Software Prefetch for Accelerating GPU Programs

Shreyas Chandrashekaran
University of Michigan
shreyasc@umich.edu

Aakash Patel
University of Michigan
aakashdp@umich.edu

Sawan Patel
University of Michigan
sawanpa@umich.edu

1 Introduction

Many modern computing systems rely on accelerators to achieve lofty performance and energy efficiency goals. Graphics processing units (GPUs) are the most commonly used accelerator which leverage thread-level parallelism for a variety of scientific computing and machine learning applications. The ability to perform a section of computation on a GPU is referred to as *GPU offloading*.

There are many programming models for GPU programming, including CUDA and OpenMP. CUDA is much more popular but requires more low-level hardware knowledge, as the programmer must specify the parallel execution configuration such as the block and thread dimensions as well as the overall kernel launch. In constrast, OpenMP requires the programmer to provide only high-level compiler directives about which regions of the code to target for parallelization, which are then implemented automatically by the compiler. Compared to CUDA, OpenMP has a much shorter learning curve and more portable perfomance [3].

Despite this, writing efficient programs leveraging GPU offloading is non-trivial in OpenMP, as the issue of managing GPU memory transfers remains the same. Though GPUs are highly optimized for parallel computation, memory transfer tends to become the performance bottleneck, and poorly managed data transfers can cause significant latencies if not done with care. Additionally, GPU performance is also limited by memory bandwidth which is a significant factor when working with large volumes of data as in the majority of scientific programming.

Unified Memory aims to solve these issues by abstracting the data transfer away from the programmer and into the hardware. Unified memory provides a single unified virtual address space between CPU and GPU memory. Data transfer is done automatically on-demand, and can even allow GPU offloading with data that is too large to fit into GPU memory. However, there are still limitations to unified memory, particularly in cases where there is a large data volume with low access rates.

In this work, we investigate the use of GPU data prefetching to improve the performance of unified memory in this context. We introduce a compiler-assisted, automated GPU prefetching scheme using LLVM and OpenMP for the identification of fixed-strided memory accesses across input objects and an experimentally-determined prefetch distance indicating the memory address which is loaded ahead of time rather than being loaded in *prefetch distance* iterations. Our experiments suggest that prefetching has potential to improve GPU program performance in certain scenarios but does not unilaterally improve performance across all benchmarks, motibating the need for more sophisticated analysis and profile-guided optimization.s

2 Related Works

Several other works have explored unified memory data management in hardware accelerators and software prefetching. [2] explores the the performance benefits of the OpenMP unified memory in addition to designing a compiler-runtime collaborative method to optimize Unified Memory performance. Their pass is implemented in Clang and the LLVM/OpenMP runtime and performs a requisite data mapping for each input object depending on object access density, size and reuse. In their initial analysis of unified memory, they demonstrate that unified memory outperforms the traditional approach for working sets fitting into GPU memory and particularly suffer when working sets are oversubscribing and have a data reuse above some threshold. With their combined compiler-runtime approach, their pass optimizes several benchmarks collected from the Rodinia

Suite in comparison to the traditional approach and the unoptimized Unified Memory approach.

Jamilan et al. [1] address the limitations of traditional data retrieval methods such as deeper caches, compile-time data locality optimizations, etc. by implementing a novel data pre-fetching scheme that can identify irregular memory access patterns. If accurately performed, data pre-fetching can hide the significant number of cache misses and ensuing memory access latency present in the majority of modern applications. Their prefetching scheme additionally does not require impractical on-chip metadata storage. Additionally, they demonstrate why state-of-the-art data prefetching mechanisms fall short of optimal performance because they fail to prefetch blocks in a timely manner due to the lack of dynamic information present (e.g. execution time of optimized code). Their pass, *APT-GET*, realizes these optimal execution times by identifying the ideal prefetch distance and prefetch insertion point which are determined through a profiling method to achieve an execution speedup of 1.30x.

3 Methods

We develop an LLVM pass that identifies all fixed-strided memory access across functions in the input code and inserts prefetch instructions for subsequent accesses. To do this, we (1) identify all loops in the function, (2) find all load instructions in the loop, (3) determine whether the load address is an add recurrence on the loop induction variable, (4) Compute the address of the memory location to prefetch following `NumPrefetchIters` iterations, (5) check if the address has already been prefetched, and (6) insert the prefetch load instruction into the IR.

