernel fusion. On one hand, the horizontal fusion enables the thread scheduler to interleave instructions with the improved thread-level parallelism. On the other hand, the fused kernel will require more registers and shared memory resources than individual kernels. If such additional requirement exceeds a breakpoint, it may cause less thread blocks being scheduled to each core to reduce the block-level parallelism. One could view HFUSE as a tool to navigate this trade-off. See Section IV-C.

Contribution: We make the following contributions:

- **Automatic Horizontal Fusion:** This paper presents automatic horizontal fusion, a novel optimization technique that is orthogonal to the standard vertical kernel fusion. The horizontal fusion can enable the GPU hardware to effectively interleave instructions from two original kernels to hide instruction latencies.
- HFUSE: This paper presents the design and implementation of HFUSE, a novel source to source CUDA compiler that implements automatic horizontal fusion.
- **Optimization Scenarios:** This paper identifies the scenarios for applying the horizontal fusion technique. Our results show that horizontal fusion is mostly beneficial when fusing kernels with instructions that have long latencies and that require different GPU resources.

II. OVERVIEW

This section presents background information of GPU architectures and an overview of kernel fusion techniques. In this paper we use the terminology of NVIDIA CUDA platform [1] and the architecture parameters of NVIDIA Pascal [19] and Volta [20] GPUs. Most of the concepts are generally applicable to other GPU platforms and architectures.

A. Background

Kernels, Blocks, and Threads: Kernels are standalone computational routines that the CUDA runtime will dispatch to NVIDIA GPUs to execute in parallel. They are C-like programs that utilize GPU resources including registers, local shared caches, and the global GPU memory. GPUs are SIMD processors, so each kernel launch will start multiple *blocks* in parallel and each block contains multiple threads. The *grid dimension* (i.e., the number of blocks) and the *block dimension* (i.e., the number of threads) are typically tunable constants. It is a common practice in GPU programming to develop kernels that can work with different block dimension parameters. This means that changing block dimensions of the kernels often only influences performance. A kernel program can access its own block id (e.g., `blockIdx.x`) and thread id (e.g., `threadIdx.x`) at the runtime to enable its different threads to potentially process different data.

Stream Multiprocessor and Occupancy: When the CUDA runtime dispatches a kernel to a GPU, the GPU eventually dispatches the blocks of the kernel to Stream Multiprocessors (SMs) to execute. Each GPU has multiple SMs depending on its hardware specification. In the Pascal and Volta architectures, each SM has 64K registers and 96K shared memory cache; each SM can host a maximum of 2048 different threads at the same time; each SM also has multiple CUDA cores for arithmetic operations and multiple memory controllers for accessing the global GPU memory.

Because each SM has the fixed amount of resources, it can execute only a limited amount of blocks in parallel, depending on the kernel resource requirement. This is called the *occupancy* of a kernel. Generally speaking, higher occupancy is usually better because it enables the kernel to exploit the *block-level* parallelism. For example, if a kernel block that uses 24K shared memory, 512 threads, and 64 registers per thread, a SM can only execute two blocks in parallel and the registers become the bottleneck. If the developer optimized the kernel block to use only 32 registers per thread, then the SM could execute four blocks and the occupancy is doubled.

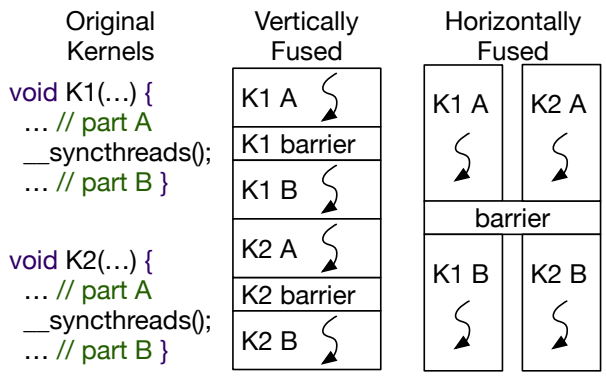

Fig. 1: Vertical and horizontal kernel fusion.

Warps, Warp Scheduler, and Instruction Latency: In SMs, each 32 consecutive threads form a warp. Threads inside a warp always execute together in a lock-step fashion and warps are minimum scheduling units in SMs.[2] The warp scheduler in a SM will select eligible warps to execute — an warp is eligible if 1) all data required for its next instruction is ready, 2) there are idle hardware resources to execute its next instruction (e.g., idle memory controllers for memory access instructions), and 3) it is not stalled by barriers.

Because each SM typically has tens of warps executing in parallel, the warp scheduler can hide *instruction latencies*. If there is a time-consuming instruction in one warp blocking its execution, warp scheduler can switch the SM to execute other eligible warps while waiting for the results of the instruction. Therefore having instructions requesting different hardware resources in a kernel is beneficial because it tends to increase the number of eligible warps for the scheduler. It reduces the chance that the SM execution is completely stalled by instruction latencies.

Synchronization Barriers: The built-in function `__syncthreads()` in CUDA corresponds to block-wide barriers. It is the main way for threads inside a kernel block to coordinate with each other. An SM will stall the thread execution inside a block at a block-wide barrier until all threads in the block reaches the barrier. Note that barriers may significantly limit the capability of warp schedulers in SMs of hiding instruction latencies, because the schedulers cannot interleave instructions across the barriers.

B. Kernel Fusion

Vertical Kernel Fusion: The standard kernel fusion in deep learning frameworks fuses kernels vertically shown as Figure 1. Suppose we have two kernels `K1()` and `K2()` and both of the kernels have the grid dimension of 512 and the block dimension of 512. The code of the vertically fused kernel will combine the source code of `K1()` and `K2()` in order. Therefore the fused kernel will also has the same gird and block dimensions, but one thread in the fused kernel will execute the instructions of two original threads, one in `K1()` and one in `K2()`. The middle part of Figure 1 shows the execution flow of one thread in the vertically fused kernel.

The major potential performance advantage of the vertical fusion comes from eliminating global memory accesses for intermediate results. In this example, the instructions from `K2()` may directly access the output of `K1()` without using expensive global memory read instructions. If some output of `K1()` is only used by `K2()`, the fused kernel can even eliminate associated global memory write instructions. Therefore deep learning frameworks typically apply vertical fusion on neighboring kernels in data dependency graphs.

Note that the vertical fusion may sometime facilitate the instruction interleaving to hide latency, but such effect is typically minimum due to the presence of synchronization barriers. The vertically fused kernel will have as many barriers as the two original kernels and the warp scheduler cannot interleave instructions across these barriers.

Horizontal Kernel Fusion: Unlike the standard kernel fusion, our horizontal fusion technique creates separate threads for instructions of different kernels. The right part of Figure 1 presents the execution flow of the horizontally fused kernel. The fused kernel has the grid dimension of 512 and the block dimension of 1024. The first 512 threads correspond to threads for instructions of `K1()` and the remaining threads correspond to `K2()`. The fused kernel uses branch statements to check the current thread id to dispatch the thread to execute the corresponding instructions.

Note that it is possible to partition the thread space of a block unevenly in the fused kernel, e.g., assigning one kernel 768 threads and another 256 threads. If the block dimensions of the two original kernels are tunable, there will be multiple ways to fuse the two kernels with different thread space partition schemes. Which one runs fastest typically depends on the workload of the original two kernels.

Hypothesis of Horizontal Fusion: Our hypothesis of the horizontal fusion is that its thread-level parallelism will enable the warp scheduler to interleave instructions from different kernels to hide instruction latencies. It may increase the average eligible warps on SMs to improve the overall performance. If our hypothesis is true, then the horizontal fusion will be mostly beneficial for fusing kernels that use different kinds of instructions and kernels that are memory intensive (because memory instructions have long latencies). Our results in Section IV validate our hypothesis.

