Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling

Larisa Stoltzfus
University of Edinburgh
Edinburgh, UK
larisa.stoltzfus@ed.ac.uk

Murali Emani
Lawrence Livermore National Laboratory
Livermore, CA
emani1@llnl.gov

Pei-Hung Lin
Lawrence Livermore National Laboratory
Livermore, CA
lin32@llnl.gov

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

ABSTRACT

Modern supercomputers often use Graphic Processing Units (or GPUs) to meet the ever-growing demands for high performance computing. GPUs typically have a complex memory architecture with various types of memories and caches, such as global memory, shared memory, constant memory, and texture memory. The placement of data on these memories has a tremendous impact on the performance of the HPC applications and identifying the optimal placement location is non-trivial.

In this paper, we propose a machine learning-based approach to build a classifier to determine the best class of GPU memory that will minimize GPU kernel execution time. This approach utilizes a set of performance counters obtained from profiling runs along with hardware features to generate the trained model. We evaluate our approach on several generations of NVIDIA GPUs, including Kepler, Maxwell, Pascal, and Volta on a set of benchmarks. The results show that the trained model achieves prediction accuracy over 90% and given a global version, the classifier can accurately determine which data placement variant would yield the best performance.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Machine learning**;

KEYWORDS

GPU, Data placement, Memory, Machine Learning

ACM Reference Format:
Larisa Stoltzfus, Murali Emani, Pei-Hung Lin, and Chunhua Liao. 2018. Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3286475.3286482

MCHPC'18, November 11, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286482

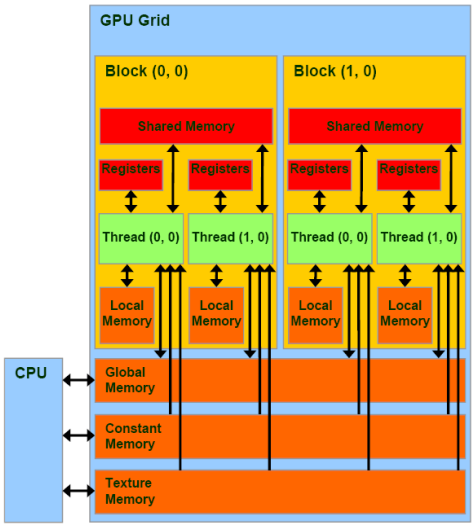


Figure 1: GPU memory hierarchy (from NVIDIA [13])

1 INTRODUCTION

Supercomputers use various types of memory and increasingly complex designs to meet the ever growing need for data by modern massively parallel processors (e.g., Graphic Processing Units or GPUs). On an NVIDIA Kepler GPU, for instance, there are more than eight types of memory (global, texture, shared, constant, various caches, etc.). Figure 1 shows a typical GPU memory hierarchy. With unified CPU-GPU memory space support on the latest GPUs (e.g., NVIDIA Volta), data placement becomes even more flexible, allowing direct accesses to data across the boundaries of multiple GPUs and CPU. This problem will critically affect the effective adoptions of new generations of supercomputers, such as Sierra hosted at LLNL [1] and Summit at ORNL [2] featuring NVIDIA Volta GPUs using 3D stacked memory, unified CPU-GPU memory space, and other memory complexities. As HPC applications get ported to run on the new supercomputers with GPUs, they will have to rely on a range of optimizations including data placement optimization to reach desired performance.

Studies [4, 8] have shown that placing data onto the proper part of a memory system also known as *data placement optimization*,

has significant impact on program performance; it is able to frequently speed up carefully written GPU kernels by over 50% (up to 13.5× on an MPEG video encoding kernel [7]). State-of-art literature explored different techniques to optimize data placement on older generations of GPUs. A portable compiler/runtime system for deciding the optimal GPU data placement is proposed in PORPLE [3, 5]. Internally it uses a lightweight performance model based on cache reuse distance to determine the data placement policy. Huang and Li [6] have proposed a new approach by analyzing correlations among different data placements. It then uses a sample placement policy to predict the performance for other placements on a Kepler GPU. A rule-based system to guide memory placement on Tesla GPUs based on data access patterns has been introduced in Jang et al. [7]. Mei et al. [11] and Jia et al. [9] have proposed microbenchmarking to analyze and expose cache parameters of Fermi, Kepler, Maxwell and Volta respectively.