At a high level, we first aim to identify all BIVs. A BIV is a basic induction variable, or a variable that is a part of a recurrence relation and whose value is incremented by a fixed value typically across loop iterations. A common example is the following:

```
for (unsigned int i; i < N; i++) {
    A[i] += 1;
}
```

In this example, i is a BIV because it is incremented by a fixed value in each iteration of the for-loop. We do not handle the case of derived induction variables, or a variable whose increment is a function of another variable that itself is BIV. BIV's are very commonly present in any loop or strided memory access, particularly in the context of scientific computations that are often offloaded to a GPU. As such, our first step is to identify such variables.

We also notably use the Scalar Evolution analysis in LLVM which can be used to identify these BIVs by relaying whether a variable belongs to a recurrence relation. If so, we can identify this variable as a BIV.

4 Results

Our experiments were run on a Tesla T4 GPU on Google Colab running Ubuntu 22.04 with Cuda Toolkit version 11.8. We use LLVM version 18.0 installed from source with the OpenMP runtime enabled. The GPU had 15GB RAM and the CPU had 12GB.

We evaluate our method across three settings: CPU only, GPU offloading with manual data transfer, and GPU offloading with unified memory. In each setting, we assess the performance with and without prefetching. Our code is available at https://github.com/aakashdp6548/eecs583-final-project.

4.1 Benchmarks

We use two benchmarks to evaluate our method. The first is an implementation of the standard saxpy BLAS routine, which computes the sum of a scalar times one vector and another vector. The C++ code for this loop is

```
for (long i = 0; i < N; ++i) {
    y[i] = a * x[i] + y[i];
}
```

There are two potential prefetch instructions in this benchmark, the loads to `x[i]` and `y[i]`.

The second benchmark is nested loop that computes $x^\top Ax$ for a vector `x` and a matrix `A`. The C++ code for this loop is

```
double total = 0;
for (long i = 0; i < N; ++i) {
    double sum = 0;
    for (long j = 0; j < N; ++j) {
        sum += A[i][j] * x[j];
    }
    total += sum * x[i];
}
```

There are three potential prefetches in this code: the two loads to `A[i][j]` and `x[j]` in the inner loop, and the load to `x[i]` in the outer loop.

4.2 Data Size

We run the SAXPY benchmark on the CPU varying the number of elements in the input arrays N between 10^6 and 10^9 and investigating the outcomes on execution time between CPU+Prefetch and standalone CPU. The raw latencies are presented in Table 1. Figure 1 presents the time for each size normalized as a percent of the time without prefetch for that size. When the data size is small (near 10^6), the latency for both is so low that we see no appreciable benefit from prefetching. However, as the data size grows, we do see improvements from prefetching compared to no prefetching.

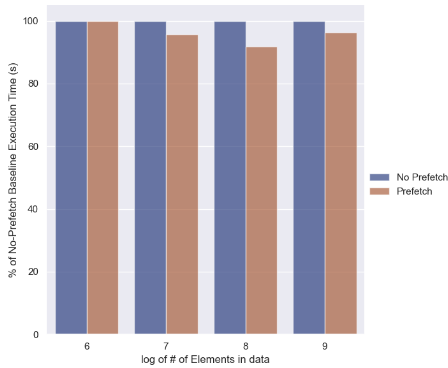

Figure 1: % of Execution Time compared to No-Prefetch Baseline vs. log_{10} # of Elements in Data

4.3 Prefetch Distance

We again run the SAXPY benchmark on the CPU varying our prefetch distance from 1 to 32 iterations to determine the optimal number of iterations to look ahead in our prefetching. The results appear in Figure 2. We note that over our trials, the optimal value of this parameter was 1 (i.e. $log_2(0)$), so we continued the rest of our experiments using this value. We also note the drop-off suggesting an efficiency gain between 2^4 and 2^5 elements, investigation of which we leave to future work due to the possibility of multiple causes.

4.4 GPU Offloading and Unified Memory

As our main experiment, we setup our GPU and benchmarks to support GPU offloading and prefetching using Google Colab and compare benchmark performance across our three modes: CPU, GPU+Offloading, and Unified Memory.

