Western

Towards Automatic OpenMP-Aware Utilization of Fast GPU Memory

Delaram Talaashrafi, Marc Moreno Maza, Johannes Doerfert

Introduction

OpenMP supports target offloading since version 4.0. Although using the memory efficiently is essential for high performance on a GPU, there has not been much work done to automatically optimize memory transactions inside OpenMP target regions at compile time.

In this work, we develop an inter-procedural LLVM transformation to improve the performance of OpenMP target regions by optimizing memory transactions. This transformation pass effectively prefetches some of the read-only input data to the *fast* shared memory via compile time code injection. Especially if there is reuse, accesses to shared memory far outpace global memory accesses. Consequently, our method can significantly improve performance if the right data is placed in shared memory.

Background

Compiling and optimizing OpenMP programs with target offload regions has been supported by LLVM/Clang since version 11. `OpenMPOpt` is an inter-procedural optimization (IPO) pass in LLVM, which is implemented for optimizing OpenMP GPU execution. This pass is enabled by default since LLVM 11 when compiling with `O2` and `O3` options. It first runs on the module and later it runs on the call graph of the program. It uses domain knowledge about OpenMP runtime calls to better optimize the LLVM-IR of the program.

While executing, GPU threads can have access to different memory spaces. All threads across all teams have access to the *global memory*. Each team of threads has access to the *shared memory*. Finally, all threads have private *local memory*.

On modern GPUs, the global memory is off-chip with high access latency. Therefore, using the global memory efficiently and reducing the number of transactions to/from it is essential to maximize a GPU's computation capability utilization. An alternative to global memory is *shared memory* which is limited on-chip and fast memory space.

A challenge while using the GPU's shared memory is to avoid *bank conflict*. The shared memory is managed in modules of equal size or memory banks. Different memory banks can be accessed simultaneously. However, multiple threads cannot access different locations in the same bank in parallel. Therefore, having multiple threads accessing the same memory bank causes the bank conflict problem, and the accesses are then serialized.

Contributions

In this work, our focus is on the shared memory. There are two kinds of shared memory: static and dynamic. Static shared memory is used when the required size of the shared memory is known at compile time, and dynamic shared memory is used when this size is unknown at compile time.

We develop a compiler optimization technique to improve the performance of OpenMP programs containing device offloading regions by *automatically prefetching parts of the required data to the shared memory through code injected at compile time*. In the current version of OpenMP, runtime functions and directives exist to explicitly allocate and use memory in the shared space. Also, the `OpenMPOpt` pass, developed as a part of the LLVM framework, implements different OpenMP-aware optimization techniques that utilize shared memory. These have proven to effectively improve the performance of a program's target regions.

We leverage the `OpenMPOpt` pass infrastructure and the LLVM/OpenMP GPU runtime functions for allocating (dynamic) shared memory for our own optimization. By identifying suitable candidate memory regions and prefetching them into the shared memory buffer automatically, we can improve the program's performance as each original load from the global memory is now significantly faster served from shared memory instead.

Problem instance and assumptions

Figure 1 shows a supported kernel, and an eligible read access for prefetching (`v1`).

```
1 #pragma omp target teams map(to:v1[0:N*M])
2   #pragma omp distribute parallel for
3     // work-sharing loop
4     for (int i=0; i<N; i++)
5       for(int j=0; j<N; j++)
6         // access loop
7         for(int k=0; k<M; k++)
8        // eligible access for prefetching
9           sum += v1[i*M+k] * 3;
```

Figure 1. Example of the supported read access.

We also assume there are no conditional branches in the target region, and the total number of available threads (number of teams multiplied by the number of threads per team) is equal to the number of iterations of the work-sharing loop. Furthermore, we assume the amount of shared memory usage per team does not exceed the shared space allocated for the program.

Prefetching to the shared memory

The first step is to find global memory locations of the array, read by each team. Memory locations accessed in each iteration can be computed by: $(\mathtt{Base_i} + \mathtt{k} \times \mathtt{Step_i}), 0 \leq \mathtt{k} < \mathtt{Number_i}$. Figure 2 shows a matrix and the values of $\mathtt{Base_i}$

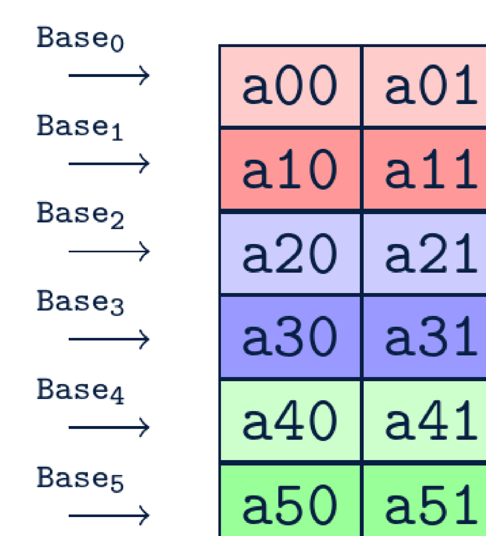

Figure 2. Values of $\mathtt{Base_i}$

The next step is to put locations to consecutive locations in the shared memory. Figure 3 shows the shared memory locations after prefetching.