The novelty in this work is in predicting optimal data placement on newer GPUs. Machine learning-based approach helps to automate the decision making process for optimal data placement without modifying the implementation of the underlying algorithm. Prior state-of-art approaches used offline profiling analysis, but were unable to capture the required features at runtime. By observing the runtime features of just the global/default version of a code, the proposed approach is able to determine which memory placement will yield best performance. In this work, we use a machine learning-based approach where a classifier model is trained once offline. During inference, runtime features are captured and passed as an input feature vector to the model that determines the optimal data placement location.

This paper makes the following novel contributions:

- determining the optimal data placement location on-the-fly during run-time
- providing a simple, lightweight solution that is applicable to diverse applications
- introducing an approach and supplying data which can be reused for other optimizations, such as determining optimal data layouts.

2 MOTIVATION

In this section, we demonstrate how the data placement in memory could impact the program execution time. Here we first provide a brief description of GPU memory hierarchy and then show how different placement of data onto various types of memories impact the kernel execution time.

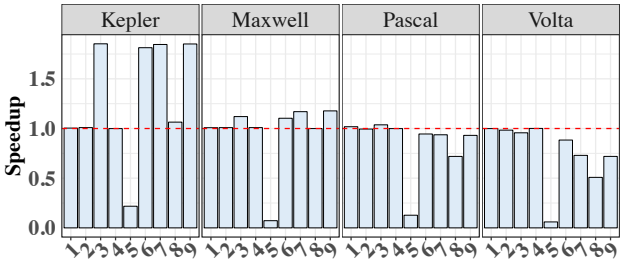
2.1 GPU Memory Structure

GPUs have a highly complex memory hierarchy in order to exploit their massive parallel computing potentials. A high-level overview of the major types of memories listed in Table 1, as exposed to programmers via the CUDA API is listed as follows:

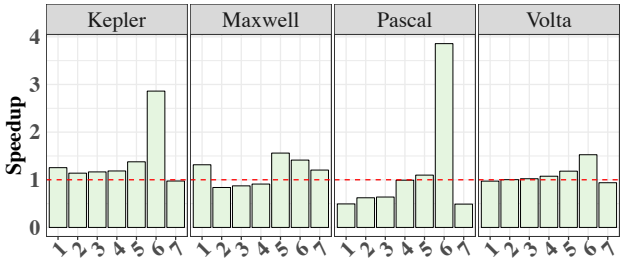
- **Global Memory**: This largest off-chip memory serves as default, main memory on a GPU. Global memory accesses are characterized with limited memory bandwidth and long latencies compared with on-chip memory or cache.
- **Shared Memory**: This on-chip memory is characterized by low-latency and high-bandwidth. It is software-managed and is accessible by active threads that belong to a streaming multiprocessor (SM) unit on a GPU.
- **Constant Memory**: This is a predefined within the global memory space and is set as read-only. The memory space is globally visible to all threads.
- **Texture Memory:** Similar to read-only constant memory, texture memory is off-chip memory space that is optimized for 2D spatial locality. It is suited for threads that access memory addresses that are closer to each other in 2D.

Memory type	Location	Access	Cached	Scope
Global	Off-Chip	Read-Write	Y	Global
Shared	On-Chip	Read-Write	N	SM
Constant	Off-Chip	Read-only	Y	Global
Texture	Off-Chip	Read-only	Y	Global

Table 1: GPU memory types



(a) Speedup of Sparse Matrix Vector Multiplication



(b) Speedup of Matrix Multiplication

Figure 2: Impact of data placement on kernel performance across different GPUs. Speedups over default vary with different memory variants shown on x-axis.

Although all NVIDIA GPUs have a similar high-level design, different generations of GPUs introduce new memory properties or implement the same memories using different physical organization. For example, Fermi GPUs introduced true cache hierarchy for global memory while previous GPUs did not have such caches. On Kepler GPUs, L1 cache and shared memory are combined together while texture cache is backed by its own memory chip. Pascal and Maxwell GPUs have dedicated memory for their shared memory. For the latest Volta GPUs, L1 cache, shared memory and texture cache are all merged into a single unified memory, All these hardware changes have a direct impact on memory optimization.

