Data Reuse Analysis for GPU Offloading using OpenMP

Alok Mishra*
alok.mishra@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA
Lawrence Livermore National Laboratory
Livermore, CA, USA

Chunhua Liao†
liao6@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA, USA

Barbara Chapman‡
barbara.chapman@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA
Brookhaven National Laboratory
Upton, NY, USA

ABSTRACT

More researchers and developers desire to port their applications to GPU-based clusters, due to their abundant parallelism and energy efficiency. Unfortunately porting or writing an application for accelerators, such as GPUs, requires extensive knowledge of the underlying architectures, the application/algorithm and the interfacing programming model (e.g. OpenMP). Often applications spend a significant portion of their execution time on data transfer. Exploiting data reuse opportunities in an application can reduce its overall execution time. In this research we present an approach to automatically recognize data reuse opportunities in an application which uses OpenMP for exploiting GPU parallelism, and consequently insert pertinent code to take advantage of data reuse on GPU. Using our approach we were able to retain reused data on the GPU and reduce the overall execution time of multiple benchmark application.

ACM Reference Format:
Alok Mishra, Chunhua Liao, and Barbara Chapman. 2019. Data Reuse Analysis for GPU Offloading using OpenMP. In *Proceedings of Supercomputing '19: ACM Student Research Competition (Supercomputing '19).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 BACKGROUND & MOTIVATION

In the HPC community, the massively parallel architecture of GPU accelerators are preferred to expedite the computational workloads in cutting edge scientific research. More researchers and developers desire to port their applications to a GPU-based clusters, due to their abundant parallelism and energy efficiency. Unfortunately porting or writing applications for GPU requires extensive knowledge of the underlying architecture, the application/algorithm and the interfacing programming model. One approach to reduce this effort of the developers is to use a directive based programming model, like OpenMP, which ever since version 4.0 have target offloading support as well. Support for offloading to GPU devices is under active development[1] in numerous compilers, like GCC, Intel, Clang/LLVM, Cray, IBM XL, etc.

OpenMP is an attractive approach due to its productivity benefits. Yet OpenMP codes may also spend a significant portion of their execution time on data transfer. Often multiple GPU kernel [1] calls may be reusing the same data, and data transfer to/from CPU and GPU is not required in between these kernel calls. Utilizing data reuse opportunities in an application can reduce its overall execution time.

*Graduate Student author
†Mentor
‡Advisor

Supercomputing '19, November 17–22, 2019, Denver, CO
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 APPROACH

In this research we present an approach to automatically recognize data reuse opportunities in an application, which uses OpenMP for exploiting GPU parallelism, and consequently insert pertinent `data map` directive to take advantage of data reuse on GPU.

For that we perform the following tasks using Clang:

(A) *Kernel Identification.* The first step is to parse the AST to identify the kernels. Our target applications already use OpenMP, so all `omp target` regions in the code are kernels.
(B) *Data Analysis.* We classify data used inside kernels into 4 groups:
  1. *to*: data assigned before and accessed inside the kernel.
  2. *from*: data updated inside and accessed after the kernel.
  3. *tofrom*: data assigned before, updated inside and accessed after the kernel.
  4. *private*: data defined and used only inside the kernel.
(C) *Loop Check.* If kernel is called from inside a loop, then there is a high probability of data reuse in multiple calls to that kernel. Retaining data on the target might be helpful in this case.
(D) *Proximity Check.* We define two kernels to be in close proximity to each other if both the kernels are called from the same function.
(E) *Common Data Check.* We use pattern matching on all data used inside the above identified kernels to check potential data reuse. Next we check if the matched data is modified on host in between kernel calls, i.e. after the first and before the second kernel is called. If data is modified on the host, it is not reused and hence data transfer to/from the CPU is required.
(F) *Data Reuse Table.* Once common data are identified, we update the Data Reuse Table with information like what is the location of the kernels, and what data is being reused, etc.
(G) *Updating Source Code.* At the end of parsing AST, we update the original source code to insert pertinent `data map` directive to take advantage of data reuse on GPU.