C. Motivating Example

We next present an example of using HFUSE to horizontally fuse two deep learning kernels. Figure 2 shows the simplified code snippet of `batch_norm_collect_statistics()`, a CUDA kernel that computes the mean and variance of an input tensor for normalization. We extracted this kernel source code from the PyTorch framework [16] and this kernel is used by ResNet [21]. The kernel in Figure 2 uses intra-warp shuffles [22] to speed up its computation. It can operate with a tunable block dimension size as long as the size is a multiple of 32. Each thread first computes the partial results of the

[2]In the Volta and Turing architectures, warps do not restrictively execute in the lock-step fashion but warps are still the minimum scheduling units

```
1 void batch_norm_collect_statistics(input, isize, output) {
2   __shared__ int shared_n[2 * 2 * WARP_SIZE + WARP_SIZE];
3   ... // Local variable declarations
4
5   // PART A: Compute the mean and varience across (batch, x)
6   // It uses shuffles to partially aggregate the results
7   shared_avg_var = (float*) &shared_n[WARP_SIZE];
8   plane = blockIdx.x; N = isize[0] * isize[2];
9   tid = threadIdx.x + threadIdx.y * blockDim.x;
10  avg = 0; var_n = 0; n = 0;
11  for (int batch = threadIdx.y; batch < isize[0]; batch +=
    ↪  blockDim.y) {
12    for (int x = threadIdx.x; x < isize[2]; x += blockDim.x) {
13      float v = input[batch][plane][x];
14      float d1 = v - avg;
15      n++; avg += d1 / n; var_n += d1 * (v - avg); } }
16  for (int i = 0; i < getMSB(WARP_SIZE); ++i) {
17    float o_avg = WARP_SHFL_XOR(avg, 1 << i, WARP_SIZE);
18    int o_n = WARP_SHFL_XOR(n, 1 << i, WARP_SIZE);
19    float factor = 1.0 / fmaxf(1.0, n+o_n);
20    var_n += WARP_SHFL_XOR(var_n, 1 << i, WARP_SIZE) +
21             (avg - o_avg) * (avg - o_avg) * n * o_n * factor;
22    avg = (n * avg + o_n * o_avg) * factor; n += o_n; }
23  __syncthreads();
24
25  // PART B: Write partially aggregated results to shared mem
26  if (tid % WARP_SIZE == 0) {
27    shared_n[tid / WARP_SIZE] = n;
28    shared_avg_var[tid / WARP_SIZE * 2] = avg;
29    shared_avg_var[tid / WARP_SIZE * 2 + 1] = var_n; }
30  __syncthreads();
31
32  // PART C: Another round of suffles to finalize the results
33  if (tid < WARP_SIZE) {
34    n = (tid < blockDim.x * blockDim.y / WARP_SIZE ?
    ↪  shared_n[tid] : 0);
35    avg = (tid < blockDim.x * blockDim.y  / WARP_SIZE ?
    ↪  shared_avg_var[2 * tid] : float(0));
36    var_n = (tid < blockDim.x * blockDim.y  / WARP_SIZE ?
    ↪  shared_avg_var[2 * tid + 1] : float(0)); }
37  for (int i = 0; i < getMSB(WARP_SIZE); ++i) {
38    float o_avg = WARP_SHFL_XOR(avg, 1 << i, WARP_SIZE);
39    int o_n = WARP_SHFL_XOR(n, 1 << i, WARP_SIZE);
40    float factor = 1.0 / fmaxf(1.0, n+o_n);
41    var_n += WARP_SHFL_XOR(var_n, 1 << i, WARP_SIZE) +
42             (avg - o_avg) * (avg - o_avg) * n * o_n * factor;
43    avg = (n * avg + o_n * o_avg) * factor; n += o_n; }
44  if (tid == 0) {
45    ... // Write results to output
46  } }
```

Fig. 2: Normalization kernel.

mean and the variance from the corresponding entries of the tensor with the loop at lines 10-15. The kernel then uses intra-warp operations to aggregate the partial results of each warp (consecutive 32 threads) at lines 16-22. It then writes the partially aggregated results of the 16 warps to the shared memory at lines 26-29 and further aggregates these partial results to produce the output at lines 33-46.

Figure 3 shows the simplified code snippet of `kernelHistogram1D()`, a tensor analysis kernel in PyTorch to generate histograms over values in an input tensor. Because investigating tensor value distributions at hidden layers is a common practice for developers to tune model parameters, this kernel could be invoked during the training of the ResNet model together with the kernel in Figure 2. `kernelHistogram1D()` uses the shared memory array `my_smem` at lines 2-3 to count the appearances of tensor values in different ranges. This kernel also operates a

```
1 __global__ void kernelHistogram1D(TensorInfo a, TensorInfo b,
  ↪  nbins, minvalue, maxvalue, totalElements, getOp) {
2   extern __shared__ unsigned char my_smem[];
3   output_t* smem;
4
5   // PART A: Initialize shared memory counters
6   smem = reinterpret_cast<output_t*>(my_smem);
7   for (int i = threadIdx.x; i < a.sizes[0];
8     i += blockDim.x) { smem[i] = 0; }
9   __syncthreads();
10
11  // PART B: Go over the input b to increment shared counters
12  FOR_KERNEL_LOOP(linearIndex, totalElements) {
13    const int bOffset = IndexToOffset::get(linearIndex, b);
14    const input_t bVal = b.data[bOffset];
15    if (bVal >= minvalue && bVal <= maxvalue) {
16      const int bin = getBin(bVal, minvalue, maxvalue, nbins);
17      atomicAdd(&smem[bin], getOp(linearIndex)); } }
18  __syncthreads();
19
20  // PART C: Increment the output a with the shared counters
21  for (int i = threadIdx.x; i < a.sizes[0]; i += blockDim.x){
22    const IndexType aOffset =
23        IndexToOffset<output_t, IndexType, ADims>::get(i, a);
24    atomicAdd(&a.data[aOffset], smem[i]);
25  } }
```

Fig. 3: Histogram kernel.

tunable block dimension size. It initializes the shared counters at lines 6-9. It then iterates the tensor values to atomically increment the shared counters at lines 12-17 and finally merges the shared counter results with the global counter output at lines 21-25.

Given the two kernels in Figures 2 and 3 as the input, HFUSE horizontally combines them to generate a faster fused kernel shown as Figure 4 with the following steps.

Generate Prologue: HFUSE first generates the prologue for the fused kernel shown as lines 2-23 in Figure 4. The fused kernel has 1024 threads per block. The first 896 threads correspond to the first input kernel (e.g., `batch_norm_collect_statistics()`), while the remaining 128 threads correspond to the second input kernel (e.g., `kernelHistogram1D()`). The prologue checks the current thread id and maps it back to the thread ids of the original kernels, storing them into the variables `threadIdx_x`, `threadIdx_y`, and `threadIdx_z`. It also sets variables like `blockDim_x` to the original input kernel dimensions. The prologue will include all variable declarations from the two input kernels and properly renames these local variables to make sure each of them has a fresh name.

Transform Original Kernels: HFUSE then transforms the original two kernels. Lines 37-40 in Figure 4 present the translated code of the first part of `kernelHistogram1D()`. HFUSE replaces the built-in special values with the corresponding defined variables in the prologue (e.g., replaces `threadIdx.x` with `threadIdx_x` and `blockDim.x` with `blockDim_x`). HFUSE then add additional branch statements to check the current thread id at lines 25 and 36. The branches will skip the execution of the statements of one kernel if the current thread is in the thread range of the other kernel.