2.2 Preliminary Experiments

Preliminary experiments show that the performance achievable from different data placements can be specific to applications and platforms. From the graphs in Figure 2, it is evident that the version of the hardware and the application both heavily influence the kernel performance. These graphs show two benchmarks utilizing different GPU memories, numbered across the x-axis and their speedup compared to the global version of the benchmark. Table 2 shows what different numbered versions correspond to for the SPMV benchmark (for brevity, only one table is shown). Figure 2a shows that the performance of SPMV application, which is a sparse matrix vector application, generally worsens with the latest version of NVIDIA GPUs when utilizing non-global memory. However for MM, the matrix-matrix multiplication benchmark, performance improvement can still be seen on the latest platforms (Volta and Pascal) using non-global memory as shown in Figure 2b. Both benchmarks show that for many applications it is clearly important to get the data placement right. Applications optimized for a particular platform will not necessarily retain that performance on future platforms and indeed future platforms with advanced hardware may not require optimizing at all.

Version	Description
1	rows array in shared
2	rows array in constant
3	vector array in texture1D, rows in shared
4	matrix values in texture1D
5	vector array in constant, rows in texture1D
6	vector array in texture
7	matrix values and columns in texture1D, rows in constant
8	matrix values, columns and vector in texture1D
9	matrix values, columns, rows and vector in texture1D

Table 2: Memory configuration details of SPMV benchmark

3 APPROACH

The machine learning model for directly predicting the appropriate placement plan is based on data access patterns and memory system specifications. The workflow involves two phases, namely *offline training* and *online inference*, as shown in Figure 3. The offline training phase involves the construction of a classifier using supervised learning. It takes the feature vector of the kernel and data as inputs, and predicts the best data placement in memory. As with any machine learning approach, we investigate different classifiers such as Random Forests, Support Vector Machines, AttributedClassifiers and pick one that yields the highest prediction accuracy and confidence. To train the classifiers, we obtain the training data by evaluating representative kernels with various data placements and labeling the samples with observed best execution times.

3.1 Offline Training

3.1.1 Design of Training Experiments. The training experiments are set up to run scripts over a selection of benchmarks with varying types of data placements and data and thread block sizes across the different GPU platforms listed in Section 4. First, the best performing versions are found by measuring the program execution

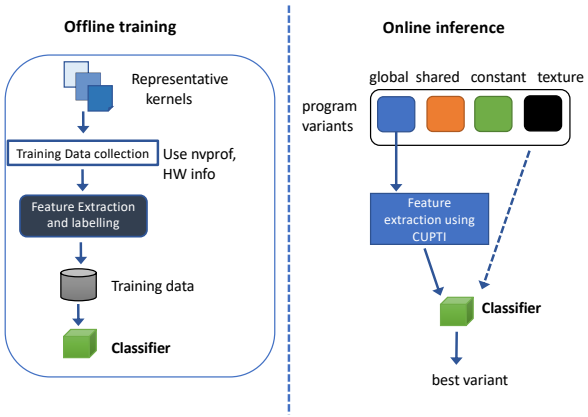


Figure 3: Workflow of the proposed machine learning-based data placement engine.

times of all possible memory variants, for a variety of data sizes and thread block sizes of each benchmark. Then the global memory versions are profiled for their feature values, selecting only those metrics and events available on all platforms. These features are paired with class labels of the best versions available for a given benchmark of a data size and thread block size. Feature values are first normalized and then re-sampled to ensure a larger spread for use. The benchmarks along with the hardware platforms can be found in Section 4.

3.1.2 Feature Engineering: As with any machine learning-based models, the classifier to make accurate predictions for the data placement, relies heavily on the input features. Here, we present how the features are extracted and important ones are selected.

(a) Feature Extraction: We use the NVIDIA profiling tool `nvprof` to extract the hardware counters and performance metrics that compose the features and use the `CUPTI` API [12] to extract these features from the codes at runtime. Hardware parameters obtained from extensive microbenchmarking [10] are then appended to each sample of features obtained `nvprof`. The raw training data set is obtained by running 8 benchmark programs with various combinations of 4 memory variants, utilizing 3 variations of data sizes and 3 different thread block sizes on 4 GPU architectures for at least ten iterations. We have 4,876 samples with 241 features per sample collected from `nvprof` and hardware. This number is larger than expected due to there being more than one version for certain benchmarks as well as additional iterations that may have been run. The benchmarks used range from 1-7 input arrays which can potentially be placed in an alternative memory.