[1]In this paper all kernels refer to GPU kernels

Benchmark Application		Number of calls to				Data Transfer (MB)	Time V100 (sec)	Time P100 (sec)	Time K80 (sec)
		Alloc	Free	HtoD	DtoH				
Laplace	Data Reuse	5002	5002	5002	5003	61	119.70	208.97	1392.55
	No Data Reuse	25000	25000	25002	25001	1220703	245.69	437.25	1721.57
LUD	Data Reuse	2	2	3	4	1525	158.70	525.75	4828.50
	No Data Reuse	7496	7496	7497	7499	5718994	3112.37	1987.10	7262.62
MM	Data Reuse	5	5	4	3	763	9.81	36.02	311.84
	No Data Reuse	6	6	8	7	2289	10.12	36.35	312.34
BFS	Data Reuse	7	7	8	3	4387	36.90	73.35	318.61
	No Data Reuse	12	12	13	14	8392	39.28	75.03	323.31
NW	Data Reuse	4	4	5	6	10300	7.54	12.86	28.77
	No Data Reuse	8	8	9	10	27468	8.88	18.77	43.44

Table 1: Results comparing data reuse with no data reuse for benchmark applications

3 EXPERIMENTS AND RESULTS

We used our approach on common benchmark applications like solving Laplace equation using Jacobi iteration and Multiplying 3 Matrices, and on some applications from the Rodinia Benchmark Suite [2], like LU Decomposition, Needleman-Wunsch optimization and Breadth First Search.

```
#pragma omp target data map(to:A,B,C) map(from:D) map(alloc:C1)
{ // data region starts
#pragma omp target teams distribute parallel for collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++)
                sum = sum + A[i * N + k] * B[k * N + j];
            C1[i * N + j] = sum;
        }
    }
#pragma omp target teams distribute parallel for collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++)
                sum = sum + C1[i * N + k] * C[k * N + j];
            D[i * N + j] = sum;
        }
    }
} // data region ends
```

Code 1: Code for multiplying 3 matrices reusing data. Automatically generated code shown in red.

For each application we ran two scenarios – with and without data reuse. Code 1 gives an example of multiplying 3 matrices reusing data. We collect information like, how much data is transferred and how much is the total run time. We also collect the data of how many times CUDA APIs like cuMemAlloc, cuMemFree, cuMemcpyHtoD and cuMemcpyDtoH were called. We ran the experiments on 3 different GPUs - NVIDIA Volta V100, Pascal P100 and Kepler K80.

From Table 1 it is evident that in case of Laplace and LU Decomposition, the performance of the application reduces significantly without data reuse. Even for Needleman-Wunsch optimization we see some reasonable improvement in the performance when data reuse is applied. Although the performance improvement is not much in other cases, there is no loss of performance.

4 CONCLUSION AND FUTURE WORK

Data reuse analysis is a topic quite relevant to the good use of accelerators in the HPC industry. In this research we demonstrate a clear benefit of data reuse. Even in cases where performance improvement is not much, we could conveniently deduce that there is no loss of performance.

In large codes explicitly managing data through manual intervention could be a burden for application scientists. The main advantage of this work lies in the automation of the entire process. This analysis can be used individually by developers to identify data reuse among multiple GPU kernels in their code. It can also help other research aimed at automatic GPU offloading of code [3, 4]. Research on identifying kernels and automatically offloading the computation to GPU can benefit from this approach to make their output more profitable. In future we will be looking into managing data to avoid potential memory over-subscription on the GPU and how to split data transfer and computation into multiple GPUs in case a single GPU runs out of memory due to data reuse.

ACKNOWLEDGEMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was also performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by LLNL-LDRD 18-ERD-006. IM Release Number: LLNL-ABS-784874.

REFERENCES

[1] 2019. OpenMP Compilers & Tools. (April 2019). Retrieved Jul 22, 2019 from https://www.openmp.org/resources/openmp-compilers-tools

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[3] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 13.

[4] Alok Mishra, Martin Kong, and Barbara Chapman. 2019. Kernel fusion/decomposition for automatic GPU-offloading. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 283–284.