Replace Synchronization Barriers: `__syncthreads()` will break the original kernel semantics in the fused kernel, because it will attempt to synchronize all threads in the fused

```
1 void fused_kernel(...) {
2   // Prologue of the fused kernel
3   int global_tid = threadIdx.x + threadIdx.y * blockDim.x +
  ↪   threadIdx.z * BlockDim.x * blockDim.y;
4   int threadIdx_x, threadIdx_y, threadIdx_z;
5   int blockDim_x, blockDim_y, blockDim_z;
6   if (global_tid < 896) {
7     blockDim_x = 896 / 16;
8     blockDim_y = 16; blockDim_z = 1;
9     threadIdx_x = global_tid % blockDim_x;
10    threadIdx_y = global_tid / blockDim_x % blockDim_y;
11    threadIdx_z = 1;
12  } else {
13    blockDim_x = 128;
14    blockDim_y = 1; blockDim_z = 1;
15    threadIdx_x = (global_tid - 896) % blockDim_x;
16    threadIdx_y = 1; threadIdx_z = 1;
17  }
18  // Variable decls for batch_norm_collect_statistics()
19  __shared__ int shared_n[2 * 2 * WARP_SIZE + WARP_SIZE];
20  ...
21  // Variable decls for kernelHistogram1D()
22  extern __shared__ unsigned char my_smem[];
23  output_t* smem;
24
25  if (!(global_tid < 896)) goto K1_end;
26  // batch_norm_collect_statistics() PART A
27  ...
28  // A PTX assembly to only sync 896 threads.
29  asm("bar.sync 1, 896;");
30  // batch_norm_collect_statistics() PART B
31  ...
32  asm("bar.sync 1, 896;");
33  // batch_norm_collect_statistics() PART C
34  ...
35 K1_end:
36  if (global_tid < 896) goto K2_end;
37  // kernelHistogram1D() PART A
38  smem = reinterpret_cast<output_t*>(my_smem);
39  for (int i = threadIdx_x; i < a.sizes[0];
40    i += blockDim_x) { smem[i] = 0; }
41  // A PTX assembly to only sync 128 threads.
42  asm("bar.sync 2, 128;");
43  // kernelHistogram1D() PART B
44  ...
45  asm("bar.sync 2, 128;");
46  // kernelHistogram1D() PART C
47  ...
48 K2_end:
49 }
```

Fig. 4: HFUSE fused kernel.

kernel, which include threads for both of the original kernels. To preserve the original kernel semantics, HFUSE replaces synchronization barriers in the original kernels with inlined PTX assembly `bar.sync` instructions at lines 29, 32, 42, and 45 in Figure 4.

The second parameter of `bar.sync` denotes the number of threads participating the barrier [23]. HFUSE passes 896 for this parameter at lines 29 and 32 and passes 128 at lines 42 and 45. Combining with the inserted branch statements at lines 25 and 36, these `bar.sync` instructions will create the desired partial barriers that only synchronize threads wihtin the corresponding thread ranges of each original kernel.

Profile Different Configurations: Because both of the two original kernels support tunable block dimensions, there are multiple ways to partition the thread space of the fused kernel. Additionally, the fused kernel will use more registers than any

Input : K_1 and K_2 are two input kernels. d_1 and d_2 are the block dimensions of K_1 and K_2.
Output : A fused kernel F

1 function $Generate(K_1, K_2, d_1, d_2)$:
2 Initialize F with local variable declarations from K_1 and K_2 and extract non declaration statements as S_1 and S_2.
3 Append "tid=threadIdx.x; tid_1=threadIdx.x; tid_2=threadIdx.x-d_1; size_1=d_1; size_2=d_2;" to F
4 Replace "threadIdx.x" and "blockDim.x" in S_1 and S_2 with "tid_1" and "size_1" or "tid_2" and "size_2" accordingly.
5 Replace "__syncthreads()" in S_1 with the inlined PTX "bar.sync 1, d_1;"
6 Replace "__syncthreads()" in S_2 with the inlined PTX "bar.sync 2, d_2;"
7 Append "if (threadIdx.x >= d_1) goto l_1;" to F
8 Mark the end of S_1 with the label l_1
9 Append S_1 to F
10 Append "if (threadIdx.x < d_1) goto l_2;" to F
11 Mark the end of S_2 with the label l_2
12 Append S_2 to F
13 **return** F

Fig. 5: Fused kernel generation algorithm.

of the two input kernels and high register usage may lower the occupancy. Enforcing a register bound in CUDA may improve the performance of the fused kernel.

HFUSE automatically profiles possible configuration combinations. For Pascal 1080Ti GPU and the default workload of these two original kernels in our experiments, HFUSE outputs the kernel in Figure 4 as the fastest fused kernel and restricts the register usage to 32 per thread. The kernel in Figure 4 runs 53.4% faster than individually executing two kernels in Figures 2 and 3 on 1080Ti. For Volta V100 GPU, the fastest fused kernel partitions the thread space differently. It assigns 768 threads instead of 896 threads for the first kernel and the remaining 256 threads to the second. It runs 15.8% faster than individually executing two kernels on V100.

III. DESIGN

We next present the design of HFUSE. In this section, we represent a kernel as a list of CUDA statements. Macros are preprocessed, function calls are all inlined, and local variable declarations are lifted to the top of the function[3]. In pseudo-codes, we use double quotations to denote CUDA statements. For simplicity, in this section we assume that the CUDA kernels have only one block sub-dimension, i.e., `blockDim.y` and `blockDim.z` are one. It is straightforward to extend our algorithm to more than one block sub-dimensions.

A. Generate Fused Kernel

Figure 5 presents the pseudo-code of Generate(). Given two kernels K_1 and K_2 together with their block dimensions

[3]The variable declaration lifting is not required but it simplifies the implementation, because goto statements cannot jump over variable declarations.

d_1 and d_2, Generate() returns the horizontally fused kernel F.

Generate Prologue: The pseudo-code in Figure 5 first copies the local variable declarations from the two input kernels to the fused kernels at line 2. It properly renames them to make sure that the local variables do not have conflict names. At line 3 the pseudo-code defines and initializes a set of special variables, `tid_1` and `tid_2` for storing the original thread id of the two input kernels as well as `size_1` and `size_2` for storing the original block dimension of the two kernels.

Replace Built-in Variables: The pseudo-code at lines 4 then replaces "`threadIdx.x`" and "`blockDim.x`" with the corresponding defined variables in the prologue. This is because in the fused kernel, these built-in values will refer to the fused kernel not the original kernels. This replacement preserves the statement semantics in the original kernels.

Replace Synchronization Barriers: `__syncthreads()` in CUDA implements a barrier for all threads in a block. In the fused kernel, the instructions from the two input kernels are running concurrently in different threads of a block, so HFUSE needs to replace `__syncthreads()` with partial barriers only for the threads of the corresponding kernel.

Fortunately, the inlined PTX instruction `bar.sync` can support partial barrier [23]. The first parameter of `bar.sync` is a constant from 0 to 15 denoting the barrier id. The second parameter of `bar.sync` is a constant denoting the number of threads participating the barrier. Internally, the GPU hardware maintains a counter to track how many threads have reached the barrier. When sufficient threads have reached the barrier, they are allowed to progress. The pseudo-code at lines 5-6 replaces `__syncthreads()` with `bar.sync` PTX instructions. These instructions pass the barrier id one for barriers in the first original kernel and two for barriers in the second kernel. They also pass the original block dimension as the second parameter to implement the desired partial barriers. When combined with the branch guards inserted at lines 7 and 10, these `bar.sync` instructions will only wait for threads from their own original kernels instead of for all threads. The fused kernel therefore has synchronization barriers at equivalent places for the equivalent sets of threads as the original two kernels.

Append Guarded Statements: The pseudo-code finally appends the translated statements of two input kernels into the fused kernel at lines 7-12. Before appending the statements of each kernel, HFUSE will insert an if statement to check the current thread index at lines 7 and 10. In the fused kernel, the threads in the index range of $[0, d_1)$ correspond to the first input kernel, while the threads in the index range of $[d_1, d_1 + d_2)$ correspond to the second input kernel. If the index is outside the range of the corresponding input kernel, it will skip the statements from the kernel.

B. Search Fusion Configuration

Figure 6 presents the pseudo-code of our main algorithm to search for the best fusion configuration. Given statements from two kernels S_1 and S_2 and the desired block dimension

Input : K_1 and K_2 are two different kernels. d_0 is the desired block dimension of the fused kernel.
Output : A fused kernel $F*$ and the register bound $r*$ for launching the kernel.