(b) Dimensionality Reduction: The dimensionality of the features is reduced from 241 to a minimal set as shown in Table 3. These significant ones are obtained using a correlation-based feature selection (CFS) algorithm. Such reduction in the required features helps to extract only those features that are important to the model.

3.1.3 Model Training: With the reduced feature set, we evaluated different classifiers to compare respective prediction accuracies, obtained using 10-fold cross validation. Such a strategy ensures that the model does not overfit the training data and the reported

Feature	Description
achieved_occupancy	ratio of average active warps to maximum number of warps
active_warps	average of active warps per cycle
l2_subp1_write_sysmem_sector_queries	number of system memory write requests to slice 1 of L2 cache
l2_subp0_total_write_sector_queries	total number of memory write requests to slice 0 of L2 cache
l2_subp0_total_read_sector_queries	total number of memory read requests to slice 0 of L2 cache
l2_subp0_read_sector_misses	total number of memory read misses from slice 0 of L2 cache
fb_subp1_write_sectors	number of DRAM write requests to sub partition 1
dram_read_throughput	device memory read throughput
eligible_warps_per_cycle	average number of warps eligible to issue per active cycle
l2_read_transactions	memory read transactions at L2 cache for all read requests
l2_write_transactions	memory write transactions at L2 cache for all write requests
gld_throughput	global memory load throughput
flop_count_sp	single-precision FLOPS executed by non-predicated threads
flop_count_sp_special	single-precision special FLOPS executed by non-predicated threads
warp_nonpred_execution_efficiency	ratio of average active threads executing non-predicated instructions to the maximum number of threads
warp_execution_efficiency	ratio of average active threads to the maximum number of threads

Table 3: List of selected features to train the classifiers

Classifier	Prediction accuracy
RandomForest	95.7%
LogitBoost	95.5%
IterativeClassifierOptimizer	95.5%
SimpleLogistic	95.4%
JRip	95.0%

Table 4: Classifiers and their prediction accuracies

prediction accuracies are based on unseen test data. A selection of classifiers with high prediction accuracies is listed in Table 4.

3.2 Online Inference

The classifier built in the offline training phase is then used for predicting the best data placement for new applications at runtime through the `CUPTI` profiler API. Honing in on the fewest number of features needed, the application is profiled in real-time, the features are then fed as input to the classifier. Based on the prediction, the best version of the application is then executed. The assumption here is that different memory variants of the code already exist and the appropriate version that is determined by the classifier is executed at runtime.

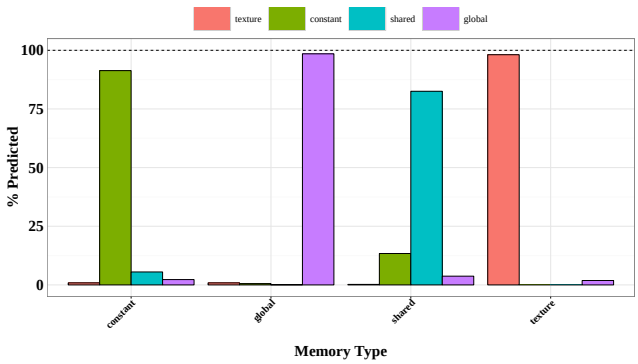


Figure 4: Graph showing the model-predicted memory classes with the best performing memory variants. Across all GPUs the model correctly predicts the best performing variant in at least 80% and up to 95% cases.

4 EVALUATION

Here we list the hardware and software platforms used to evaluate the proposed approach, followed by the experimental results that show how close the model predicted memory variant is with the best possible one. Table 5 shows the four machines used for our experiments. They contain four generations of GPUs namely Kepler, Maxwell, Pascal and Volta. The programs run include *Sparse Matrix Vector Multiplication* (SPMV), *Molecular Dynamics Simulation* (MD), *Computational Fluid Dynamics Simulation* (CFD), *Matrix-Matrix Multiplication* (MM), *ParticleFilter*, *ConvolutionSeparable*, *Stencil27*, and *Matrix Transpose*.

In the first run, the best performing version of each benchmark is determined. The benchmarks were run using a script to collect average values over ten iterations and the median of these values is taken. Each kernel was run for five times to warm up the GPU before timings were taken. In order to collect consistent performance times, the performance benchmarks used `cudaEventRecord()` to denote kernel start and stop places and `cudaEventElapsedTime()` to calculate the time elapsed. `cudaDeviceSynchronize()` was used after these calls and data transfer times are excluded in order to isolate performance differences from memory usage.