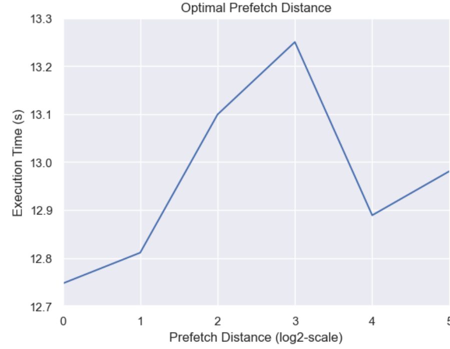

Figure 2: Execution Time vs. log_2 # of Prefetching Lookahead Iterations

We find that in the SAXPY benchmark, enabling prefetching leads to a speedup in all cases. Interestingly, we also find that the GPU+Offloading performed the fastest of our methods, followed by the Unified Memory, and finally the CPU. The results of this experiment are summarized in Table 2 as well as Figure 3. The slowdown of unified memory compared to offloading only suggests that the additional overhead of unified memory is dominating the performance gain; however, this is independent of our method.

In the Nested Loop benchmark, enabling prefetching actually leads to a slowdown in all cases, suggesting that our compiler-based prefetching method may have been too aggressive in data selection. We identify this as a symptom for the need for more intelligent compiler-runtime frameworks and therefore a limitation of the current prefetching approaches, since it seems as though the overhead of prefetching in all three cases is more expensive than the benefits it brings. The results of this experiment are also in Table 2 and Figure 4.

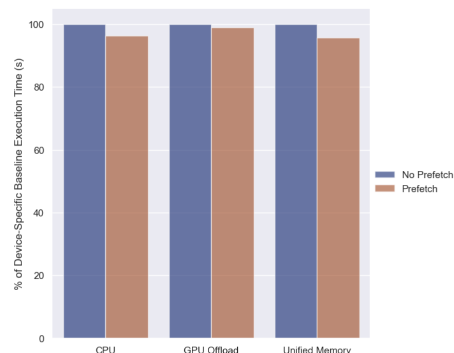

Figure 3: % of Device-Specific Baseline Execution Time vs. Device

log N	Data Size (GB)	No Prefetch	Prefetch
6	0.007451	0.019	0.019
7	0.074568	0.139	0.133
8	0.745058	1.406	1.290
9	7.450581	13.237	12.747

Table 1: Latencies (in seconds) for saxpy benchmark as a function of the number of elements N in the data. This experiment was performed on a CPU, without using OpenMP.

	saxpy		Nested Loop	
	No Prefetch	Prefetch	No Prefetch	Prefetch
CPU	13.237	12.747	9.178	12.706
GPU Offload	12.185	12.074	7.378	7.774
Unified Memory	12.820	12.281	6.456	8.905

Table 2: Latencies (in seconds) for saxpy and nested loop benchmarks on CPU only, GPU offloading without unified memory, and GPU offloading with unified memory. The saxpy program used 7.45GB data and the nested loop program used 6.70GB.

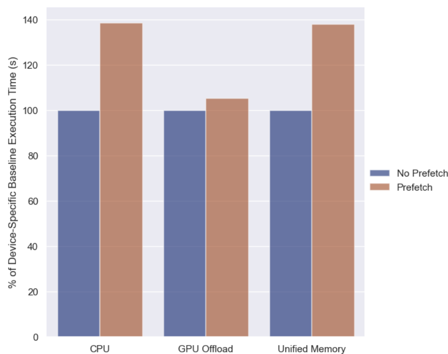

Figure 4: % of Device-Specific Baseline Execution Time vs. Device

5 Conclusion

Here, we demonstrate the performance gains from a static-level analysis for GPU prefetching. We leveraged analyses already available within LLVM in addition to identifying all load instruction operands within general loop structures who participate in recurrence relations as identified by *ScalarEvolution*. Following the identification of these induction variables, we compute the appropriate prefetch distance based on an experimentally-tuned parameter and, after verifying that the address has not already been identified, construct a prefetch pointer for the prefetched load and insert this instruction into the intermediate representation.

Our method demonstrates performance improvements on particular benchmarks, namely the SAXPY benchmark. Though our analysis only leverages compile-time statistics, we aim to generalize our performance improvements to all benchmarks by also performing a profiling analysis as well for application on larger benchmarking suites (e.g. Rodinia). Moreover, further experimentation with including additional LLVM passes upstream, such as induction variable strength reduction, could also lead to demonstrable performance improvements across the board. Analyses such as ours can continue to abstract away the complicated memory transfer procedure in unified memory architecture while also reducing the execution time for any applications containing scientific computations.

References

[1] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 747–764.

[2] Lingda Li. Manage openmp gpu data environment under unified address space.

[3] Tim Mattson and Larry Meadows. 2014. A “hands-on” introduction to openmp. *Intel Corporation*.