Learning Indirect Memory Access Patterns of GPU Applications

Nafis Mustakin*
University of California Riverside
nmust004@ucr.edu

Sakshar Chakravarty*
University of California Riverside
schak026@ucr.edu

Abstract

Given memory load information collected by profiling simulations from GPGPU-sim[2], we propose to use & compare both RNN and LSTM along with clustering techniques to predict memory access made by GPU threads. Learning these patterns will expose dependency among adjacent kernels thereby enabling independent kernels to run without blocking.

Keywords: GPU, memory access pattern, neural networks, clustering, RNN, LSTM

ACM Reference Format:
Nafis Mustakin and Sakshar Chakravarty. 2021. Learning Indirect Memory Access Patterns of GPU Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The current trend of excelling computation power flows in two directions. One is focused towards increasing single-thread performance of commercial super-scalar microprocessors and the other is through multi-threaded hardware that can provide parallelism explicitly whenever possible. Parallel computation is not a new concept. However, modern graphics processing unit (GPU) has taken over the more traditional CPUs in this aspect. The usage of GPU has transgressed into more general purpose high performance computation from its original graphical computation. There are lots of works going on to make GPU programming suitable for non-graphics applications. CUDA programming model introduced by NVIDIA has made it so easy for programmers to utilize GPUs for general purpose high performance computation requiring both process and data parallelism. GPUs are now widely being used in data center applications and machine learning. The requirement for more capable architectures to cope with the high computational demand of these is ever increasing. The implementation of complex and highly computation based programs using raw GPU architecture already produces better performance in throughput and higher power conservation. Still further analysis shows that there are scopes for improvement, like apart from designing GPU architecture with more scalability at hardware level, looking into the behavior of applications may open a few doors.

Another caveat of GPU architecture is that there is no fine-grain support for data-dependent parallelism and synchronization. This causes significant amount of load imbalance and resource under utilization in the hardware. It has been found that processing unit may remain idle even when there are thread blocks whose dependencies are met already. Based on this observation, Wireframe[1], a data-dependency detection model was proposed, which gives a coarse-grained data dependency by forming dependency graphs between thread blocks during compile time. However, to get dependency information at fine-grained level, it is required to predict the memory access patterns by an individual thread.

One way to find kernels that are data-independent would be to analyze GPU application memory access patterns. In most GPU applications, the memory access pattern is highly predictable as most of the access is base address offset by the thread ID. However, GPUs can do indirect memory loads for example of the form $B[A[0]$. Indirect access can be both regular which are more predictable and irregular which seem rather random. The regular types are like the example mentioned above which are address translation from another table. The irregular types are often associated with graph applications and are harder to predict. Practical models of address prediction fail to capture the behaviour of irregular indirect accesses. But Hashemi et. al.[7] have shown that RNN combined with clustering based prediction models can model these kind of accesses for CPU workload, thereby making the case of this approach being extensible for GPU workloads as well.

In this work, we apply a machine learning model that first clusters the thread blocks based on their memory accesses differentiating between strided or indirect using K-Nearest Neighbors (KNN). For the thread blocks with strided memory access patterns are ignored as per the scope of this project

*Both authors contributed equally to this research.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

as predicting them is merely a stride prediction. But for the thread blocks with indirect memory accesses, we train both an RNN and an LSTM on the indirect access portion of the memory access sequence. Also, we offer some performance comparisons between these two models with proper justification. Using the predictions from these models, we can predict the memory access locations or rather a range of locations for a particular thread block and therefore a kernel. Using this knowledge we can identify kernels which exhibit data independence between them and thereby can be parallelized.

2 Related Work

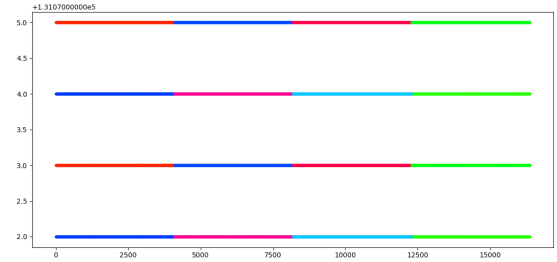
Learning memory access pattern for the purpose of prefetching is an age old problem. Hardware prefetchers have existed for a long a time which can be broadly classified into two separate categories - Stride prefetchers ([6] [11] [13]) and Correlation prefetchers ([4] [12]). But these require large costly tables in hardware, and prefetching is not required anyway in GPUs.

