Data Reuse Analysis for GPU Offloading using OpenMP

Alok Mishra[1,2], Chunhua Liao[2], Barbara Chapman[1]

alok.mishra@stonybrook.edu, liao6@llnl.gov, barbara.chapman@stonybrook.edu

[1]Stony Brook University – USA, [2]Lawrence Livermore National Laboratory – USA

Abstract

Writing applications for GPUs requires broad knowledge of the underlying architecture, the algorithm and the interfacing programming model. Directive based programming models, such as OpenMP, are an attractive approach due to their productivity benefits. Yet OpenMP codes may also spend a significant portion of the offloading time on data transfer. Exploiting data reuse opportunities in these application can reduce its overall execution time. In this research, we develop an approach to automatically recognize data reuse opportunities in an application which uses OpenMP for parallelism, then insert pertinent code to automatically offload kernels to GPU reusing the same data.

GPU

- Ideal for heavy computational workload, like Matrix Multiplication, Linear Algebra, etc.

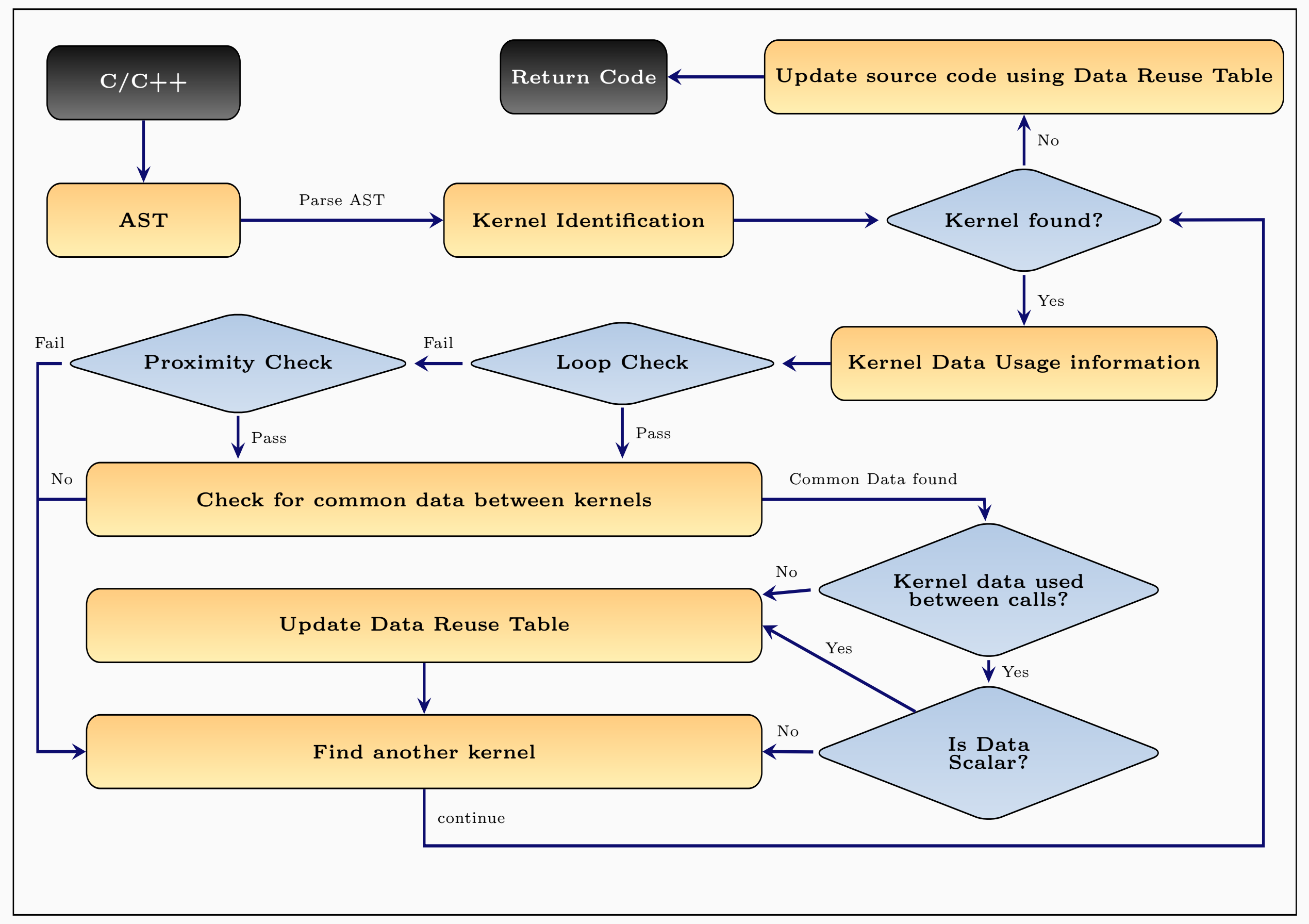
Challenges

- Portability
  - Highly dependent on underlying architecture and choice of programming model
- Data Handling
  - Requires explicit data transfers
  - Applications spend a significant portion of their offloading time on data transfer

OpenMP

- The de-facto portable programming interface for node-level programming
- Comparatively easier to code
- Ever since version 4.X, supports offloading to GPU using nvptx back-end
- Making the code parallel and handling data is still the responsibility of scientists.