1 function $Main(K_1, K_2, d_0)$:
2 $\langle t^*, F^*, r^* \rangle \leftarrow \langle \infty, \emptyset, \perp \rangle$
3 $d_1 \leftarrow 128$
4 **while** $d_1 < d_0$ **do**
5 $F \leftarrow \text{Generate}(K_1, K_2, d_1, d_0 - d_1)$
6 $t \leftarrow$ Profile the running time of F
7 **if** $t < t^*$ **then**
8 $\langle t^*, F^*, r^* \rangle \leftarrow \langle t, F, \perp \rangle$
9 $b_1 \leftarrow \frac{\text{SMNRegs}}{d_1 * \text{NRegs}(S_1)}$
10 $b_2 \leftarrow \frac{\text{SMNRegs}}{d_2 * \text{NRegs}(S_2)}$
11 $b_0 \leftarrow \min(\min(b_1, b_2), \frac{\text{SMShMem}}{\text{ShMem}(F)}, \frac{\text{SMNThreads}}{d_0})$
12 $r_0 \leftarrow \frac{\text{SMNRegs}}{b_0 * d_0}$
13 $t \leftarrow$ Profile F with the register bound r_0
14 **if** $t < t^*$ **then**
15 $\langle t^*, F^*, r^* \rangle \leftarrow \langle t, F, r_0 \rangle$
16 $d_1 \leftarrow d_1 + 128$
17 **return** F^*, r^*

Fig. 6: Configuration search algorithm.

of the fused kernel d_0, the algorithm produces a horizontally fused kernel kernel F^* as its output.

Thread Space Partition: The pseudo-code uses a loop at lines 4-16 to search for the best thread space partition. At each iteration, it tries a different block dimension for the first kernel (i.e., d_1), generates the fused kernel at line 5, and profiles the running time of the fused kernel twice, once without any register bound at line 8 and once with a calculated register bound at line 12. At lines 8 and 15, the pseudo-code records the fastest fused kernel together with its configuration. Note that HFUSE searches the block dimension of the first kernel at a granularity of 128, because using an irregular block dimension often breaks memory access patterns and causes CUDA kernels to run slower.

Limit Register Usage for Occupancy: The fused kernel may require more registers than each of the original two kernels. This additional register requirement may lower the occupancy, each SM will be able to execute less blocks concurrently due to the available total registers per SM. In practice, the CUDA compiler can enforce a bound to limit the number of registers used in a compiled kernel. Excessive registers will be spilled into the global GPU memory. It is therefore possible to recover the occupancy loss at the cost of introducing expensive memory instructions.

The pseudo-code in Figure 6 automatically explores this trade-off with profiling. For each different thread space partition, HFUSE will attempt to compile the fused kernel twice with different configurations, one without the register bound and one with it. When the algorithm sets the bound, it computes the bound r_0 at lines 9-12. Note that SMNRegs is the number of

registers per SM (64K for Pascal and Volta GPUs), $\mathrm{SMShMem}$ is the shared memory size per SM (96K for Pascal and Volta GPUs), $\mathrm{SMNThreads}$ is the maximum number of concurrent threads per SM (2048 for Pascal and Volta GPUs), b_1 and b_2 are the numbers of concurrent active block while launching the original two kernels, $\mathrm{ShMem}()$ denotes the used shared memory size of a kernel, and $\mathrm{NRegs}()$ denotes the number of used registers of a kernel. Note that HFUSE obtains the shared memory size and the number of used registers of a kernel from the output of NVIDIA CUDA compilers. The intuition is to make the fused kernel to run as many blocks per SM as the two input kernels, unless the occupancy is otherwise bounded by the number of threads or the shared memory usage.

C. Implementation

We implemented HFUSE[4] based on the front-end CUDA parser of the LLVM Clang framework [24]. For each input kernel file, our implementation uses Clang to pre-process all macros and included headers. We use clang-expand [25], an open-sourced tool built on Clang for source code refactoring, to inline function calls in the input kernel functions. Additionally, HFUSE traverses the AST of the input kernel to locate all local variable declarations. It renames each local variable to make sure that they will not cause name conflicts in the fused kernel. It also lifts their declarations to the start of the kernel.

IV. Experimental Results

We next evaluate HFUSE with five deep learning computational kernels and four cryptography kernels. The goal of this evaluation is to answer the following questions: 1) How effective is HFUSE? 2) Why do the horizontal fused kernels run faster? 3) What is the right scenario to apply the horizontal fusion? 4) How much improvement does the automatic profiling technique have on fusing kernels with barriers?

A. Methodology

Benchmark Kernels: We collect nine GPU kernels including five deep learning computational kernels and four cryptography computational kernels: Maxpool applies a 2D maxpooling over an input matrix; Batchnorm collects the batch mean and variance a 2D input matrix, which will then be used for normalization; Upsample applies a 2D bilinear upsampling over a input matrix; Im2Col rearranges the input image blocks into columns; Hist computes the histogram of an input matrix. These kernels have been widely used in AI models such as ResNet [21], BigGAN [26], and UVC [27]. Ethash is a memory intensive hash function used by Ethereum [28] for its proof of work mining. SHA256, Blake256, and Blake2B are three computational intensive hash functions used for the proof-of-work of several cryptocurrencies.

All deep learning computational kernels are extracted from PyTorch [16]. Ethash is extracted from ethminer [17], and the rest three cryptography kernels are collected from ccminer [18]. To minimize the bias, we evaluated HFUSE on all pairs of kernels we collected in the deep learning and cryptography domains. The five deep learning kernels form ten possible benchmark pairs, while the four crypto kernels form six possible benchmark pairs. All deep learning kernels support tunable block dimensions while crypto kernels do not.

Apply HFUSE**:** For each benchmark pair, we apply HFUSE to horizontally fuse the two kernels. We run the fused kernel and measure its running time. For comparison, we measure the running time of launching the original kernels individually via parallel CUDA streams. We implement the standard vertical fusion and compares its running time with HFUSE as well. To evaluate the effect of our profiling techniques, we also run a version of HFUSE that evenly partition the thread space for two kernels without profiling. Note that because crypto kernels do not support tunable block dimensions, HFUSE always evenly partition the space instead. We use nvprof, a profiling tool provided by CUDA toolkit, to collect the performance data of each kernel. In all experiments we take into account two generations of NVIDIA GPU cards: GeForce GTX 1080 Ti graphic card based on Pascal, and Tesla V100 graphic card based on Volta. We run our experiments with CUDA Toolkit version 10.02 and LLVM toolchain version 9.0.0.

Run Different Workload: All benchmark kernels can operate with variable workload. Deep learning kernels can process inputs with different sizes, while cryptography kernels can run iterations to compute multiple hashes. The speed up of any fusion technique depends on the execution time ratio of two input kernels, i.e., fusing two kernels with similar execution time will be typically more beneficial. To understand how horizontal fusion works in different workload ratio, we run our experiments with different input sizes for each kernel. For each benchmark pair, we will report the speed up under different execution time ratios of the two original kernels.

Execution Time Measurement: Since it may take a while for the GPU performance to stabilize, we launch a dummy kernel on the GPU for about 500 millisecond before launching any experimental kernels. For each pair of kernels, we record elapsed time after the first kernel launches and before the second kernel finishes with nvprof as the native execution time.

Performance Analysis: For each benchmark kernel, we select a representative input size so that the execution time ratios of the ten benchmark pairs are close to one. We use nvprof to collect three metrics besides its execution time:

- **Issue Slot Utilization:** Percentage of issue slots that issued at least one instruction. The streaming multiprocessor is stalled because of instruction latencies.
- **MemInst Stall:** Percentage of stalls caused by waiting for memory instructions.
- **Occupancy:** Ratio of the average active wraps per active cycle to the theoretically number of warps supported on a multiprocessor.