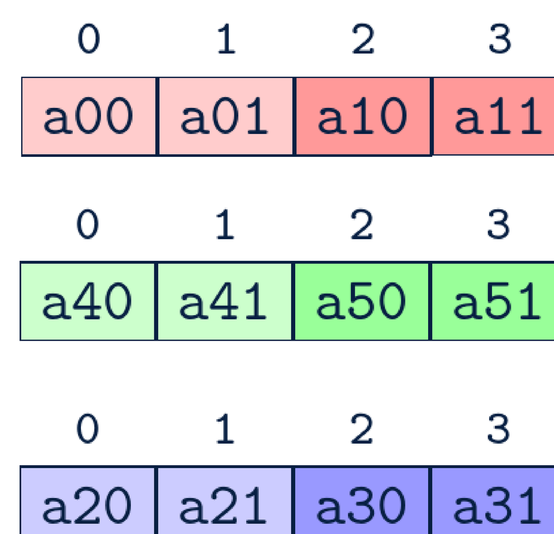

Figure 3. Shared memory locations after prefetching.

Replace accesses

The final step is to replace accesses to the global memory with accesses to the shared memory. Figure 4 shows the replacements.

```
A[0][0] → A_sh[0]
A[0][1] → A_sh[1]
A[1][0] → A_sh[2]
A[1][1] → A_sh[3]
```

Figure 4. Replacement of global memory locations with the shared buffer.

Evaluation

Figure 5 shows the speed-up we gain by using prefetching method, combined with padding technique for matrix multiplication, considering different sizes.

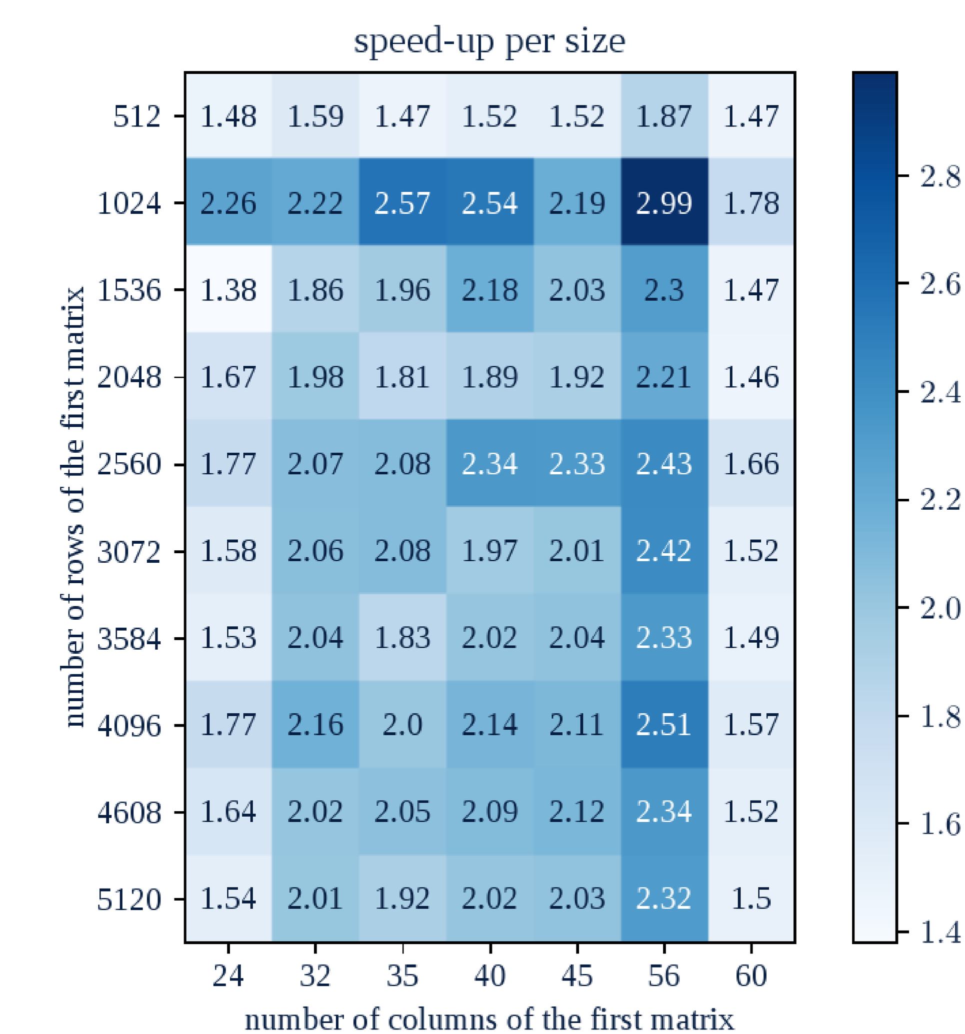

Figure 5. Speedup gains with prefetching.