Proposal

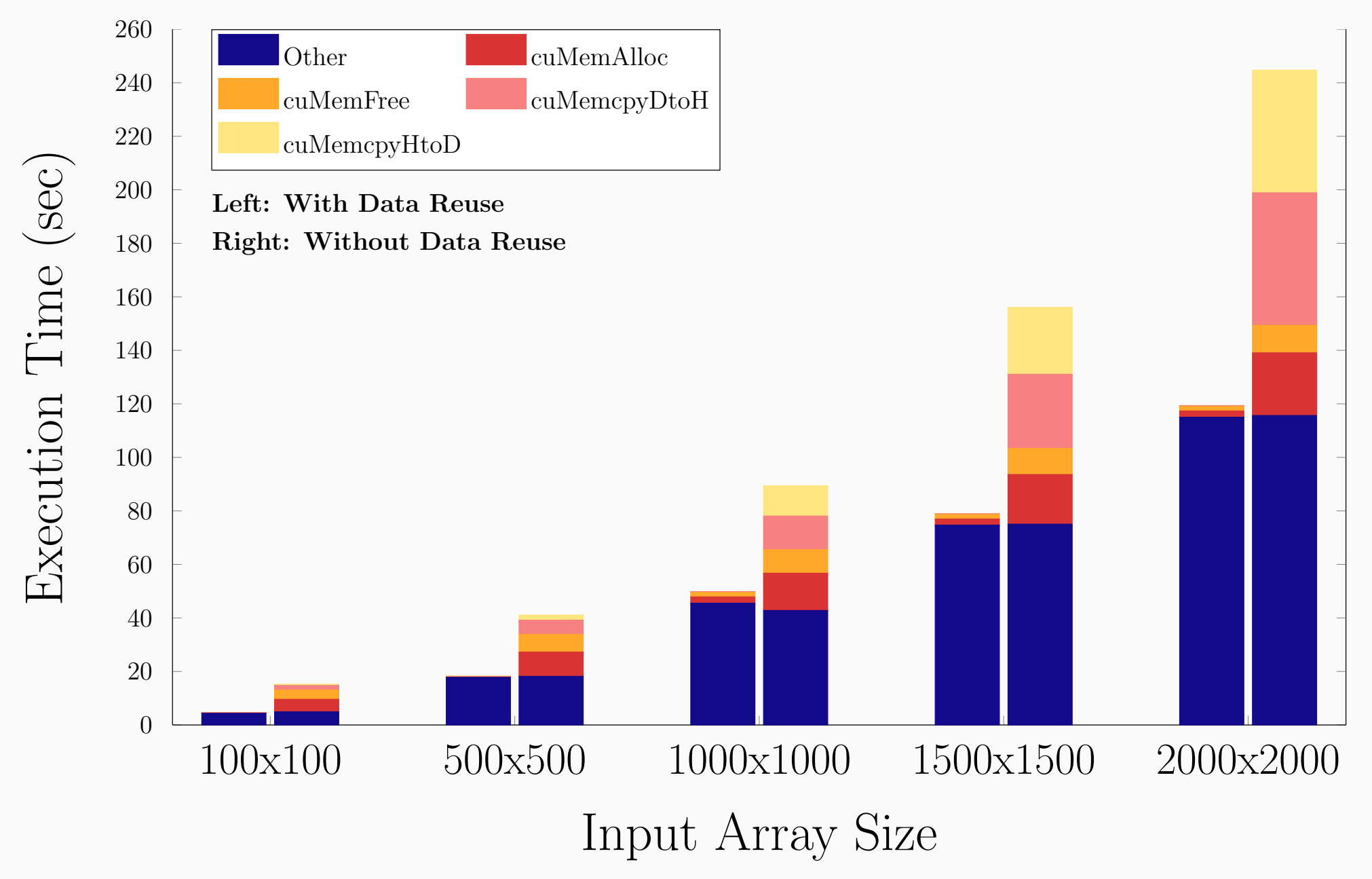


Design & build a compiler framework that can automatically recognize data reuse opportunities in an application and modify the code accordingly.

- Kernel Data:
  - *to:* Data assigned before & accessed in kernel
  - *from:* Data declared before & updated in kernel
  - *tofrom:* Data declared before, updated in & used after kernel
- Loop Check: Check if the kernel is called from a loop
- Proximity Check: Check if multiple kernels are called from same function
- Check Common Data: Compare kernel data between all close kernels to find common data
- Data Reuse Table: Stores information about all data reuse
- Code Generation: insert pertinent code to automatically offload to GPU reusing same data between multiple kernel calls.

We build our framework upon the Clang/LLVM compiler tool.

Initial Experiments & Results



The above graph shows the results of solving the Laplace equation in two dimensions with finite differences using jacobi iteration, for varying matrix size.

cuMemAlloc: Allocate Memory on Device
cuMemFree: Free Memory on Device
cuMemcpyHtoD: Transfer from Host to Device
cuMemcpyDtoH: Transfer from Device to Host

These experiments were run on Lassen[1].

		Number of calls to				Data Transfer	Time V100	Time P100	Time K80
Benchmark Application		Alloc	Free	HtoD	DtoH	(MB)	(sec)	(sec)	(sec)
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

Tab. 1: Comparison of Results on different benchmark algorithm

The above benchmarks were run on the SeaWulf computational cluser[2] on different GPUs - NVIDIA V100, P100 and K80. In case of Laplace and LUD, the kernels are called from within a loop and if data is not reused on the GPU the performance of the application reduces significantly. Even though the performance improvement is not much in other cases, there is no loss of performance.