B. Performance Results

Figure 7 shows the kernel execution time speedup with respect to the native execution of 16 pairs of kenels. In each subplot, the x-axis represents the ratios of execution time of

[4]https://github.com/aoli-al/HFuse

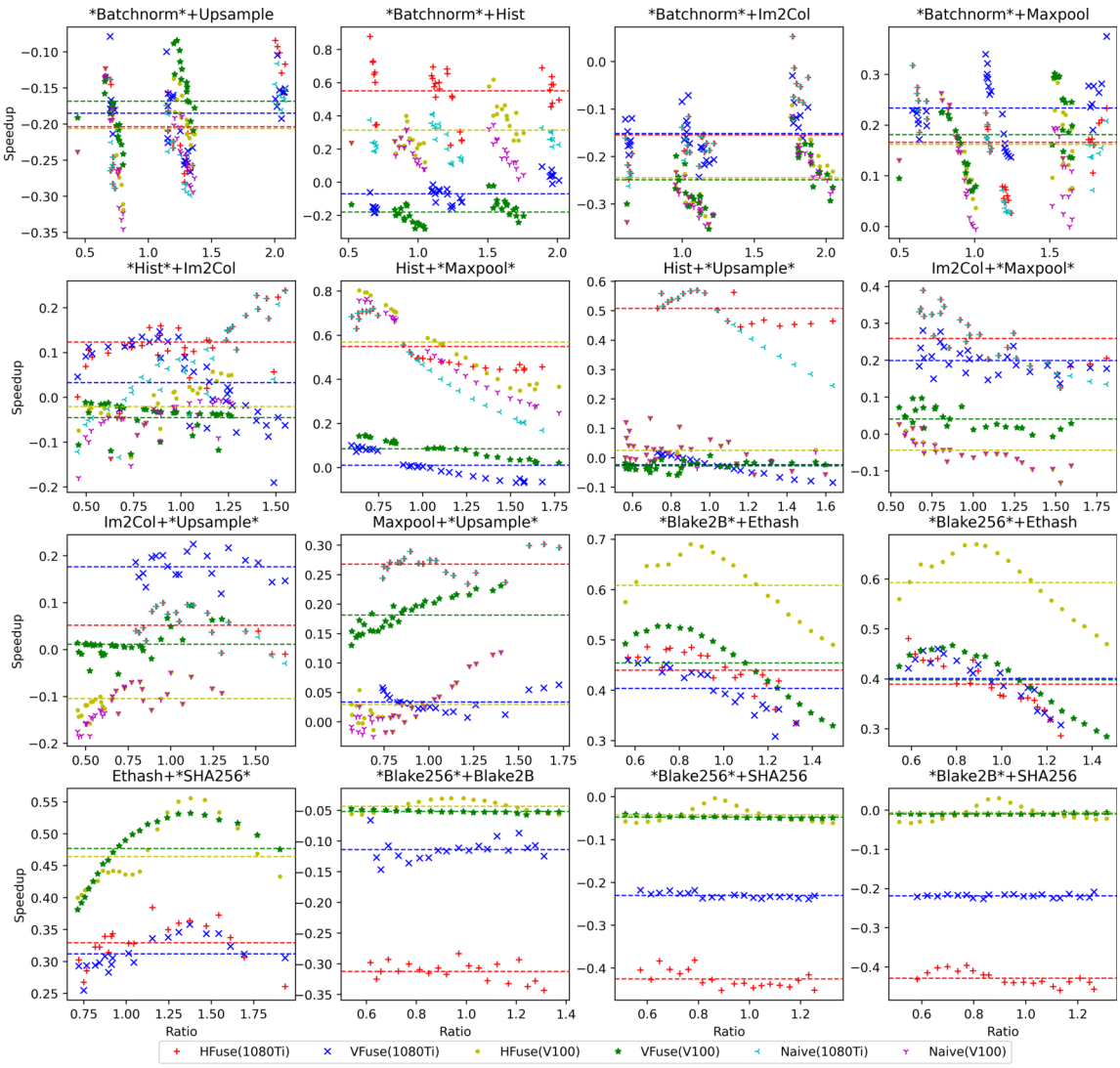

Fig. 7: Kernel execution time speedup.

two kernels; the y-axis represents the speedup of the fused kernel with resepct to the native execution. Each subplot has four kinds of markers which represent standard fusion (VFuse) and horizontal fusion (HFuse) on two different GPU generations (1080Ti and V100). For deep learning kernels, we also include two additional kinds of markers to represent horizontal fusion without thread space profiling (Naive) on the two GPU generations.

For each benchmark pair, we change the input size of one benchmark kernel (marked with "*" in the pair name in Figure 7) to obtain results on different execution time ratios of the two kernels. Different marks of the same kind correspond to experimental results of different input sizes. Each subplot in Figure 7 also draws four horizontal lines in the corresponding color to represent the average speedup of the fused kernel across different execution ratio data points. Note that the execution time of Batchnorm changes non-continuously as its input size changes, so the marks in the four pairs involving Batchnorm appear in clusters.

Our results highlight the effectiveness of our automatic horizontal fusion technique across two different domains. For five out of the ten deep learning cases (*Batchnorm*+Hist, *Batchnorm*+Maxpool, Hist+*Maxpool*, Hist+*Upsample*, and Maxpool+*Upsample*) and for three out of the six crypto cases (pairs with Ethash), the HFUSE fused kernel outperforms the native execution across different execution time ratios on both 1080Ti and V100. For these cases, the HFUSE marks are almost always on the positive side of the y-axis. The average speedup of HFUSE over different execution time ratios on these nine cases are 12.4%-55.1% on 1080Ti and 2.5%-60.8% on V100. For eight cases on 1080Ti and five cases on V100 (out of the total 16 cases), the HFUSE fused kernel on average over different execution time ratios outperforms both the standard fusion and the native execution.

TABLE I: Metrics of individual kernels.

Kernel	Execution Time (ms)	Issue Slot Utilization (%)	MemInst Stall (%)	Occupancy (%)
Im2Col	1.92 / 1.69	87.18 / 63.81	27.5 / 38.2	48.0 / 48.1
Maxpool	1.93 / 1.97	7.99 / 8.55	95.2 / 97.2	89.5 / 92.2
Upsample	1.72 / 2.41	34.32 / 23.29	77.8 / 81.3	48.3 / 49.7
Hist	1.70 / 1.90	14.46 / 50.70	1.4 / 7.3	99.0 / 74.3
Batchnorm	2.15 / 1.90	61.83 / 63.27	52.2 / 60.3	96.2 / 98.1
Blake256	38.43 / 37.33	91.01 / 53.22	1.3 / 0.0	48.9 / 48.9
Blake2B	39.20 / 39.40	90.15 / 52.44	1.7 / 0.0	49.1 / 48.9
SHA256	42.88 / 39.69	65.62 / 49.07	0.0 / 0.0	30.6 / 24.6
Ethash	46.01 / 37.23	10.88 / 4.17	96.1 / 96.6	36.7 / 18.2

We observe that the HFUSE fused kernels have more significant speedup on those cases where one of the original kernels is memory intensive. This is because the horizontal fusion interleaves memory instructions of such a kernel with other instructions to hide latencies of these expensive memory instructions. Note that both kernel fusion techniques perform badly on *Blake256*+Blake2B, *Blake256*+SHA256, and *Blake2B*+SHA256, because those cryptography computational kernels require similar computational resources and fusing such kernels together will bring little benefits but harm the occupancies.

Our results also show the importance of the thread space partition. The automatic profiling technique enables HFUSE to better fuse kernels that have different execution times. For all deep learning cases except *Batchnorm*+Im2Col, the thread space profiling technique is able to find a thread space partition scheme that performs better than the naive approach for some execution time ratio. Better partitioning the thread space will make threads for the two original kernels to co-exist longer so that the warp scheduler can better interleave their instructions.