Crago et. al.[5] have used the stride and regular indirect access patterns to introduce new instructions that can exploit these for better efficiency in GPU address calculation. Besides, there have been lots of works based on machine learning in microarchitecture and analyzing program behavior using memory traces. Moreover, the use of deep learning is rather new in this area. One of the first works on microarchitectural problem was the perceptron branch predictor [10] that used a linear classifier for predicting whether a branch is taken or not. The model was quite simple and had a very low latency. Also, reinforcement learning has been applied for optimizing long-term performance of memory controller scheduling algorithms [9], tuning performance knobs [3], and identifying patterns in hardware and software that can be tied to a memory access [14]. Recently an LSTM-based prefetcher [15] was proposed that tried to capture regular patterns. However, irregularity in memory access patterns made such regression-based model difficult to perform better. Lastly, an RNN based model combined with clustering [7] has been able to outperform practical models for memory accesses in generalized computing. The memory access patterns in GPUs are not much different from CPUs, even more straighforward in some cases. This can be viewed as a great opportunity for learning the memory access patterns in GPUs using a quite similar approach.

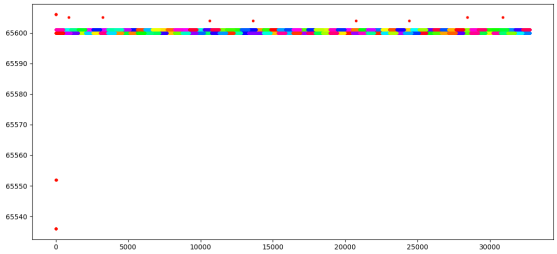
3 Proposed Method

3.1 Data Visualization

We started by trying to visualize the memory addresses accessed by the program traces available to us. We separate the memory accesses by thread blocks as shown in Figure 1. As we can see, the vector addition program accesses a contiguous block of memory. But the BFS trace has a number of outlying access points along with the accesses to a contiguous block. We can see that the outlying points are being accessed by the same thread block. Our observation varies significantly from the CPU memory access paper as i) most accesses are to a contiguous memory block, and ii) there are no discernible clusters. But the outlying points in Figure 1b are actually more useful for our purpose than the contiguous blocks. Predicting memory addresses within the contiguous block is fairly straightforward as it boils down to simple stride prediction based on the thread block and thread ID. But the more complicated patterns arise from the indirect memory accesses, which are the outliers in our findings.



(a) Vector Addition



(b) BFS

Figure 1. Addresses accessed per Thread Blocks

We have to isolate these outliers to focus our model training on these datapoints (and we can choose to ignore the contiguous block for now). For that we run an unsupervised outlier detection using local outlier factor. We visualize the separation in Figure 2.

As working with the entire address space increases our input vocabulary, we calculate the deltas of the accesses made by thread block 0. Deltas are the difference between the addresses of the current memory access and the previous access. We visualize the deltas as shown in Figure 4a. As we can see from Figure 4a thread block 0 also has a large range of purely stride accesses which correspond to a delta of 1. The indirect accesses occur on the tail end which is where we want to focus our efforts on. We can get a closer look at the indirect access deltas in Figure 4b