[1] **Lassen** – Supercomputing cluster at Lawrence Livermore National Laboratory. [https://computing.llnl.gov/computers/lassen].
[2] **SeaWulf Cluster** – Computational cluster at Stony Brook University. [https://it.stonybrook.edu/help/kb/understanding-seawulf].
[3] **Rodinia Benchmark Suite** – Benchmark applications and kernels to run on multi-core CPUs (OpenMP), GPUs (CUDA) and OpenCL. [http://rodinia.cs.virginia.edu/doku.php]

Example Code

```
#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++)
      for (int k = 0; k < N; k++)
        C1[i * N + j] += A[i * N + k] * B[k * N + j];

#pragma omp target teams distribute parallel for \
                        collapse(2)
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        D[i * N + j] += C1[i * N + k] * C[k * N + j];
} // data region ends
```

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

Benchmark Application

- **Laplace** – the Laplace equation in two dimensions with finite differences using jacobi iteration. The experiment used a matrix of size 2000x2000.
- **LUD** – LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The experiment used implementation from Rodinia Benchmark Suite[3] with a matrix of size 20000x20000.
- **MM** – Matrix Multiplication of 3 large matrices of size 5000x5000 each.
- **BFS** – An implementations of breadth-first search algorithm.The experiment used implementation from Rodinia Benchmark Suite[3] with 200 million graph nodes.
- **NW** – Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments. The experiment used implementation from Rodinia Benchmark Suite[3] with a matrix of size 30000x30000.

Conclusion

- Can help individual developers to identify data reuse among multiple GPU kernels in their code
- Can help other research aimed at automatic code generation for GPU offloading of code

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 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-POST-784875)