Volta vs. Pascal: HFUSE performs better on Pascal (i.e., 1080Ti) than on Volta (i.e., V100). One reason is that Volta handles instruction latencies better. Volta architecture reduced the execution latency of math operations to four cycles according to the documentation [29]. Another reason is that none of our benchmark kernels utilize the new tensor core functionalities of Volta. We expect the results will be different with tensor core computations because tensor core instructions have much higher performance than regular floating-point instructions. They will push the bottleneck toward memory latencies which HFUSE can address with horizontal fusion.

Fusing More Than Two Kernels: It is straightforward to extend HFUSE to fuse more than two kernels. We implemented a prototype of HFUSE that is capable of fusing three kernels. For our benchmark kernels, fusing three kernels does not provide better performance than fusing just two out of the three. One possible explanation is that the horizontal fusion is most beneficial when fusing a compute-intensive kernel together with a memory-intensive kernel. Fusing one additional kernel often reduces the occupancy with little latency benefit.

C. Kernel Metrics Results

To understand in what scenario HFUSE performs best, we collect the performance metrics of the original kernels and the HFUSE fused kernel variants under a representative workload in which the execution time of benchmark kernels is close to each other. Table I shows the results of individual kernels. In Appendix, Table II shows the results of the fused kernels. Each row corresponds to the metrics of one kernel. Note that the issue slot utilization denotes the percentage of GPU cycles that at least one warp is active for a kernel (that SMs are not stalled due to instruction latencies). Typical reasons of the stalls are memory instructions and/or insufficient occupancies of the kernels.

Issue Slot Utilization: Our results indicate that HFUSE is effective because horizontal fusion interleaves instructions to hide the instruction latencies. For all cases, a fused kernel runs faster than the native execution if the fused kernel has a higher issue slot utilization. On one hand, the fused kernel will have a much better performance if two kernels use two different computational resources. For example, Ethash is a memory intensive kernel and Blake256 is a compute intensive kernel. As shown in Table I and Table II, the percentage of stalls caused by waiting for memory instructions of Ethash is 96.1% on 1080Ti GPU. The percentage of Blake256 is only 1.3%. Therefore, the issue slot utilization of the fused kernel of Ethash and Blake256 is 23.9% higher than the native execution. The fused kernel hides high latency of the memory instructions in Ethash by interleaving computation instructions from Blake256. On the other hand, fusing two compute-intensive kernels is not very beneficial, as shown by the Blake256+Blake2B, Blake256+SHA256, and Blake2B+SHA256 cases.

Thread-level v.s. Block-level Parallelism: The horizontal fusion may lower the occupancy, which is another key factor that influence the performance. Occupancy indicates the ratio of the average active wraps per active cycle to the theoretically number of wraps supported on a Stream Multiprocessor (SM). If the number of blocks which can execute concurrently on an SM is low, the occupancy of the kernel will also be low because there are not enough eligible warps to be launched. The horizontal fusion may increase the number of registers per thread of the fused kernel, which may limit the maximum number of active blocks on an SM.

Therefore one could view the horizontal fusion as a technique to navigate the inherent trade-off between the thread-level parallelism of interleaving instructions from more threads and the block-level parallelism of running more blocks per SM. Our results show that it is often beneficial to apply horizontal fusion to gain thread-level parallelism even at the cost of block-

level parallelism. For cases including Batchnorm+Maxpool and Hist+Maxpool, the fused kernels have lower occupancies on 1080Ti and V100 GPUs than both of the corresponding original kernels but they run faster.

Register Bound: The register bound may recover the occupancy loss at the cost of additional memory instructions for spilled registers. Our results show that the fused kernel with the register bound may perform better than the kernel without it. As shown in Table II, Hist+Upsample only achieves 38.5% occupancy without a bound, but it achieves 77.6% occupancy with the bound on 1080Ti GPU. Because of the large improvement of the occupancy, for this case the version with the register bound runs significantly faster. We also noticed that the register bound may cause register spilling and increase the percentage of stalls caused by memory instructions. The MemInst Stall of Im2Col+Upsample is 43.0% on 1080Ti without a bound, and the number increases to 73.6% when the kernel is launched with the bound. Because of the cost of spilled registers, for this case the version without the register bound runs faster. Fortunately, HFUSE automatically profiles two different versions to decide whether to set the bound.

V. Related Work

Kernel Fusion: Wang et al. proposed three different strategies to fusion including concatenating the computation of two kernels similar to the standard vertical fusion, and the distribution of the computation among different threads similar to horizontal fusion [11]. However, their proposed technique cannot handle barriers and it is not automated. Due to these limitations, their results show that distributing computation among different threads is the worst fusion strategy out of the three proposed fusion strategies. In contrast, our technique do not have these limitations. Our results show that the horizontal fusion, after appropriately handling barriers and automatically profiling the best thread space partition scheme, often outperforms the vertical fusion.

Rammer adopts a different fusion policy that attempts to fit instructions of input kernels into different blocks of the fused kernel [30]. It exploits block-level parallelism, while HFUSE exploits thread-level parallelism. One limitation of the fusion policy of Rammer is that it may require heavy rewriting of the existing operators so that all operators have the same block dimension and the execution time of each fused operator is roughly the same. For example, in Rammer, the authors use 32x32 as the tile size for their GPU matrix multiplication, whereas 128x128 is commonly considered to be the optimal. In contrast, HFuse automatically fuses existing GPU kernels directly without any manual modification and the execution time of the two original kernels can be different.

Wen and O'Boyle implemented a JIT compiler which is able to fuse two OpenCL kernels automatically and uses a model based approach to generate fusion configurations automatically [31]. In contrast, HFUSE works on NVIDIA CUDA programs instead of OpenCL. HFUSE uses a deterministic algorithm to find the best fusion configuration for the fused kernels. Another difference is that HFUSE is able to handle barriers but the approach of Wen and O'Boyle cannot. Many machine learning kernels have barriers and we believe HFUSE is more applicable to the machine learning domain.

There is a rich set of previous work that targets automatic vertical fusion. Fousek et al. presents a searching technique that finds a linearized kernel with lowest memory requirement [12]. Wahib and Maruyama proposes to formalize kernel fusion as an optimization problem [13]. Springer et al. proposes a new language, called Ikra, for efficient GPU programming that allows a programmer to implement GPU programs of multiple reusable parallel sections [32]. Ikra then fuses those parallel sections into a small number of GPU kernels. Filipovič et al. present a source-to-source compiler that is able to automatically fuse kernels that can be expressed in the form of map and reduce calls [14]. TASO [4] is a deep neural network computation graph optimizer that fuses different matrix operators using graph substitution. All these prior works only consider vertical fusion, rather than the horizontal fusion proposed in our work.

Warp Specialization: Singe [33] and CudaDMA [34] use warp specialization techniques to speed up domain-specific applications (e.g., chemistry for Singe and direct memory access library for CudaDMA). Similar to horizontal fusion, the idea of warp specialization is to allow warps in a block to perform different tasks in parallel. Comparing to HFUSE, Singe and CudaDMA have more significant speed up but can only apply to specific domains.

Multi-Application Concurrency: Previous work also proposes techniques to enable better multi-application concurrency [35–38]. These systems modify the GPU runtime and may introduce overhead. Pai et al. proposed a kernel rewrite method, which decouples logical thread block ands physical thread blocks to minimize the leftover thread block resources. KernelMerge [38] modifies the OpenCL runtime to launch and execute two kernels concurrently, which is similar to CUDA stream parallelization. Similarly, Ausavarungnirun et al. suggests to redesign the GPU memory virtualization to mitigate address transaction overhead while supporting multi-application concurrency [39].

VI. Conclusion

Automatic horizontal fusion is an effective optimization technique that complements the standard vertical kernel fusion and it can speedup GPU programs in domains like deep learning and cryptocurrency mining. Our experimental results show that the horizontal fusion can enable warp schedulers in NVIDIA GPUs to interleave instructions from different kernels to hide instruction latencies. It is especially beneficial to apply this technique to fuse kernels with instructions that require different kinds of hardware resources.