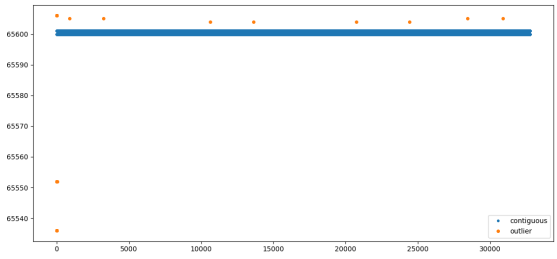
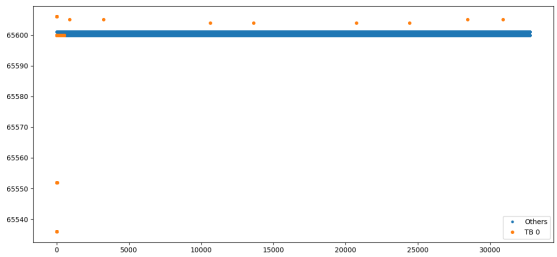
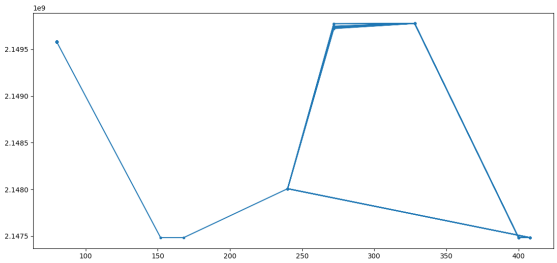


Figure 2. Outlier addresses

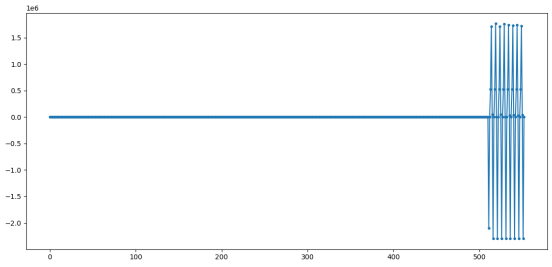


(a) TB 0 addresses

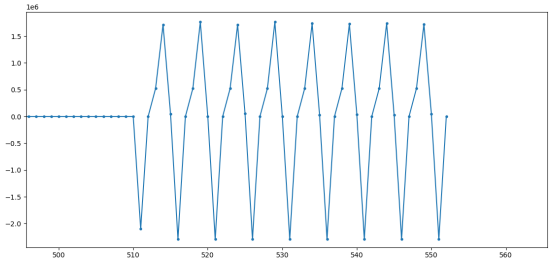


(b) TB 0 PC vs Address

Figure 3. Addresses accessed by Thread Block 0



(a) TB 0 deltas



(b) TB 0 indirect access deltas

Figure 4. Deltas of Thread Block 0

model then predicts the output deltas which can be added with the previous address to get the address predictions.

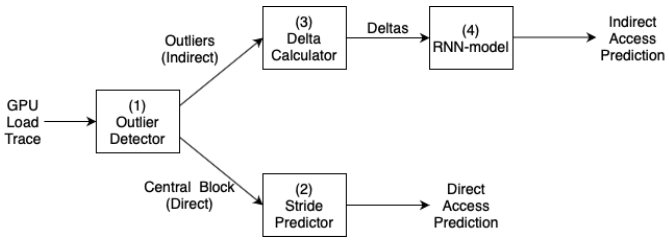


Figure 5. Model Architecture

3.2 Model Architecture

Our model architecture consists of two distinct steps being-

1. Indirect Access Detection: where we isolate the thread block(s) making the indirect accesses
2. Indirect Access Prediction: where we train the model with the deltas from the thread block(s) making the indirect accesses.

The model architecture can be seen in Figure 5. The preprocessed data is first put through an outlier detector (1) which then categorizes the data into the central block which are direct accesses and the outliers which are the irregular accesses. The outliers are then fed into the delta calculator (3) as a preprocessing step of training the RNN model in (4) which can be either a simpleRNN or an LSTM[8]. The RNN

3.2.1 Indirect Access Detection. As mentioned in Section 3.1 the BFS load data has easily identifiable indirect accesses. Hashemi et. al. [7] uses k-means algorithm to cluster memory accesses into separate localized regions. However, the data here has one distinct central cluster which constitutes most of the memory accesses and a small fraction of outliers which are the indirect accesses. Therefore the indirect access detection becomes an outlier detection problem. We perform an unsupervised outlier detection with a local outlier factor. We find that to accurately separate the outliers, we need to consider $k > 25$ neighbours.