5 ANALYSIS

In this section, we demonstrate how well the `JRIP` model performs and the accuracy in its predictions for each of the benchmarks across the different platforms. We have selected this model because it is the most portable of those available and all of the top performing models hover around ~95% prediction accuracies. This model has a relative absolute error of ~9.5%. The prediction accuracies obtained from other classifiers are listed in Table 4. The results of our machine learning model experiments show that we are able to get very good results from tree-based models. Even if the model may not accurately predict the best performing data placement, it may still predict a better data placement than the global version.

The graph in Figure 4 shows the results for the best data placement class comparing the model predicted memory variants with

	Kepler (K40)	**Maxwell (M60)**	**Pascal (P100)**	**Volta (V100)**
CPU	IBM Power8 @2.2GHz	Intel Xeon E5-2670 @2.60 GHZ	IBM Power8 @2.2GHz	Intel Xeon E5-2699 @2.20GHz
Computation capability	3.5	5.2	6.0	7.0
SMs	15	16	56	80
Cores/SM	192 SP cores/64 DP cores	128 cores	64 cores	64 SP cores/32 DP cores
Texture Units/SM	16	8	4	4
Register File Size/SM	256 KB	256 KB	256 KB	256 KB
L1 Cache/SM	Combined 64K L1+Shared	Combined 24KB	Combined 24 KB	128 KB Unified
Texture Cache	48KB			
Shared Memory/SM	Combined 64K L1+Shared	96 KB	64 KB	
L2 Cache	1536 KB	2048 KB	4096 KB	6144KB
Constant Memory	64 KB	64 KB	64 KB	64 KB
Global Memory	12 GB	8 GB	16 GB	16 GB

Table 5: Key specifications of selected GPUs of different generations

the best performing ones. We found that the memory class predicted by the machine learning model is accurate in at least 80% and up to 95% of cases. Specially, given a global memory variant, this model is able to correctly predict which of the memory variants would yield the best performance (lowest execution time). These results are averaged across different GPUs.

It can be observed from the figure that, except for few instances, the offline trained model is able to correctly classify the best memory variant by observing the default global variant. The model correctly classifies global and texture memory variants to the actual best performing variants. The percentage of correct predictions is however lower for shared memory variants.

6 CONCLUSION

We have presented an automated approach to data placement optimization using machine learning to tune applications on GPUs. We built a classifier to determine the memory variant that could yield the best performance. The evaluations demonstrate that the model predictions are nearly as accurate as the best achievable cases.

This work has shown that there is an immense possibility for machine learning to be applied to automate data placement optimizations. As part of the future work, this technique would be integrated into an automated workflow and have runtime code generation of the appropriate memory variant based on the model decision. Ideally, a compiler would handle these low-level details. Additionally, the overhead of using `CUPTI` profiling tools interface means that performance suffers. Essentially, the profiler must run the kernel of interest for each metric or event, meaning that extra iterations are added. This could be alleviated by running cut down data sizes. Finally, this work could easily be applied to other areas such as different data layout optimizations.

ACKNOWLEDGMENTS

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 was supported by the LLNL-LDRD Program under Project No. 18-ERD-006. (LLNL-CONF-758021)

REFERENCES

[1] 2018. Sierra- next generation HPC system at LLNL. https://computation.llnl.gov/computers/sierra. (2018).
[2] 2018. Summit at Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/summit/. (2018).
[3] Guoyang Chen, Xipeng Shen, Bo Wu, and Dong Li. 2017. Optimizing data placement on GPU memory: A portable approach. *IEEE Trans. Comput.* 66, 3 (2017), 473–487.
[4] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU. In *The 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 88–100.
[5] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 88–100.
[6] Yingchao Huang and Dong Li. 2017. Performance modeling for optimal data placement on GPU with heterogeneous memory systems. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 166–177.
[7] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel & Distributed Systems* 1 (2010), 105–118.
[8] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *The 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 1–14.
[9] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).
[10] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826
[11] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86.
[12] NVIDIA. 2018. CUDA Profiling Tools Interface. (2018). https://developer.nvidia.com/cuda-profiling-tools-interface
[13] CUDA NVIDIA. 2007. NVIDIA CUDA programming guide (version 1.0). *NVIDIA: Santa Clara, CA* (2007).