Acknowledgments

We thank the anonymous reviewers for their insightful comments on the early draft of this paper. This research was supported by Connaught Fund #507141 and Tri-Council Bridge Funding from University of Toronto. Note that an early version of this paper appears in arXiv [40].

APPENDIX

Our research artifact provides the source code and data set of HFUSE to reproduce the experiment results. It is publicly available on github[5]. The artifact includes a docker image that contains all dependencies and data set to run HFUSE[6].

HFUSE dependens on Clang-10 and LLVM-10 and CUDA library. An NVIDIA GPU card is required to run fused kernels generated by HFUSE. Note that, HFUSE is only tested on 1080 Ti and V100, you may have different experiment results on different GPU cards. There are also library dependencies for benchamrk kernels. Specifically, fusing deep learning kernels depends on PyTorch library and fusing crypto kernels depends on ethminer library.

A. How to Use

Download: To reproduce our experimental results, download the pre-configured docker image: `docker run –rm –privileged –gpus all -it leeleo3x/hfuse:latest bash` Note that the `–privileged` flag allows `nvprof` to collect performance information of kernels. The source code of HFUSE locates in the `/root/HFuse` folder. The docker image has prebuilt the project. The program takes the configuration of fusion, the configuration of kernels, and the source directory that contains the source code of kernels as input. It generates fused kernels in current directory.

Fuse Benchmark Kernels: Run the following commands to fuse deep learning benchmark kernels:

- `cd /root`
- `mkdir fused-torch`
- `HFUSE_PARALLEL=1 ../HFuse/build/tools/llvm-smart-fuser/llvm-smart-fuser ../HFuse/configs/ml_fusion.yaml ../HFuse/configs/ml_kernels.yaml ../TorchKernel/fused/`

Then run the following commands to fuse cryptography benchmark kernels:

- `cd /root`
- `mkdir fused-crypto`
- `HFUSE_PARALLE=0 ../HFuse/build/tools/llvm-smart-fuser/llvm-smart-fuser ../HFuse/configs/crypto_fusion.yaml ../HFuse/configs/crypto_kernels.yaml ../ethminer/libethash-cuda`

Note that `HFUSE_PARALLEL` is a flag to enable parallel fusing to speed up the experimental process. It can only be enabled while fusing deep learning kernels. Fusing all benchmark kernels takes around 30 minutesi.

Inspect Fused Kernels: After running all commands, the fused deep learning kernels are stored in `/root/TorchKernel/fused` and the fused crypto kernels are stored in `/root/ethminer/libethash-cuda/four.cu`. Each fused kernel is named as `{kernel1}_{kernel2}.inc`. HFUSE automatically generates kernels with different thread space partition and launch bounds, which are distinguished by kernel name. `{kernel1}_{kernel2_hfuse_lb_idxI}` are kernels fused horizontally with register bounds. `{kernel1}_{kernel2_hfuse_idxI}` are kernels fused horizontally without register bounds. The I-th kernel allocates $(I + 1) * 128$ threads to the first input kernel and allocates the rest threads to the second one. HFUSE also generates kernels that use traditional vertical fusion e.g. `{kernel1}_{kernel2_vfuse_idx0}`.

Run Fused Kernels: To run the fused kernels, first move the fused kernels to the directory that contains kernel drivers.

- `mv /root/fused-torch/* /root/TorchKernel/fused`
- `mv /root/fused-crypto/* /root/ethminer/-libethash-cuda`

Next, compile the kernel driver. Note that we only tested the kernel drivers on 1080Ti and V100 GPUs. The compiler may fail to compile the driver if you use different GPUs. The compilation takes roughly 30 minutes.

- `cd /root/TorchKernel`
- `./build.sh`
- `cd /root/ethminer`
- `mkdir build`
- `cd build`
- `cmake ..`
- `make fuser -j4`

Collect Performance Metrics: To collect performance metrics, first run `nvprof` to get the execution profile data of each kernel. For deep learning kernels, run the following commands and the result is stored in `/root/TorchKernel/performance.csv`:

- `cd /root/TorchKernel`
- `/usr/local/cuda-11.5/bin/nvprof --csv --log-file performance.csv python3 ./call_{arch}.py`

And replace `{arch}` with the GPU you use (1080 or v100).

For cryptography kernels, run the following commands and the result is stored in `/root/ethminer/performance.csv`:

- `cd /root/ethminer`
- `/usr/local/cuda-11.5/bin/nvprof --csv --log-file performance.csv ./build/fuse/fuser`

Then to collect the metrics of deep learning kernels, run the following commands and the result is stored in `/root/TorchKernel/metrics.csv`:

- `cd /root/TorchKernel`
- `/usr/local/cuda-11.5/nvprof --csv --log-file metrics.csv --events "elapsed_cycles_pm" --metrics "issue_slot_utilization,achieved_occupancy,stall_memory_dependency" python3 call_{arch}.py`

To collect the metrics of cryptography kernels, run the following commands and the result is stored in `/root/ethminer/metrics.csv`:

- `cd /root/ethminer`
- `/usr/local/cuda-11.5/nvprof --csv --log-file metrics.csv --events "elapsed_cycles_pm"`

[5]https://github.com/aoli-al/HFuse

[6]https://hub.docker.com/r/leeleo3x/hfuse

TABLE II: Metrics of HFUSE fused kernels.

Pairs	Type	Speedup (%)	Issue Slot Utilization (%)		MemInst Stall (%)	Occupancy (%)
			HFUSE	Native		
Batchnorm+Upsample	N-RegCap	-35.0 / -37.4	32.05 / 25.93	52.29 / 41.69	67.2 / 76.1	42.9 / 42.9
	RegCap	-23.4 / -28.9	38.35 / 29.73		75.0 / 80.7	87.1 / 83.2
Batchnorm+Hist	N-RegCap	51.2 / -28.5	55.89 / 33.43	40.55 / 57.22	45.6 / 49.7	90.7 / 43.0
	RegCap	53.4 / 15.8	56.74 / 53.33		46.1 / 56.3	91.1 / 96.0
Batchnorm+Im2Col	N-RegCap	-31.1 / -42.8	44.43 / 31.55	73.65 / 64.81	50.8 / 67.6	37.3 / 42.0
	RegCap	-12.7 / -31.1	58.13 / 41.55		63.4 / 74.5	92.0 / 85.3
Batchnorm+Maxpool	N-RegCap	7.8 / 3.4	32.58 / 28.41	35.73 / 35.28	67.3 / 78.2	64.1 / 72.7
	RegCap	7.8 / 3.4	32.58 / 32.20		67.5 / 78.2	64.0 / 72.7
Hist+Im2Col	N-RegCap	12.2 / -17.3	60.34 / 40.69	51.90 / 58.03	20.0 / 31.6	38.1 / 37.4
	RegCap	11.3 / -0.1	60.09 / 51.32		19.8 / 46.1	38.3 / 78.6
Hist+Maxpool	N-RegCap	52.5 / 56.6	19.07 / 36.25	11.05 / 28.58	26.7 / 43.5	67.5 / 59.0
	RegCap	53.4 / 57.1	19.10 / 39.07		25.0 / 43.0	67.7 / 57.5
Hist+Upsample	N-RegCap	4.4 / -5.7	30.17 / 35.01	26.87 / 35.70	40.8 / 41.2	38.5 / 43.5
	RegCap	51.4 / 5.7	41.20 / 36.37		48.0 / 57.0	77.6 / 82.6
Im2Col+Maxpool	N-RegCap	7.0 / -12.0	51.52 / 30.47	45.95 / 34.05	54.7 / 69.3	32.6 / 32.5
	RegCap	25.3 / -7.5	57.87 / 33.87		62.5 / 74.4	63.5 / 58.4
Im2Col+Upsample	N-RegCap	5.4 / -10.8	71.92 / 36.72	64.76 / 41.11	43.0 / 72.2	42.7 / 44.9
	RegCap	-24.1 / -45.5	49.50 / 24.24		73.6 / 78.9	73.7 / 74.0
Maxpool+Upsample	N-RegCap	-1.6 / -3.4	23.39 / 16.18	22.47 / 17.00	79.3 / 86.4	30.0 / 33.1
	RegCap	29.4 / 1.1	30.32 / 17.97		81.0 / 88.3	60.9 / 62.3
Blake2B+Ethash	N-RegCap	15.9 / 30.1	58.93 / 36.73	47.39 / 28.29	22.8 / 25.5	15.8 / 8.6
	RegCap	42.9 / 65.8	70.08 / 46.85		19.8 / 23.9	29.0 / 29.2
Blake256+Ethash	N-RegCap	17.0 / 30.3	57.89 / 37.05	47.49 / 28.46	19.5 / 26.5	16.1 / 8.6
	RegCap	47.4 / 64.7	71.41 / 46.79		17.6 / 24.7	29.3 / 29.3
Ethash+SHA256	N-RegCap	8.8 / 37.0	39.25 / 36.62	36.97 / 26.81	10.3 / 26.8	15.7 / 8.8
	RegCap	35.1 / 44.1	50.51 / 39.37		18.4 / 16.5	28.8 / 28.8
Blake256+Blake2B	N-RegCap	-26.5 / -2.7	66.08 / 51.34	90.58 / 52.82	2.3 / 0.0	37.5 / 36.8
	RegCap	-96.5 / -96.1	3.60 / 3.31		72.0 / 62.4	98.4 / 96.0
Blake256+SHA256	N-RegCap	-44.3 / -1.0	41.13 / 50.22	77.81 / 51.11	0.9 / 0.0	22.7 / 24.4
	RegCap	-51.2 / -37.4	42.57 / 34.32		43.0 / 7.7	56.2 / 51.6
Blake2B+SHA256	N-RegCap	-42.9 / 2.8	41.40 / 50.26	77.49 / 50.74	0.8 / 0.0	22.7 / 24.5
	RegCap	-50.9 / -31.7	38.27 / 35.30		48.0 / 7.6	54.9 / 50.6

```
--metrics "issue_slot_utilization,achieved_-
occupancy,stall_memory_dependency"
./build/fuse/fuser
```

