Using Machine Learning for OpenMP GPU Offloading in LLVM

Alok Mishra[1], Abid M. Malik (Mentor)[2], Barbara Chapman (Advisor)[1,2]

alok.mishra@stonybrook.edu, amalik@bnl.gov, barbara.chapman@stonybrook.edu

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

Abstract

This work explores a novel approach for the automated handling of OpenMP GPU offloading using machine learning techniques. We aim to develop an end-to-end application framework, from legacy code to GPU offloading, that integrates machine learning techniques into the LLVM compiler.

GPU

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

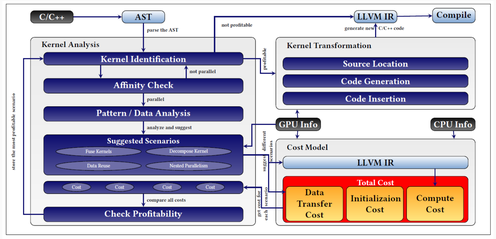
Challenges

- Portability
  - Highly dependent on underlying architecture and choice of programming model
- Programmability
  - Different from existing programming languages. Extensive refactoring of code is required
- Parallelism
  - What is the degree of parallelism?
- 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 parallelism
- 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.

Motivation



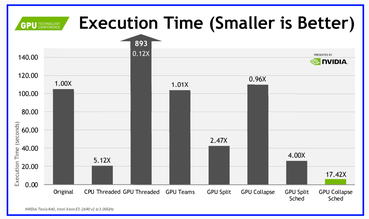
Program Transformation for Automatic GPU offloading using OpenMP.

- Parallel regions detection - Identify Loops
- Patterns Analysis - Suggest Kernel Variants
- Cost model - determine profitability of kernels
- Code Generation - OpenMP directives

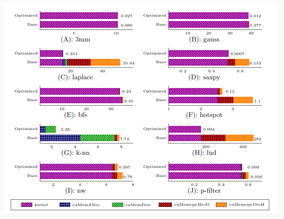
State of the Art Cost Models

- **Hong & Kim, 2009** – uses memory and compute wrap parallelism to estimate effective cost
- **Chen & Aamodt, 2009** – accurately predict the number of extra cache misses
- **Baghshorki et al, 2010** – analyze GPU kernels and identify how each kernel exercises major GPU microarchitecture features
- **Zhang & Owens, 2011** – measures the execution time spent on the instruction pipeline, shared memory, and global memory access to find the bottlenecks and quantitatively analyzes performance
- **Sim et al, 2012** – suggests optimization based upon inter-thread instruction-level parallelism, memory-level parallelism, computing efficiency, and serialization effects
- **Peng et al, 2016** – estimates the cost of different memory access operations
- **Barua et al, 2018** – estimates the memory throughput of a given application
- **Schrödter et al, 2018** – estimates for the overall execution time including compute and transfer times
- **Zhang et al, 2020** – propose a novel active learning based method to exploit the information of evaluated samples.

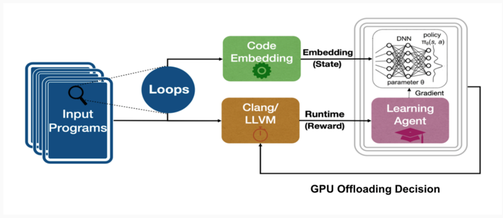
Initial Experiments & Workflow



Different execution time for different optimizations when applied to a simple jacobi iteration



The difference in total execution time on NVIDIA V100 GPU when data transfer is optimized for different benchmark applications.



The workflow for Machine Learning for GPU offloading problem based on Haj-Ali et.al.'s work which uses deep reinforcement learning for automatically tuning compiler.

Common Parameters

Parameters	Description
Thread_per_wraps	Number of thread per wrap
Issue_cycles	Number of cycles to execute one instruction
Frequency	Clock frequency of the SM processor
Mem_bandwidth	Bandwidth between the DRAM and GPU cores
Mem_LD	DRAM access latency

Tab. 1: Common GPU Parameters

Parameters	Description
Thread_per_block	Number of thread per block
Blocks	Total number of blocks in a program
Active_SM	Number of Active SMs
Active_Blocks_per_SM	Number of concurrently running blocks per SM
Active_Wraps_per_SM	Number of concurrently running wraps per SM

Tab. 2: Common Machine Resource Parameters

Parameters	Description
Total_Inst	#Comp_Inst + #Mem_Inst
Comp_Inst	Total number of computation instructions in one thread
Mem_Inst	Total number of memory instructions in one thread
Uncoal_Mem_Inst	Total number of uncoalesced memory instructions in one thread
Coal_Mem_Inst	Total number of coalesced memory instructions in one thread
Sync_Inst	Total number of synchronized instructions in one thread
Coal_per_MW	Number of memory transaction per wrap (coalesced access)
Uncoal_per_MW	Number of memory transaction per wrap (uncoalesced access)
Load_byte_per_wrap	Number of bytes per wrap

Tab. 3: Common Source Code Analysis Parameters

Cost Functions

$$Total\_Cost = \Theta_0 \times C_{compute} + \Theta_1 \times C_{data\_transfer} + \Theta_2 \times C_{init}$$

$$C_{compute} = \Theta_{C0} \times Common\_GPU\_Parameters + \Theta_{C1} \times Machine\_Resource\_Parameters + \Theta_{C2} \times Source\_Code\_Analysis\_Parameters$$

$$C_{data\_transfer} = \Theta_{d0} \times D + \Theta_{d1} \times N + \Theta_{d2} \times S_L + \Theta_{d3} \times N_L + \Theta_{d4} \times S_q$$

Data Transfer Time can be calculated by training models using gradient descent technique. Here
D is Total Data transferred,
N is the number of variables transferred,
S_L is speed of the PCIe link,
N_L is the Number of lanes in the PCIE link and
S_q is the speed of QPI link between CPUs.

C_{init} = Initialization Time has impact only once for an application, when the first target region is encountered. For small programs, it can have a significant impact. It can be calculated experimentally.

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.