B. Expected Results

The artifact will reproduce our experimental results in Figure 7, Table I, and II with 1080Ti and V100 GPUs. Note that Similar to Table I, each entry of Table II is of the form "X / Y", where X is the result for 1080Ti GPU and Y is the result for V100 GPU. In order to understand some key factors that influence the performance of the fused kernels. We collect metrics for kernels both with register bound (RegCap) and without register bound (N-RegCap). The third column in Table II shows the speedup of the fused kernel against the native execution. The fourth column presents the instruction issue slot utilization for the fused kernels and the fifth column presents the average instruction issue slot utilization computed from the metrics of two individual kernels (from Table I).

C. Experiment Customization

It is possible to use HFUSE in the artifact to fuse fuse other kernels. To do so, an user need to provide the kernel specification, the fusion specification, and the source code of kernels. The kernel specification includes kernel dimension, number of register required by the kernel, and whether the kernel has barriers. You can set the `ExecTime` of the kernel to 1 since this field is not used. An example kernel specification can be found in `/root/HFuse/configs/ml_-kernels.yaml`, i.e., specification for deep learning kernels. The fusion specification includes the source file name of original kernels and the function name of two kernels to be fused. An example fusion specification can be found in `/root/HFuse/configs/ml_fusion.yaml`. HFUSE depends on the `compile_commands.json` file to specify how to compile the source code of input kernels. One example of the file can be found in `/root/TorchKernel/compile_commands.json`.

References

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[2] "Mlperf training v0.6 results," https://mlperf.org/training-results-0-6/.

[3] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[4] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: ACM, 2019, pp. 47–62. [Online]. Available: http://doi.acm.org/10.1145/3341301.3359630

[5] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018. [Online]. Available: http://arxiv.org/abs/1805.00907

[6] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "ngraph-he: A graph compiler for deep learning on homomorphically encrypted data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, ser. CF '19. New York, NY, USA: ACM, 2019, pp. 3–13. [Online]. Available: http://doi.acm.org/10.1145/3310273.3323047

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[8] R. Wei, V. S. Adve, and L. Schwartz, "Dlvm: A modern compiler infrastructure for deep learning systems," *ArXiv*, vol. abs/1711.03016, 2017.

[9] J. Appleyard, T. Kociský, and P. Blunsom, "Optimizing performance of recurrent neural networks on gpus," *CoRR*, vol. abs/1604.01946, 2016. [Online]. Available: http://arxiv.org/abs/1604.01946

[10] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, "Persistent rnns: Stashing recurrent weights on-chip," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, pp. 2024–2033. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045390.3045604

[11] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*, Dec 2010, pp. 344–350.

[12] J. Fousek, J. Filipovič, and M. Madzin, "Automatic fusions of cuda-gpu kernels for parallel map," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 98–99, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2082156.2082183

[13] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound gpu applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 191–202.

[14] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, "Optimizing cuda code by kernel fusion: application on blas," *The Journal of Supercomputing*, vol. 71, no. 10, pp. 3934–3957, Oct 2015. [Online]. Available: https://doi.org/10.1007/s11227-015-1483-z

[15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[16] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS Autodiff Workshop*, 2017.

[17] "Ethminer, ethereum miner with opencl, cuda and stratum support," https://github.com/ethereum-mining/ethminer.

[18] "ccminer, a cuda accelerated mining application," https://github.com/tpruvot/ccminer.

[19] "Nvidia tesla p100, the most advanced datacenter accelerator ever built," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[20] "Nvidia tesla v100 gpu architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 06 2016, pp. 770–778.

[22] "Faster parallel reductions on kepler," https://devblogs.nvidia.com/faster-parallel-reductions-kepler.

[23] "Parallel thread execution isa version 6.5," https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[24] "Clang: a c language family frontend for llvm," https://clang.llvm.org/.

[25] "clang-expand," https://github.com/goldsborough/clang-expand.

[26] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," *ArXiv*, vol. abs/1809.11096, 2018.

[27] X. Li, S. Liu, S. D. Mello, X. Wang, J. Kautz, and M.-H. Yang, "Joint-task self-supervised learning for temporal correspondence," in *NeurIPS*, 2019.

[28] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[29] "Tuning cuda applications for volta," https://docs.nvidia.com/cuda/volta-tuning-guide/index.html.

[30] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, *RAMMER: Enabling Holistic Deep Learning Compiler Optimizations with Rtasks*. USA: USENIX Association, 2020.

[31] Y. Wen and M. F. O'Boyle, "Merge or separate? multi-job scheduling for opencl kernels on cpu/gpu platforms," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. New York, NY, USA: Association for Computing Machinery, 2017, p. 22–31. [Online]. Available: https://doi.org/10.1145/3038228.3038235

[32] M. Springer, P. Wauligmann, and H. Masuhara, "Modular array-based gpu computing in a dynamically-typed language," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY 2017. New York, NY, USA: ACM, 2017, pp. 48–55. [Online]. Available: http://doi.acm.org/10.1145/3091966.3091974

[33] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on gpus," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 119–130. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555258

[34] M. Bauer, H. Cook, and B. Khailany, "Cudadma: Optimizing gpu memory bandwidth via warp specialization," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[35] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 407–418. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451160

[36] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A gpu memory manager with application-transparent support for multiple page sizes," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2017, pp. 136–150.

[37] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 358–369.

[38] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent gpgpu kernels," in *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342788.2342798

[39] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," *SIGPLAN Not.*, vol. 53, no. 2, pp. 503–518, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3296957.3173169

[40] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for GPU kernels," *CoRR*, vol. abs/2007.01277, 2020. [Online]. Available: https://arxiv.org/abs/2007.01277