Using this k-nearest neighbour (KNN) approach, we can determine which thread block(s) are making the indirect accesses. Once we have isolated such thread blocks, we use the data specific to that block to train our model as specified

in the next section. A key point to note here is that our data has very few indirect access points, 42 to be exact, which indicates that the model required to accurately capture this access pattern might not be too complex.

3.2.2 Indirect Access Prediction.

LSTM:. Hashemi et. al. [7] uses LSTM with a limited vocabulary to predict the output deltas. Following their approach, we initially train an LSTM on the deltas calculated for thread block 0. Our initial setup of small number of LSTM blocks and short lookback factor performs rather poorly. Upon tuning the hyperparameters, we find that a longer lookback factor usually performs better. But given that we have very few datapoints, increasing the lookback factor leads to a significant portion of the datapoints remaining unused for testing. So in general, if we have a similar or even slightly worse performing model for a shorter lookback factor, we will prefer it over the longer lookback model since this will leave us with more usable datapoints. Our best model using LSTM was found for 220 LSTM nodes, using a lookback factor of 3. We visualize the performance of the model in Figure 6.

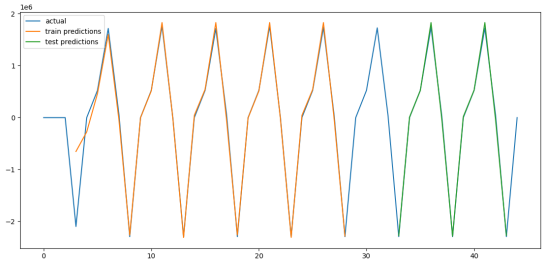


Figure 6. Performance of 220 block, 3 lookback LSTM

Simple RNN:. As stated before, we have very few datapoints, and the pattern of indirect accesses are somewhat repetitive. Since LSTM is a very complex model which usually requires a large dataset to work effectively, we trained a different model using simple RNN as well. Simple RNN performs significantly worse for a shorter lookbacks, but if we increase the lookback factor we find that the performance is on par with LSTM and even better for some combinations of hyperparameters. According to our tests, a simple RNN with 128 blocks and 12 lookback factor has the lowest RMSE error but we prefer the model with 512 blocks and 8 lookback factor since it has slightly worse RMSE but has shorter lookback. The performance of this model is visualized in Figure 7.

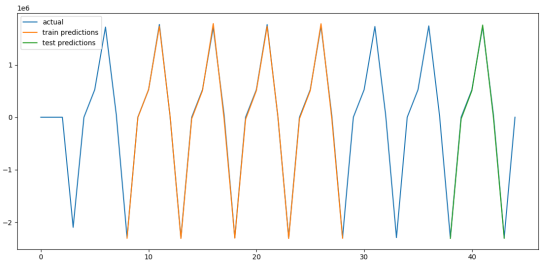


Figure 7. Performance of 512 block, 8 lookback Simple RNN

4 Experimental Evaluation

4.1 Metrics

Since this is a novel approach to detecting kernel dependency on GPUs, we do not have any existing methods to evaluate the performance of our model against. Hashemi et. al. [7] uses precision-at-10 and recall-at-10 as their model makes 10 predictions at a time and they treat it as a *success* if any of the 10 predictions are correct and a *failure* if none of them are. Since our model is not setup the same way to predict a number of deltas at a time and take the correct prediction, we cannot use precision and recall for evaluating our model. Instead, we use the Root Mean Squared Error (RMSE) to compare the performance of our models.

4.2 Model Comparison

We evaluate both the LSTM and the simple RNN model for a number of different lookback factors and block numbers. For the simple RNN model we can see from Tables 1 and 2 that increasing the length of the lookbacks usually leads to better RMSE values. But as mentioned before, we prefer a model with shorter lookback with comparable RMSE, which is why we choose 512 block & 8 lookback model to be the best among the simple RNN based models.

Similar to the case of simple RNN we tune the hyperparameters of the LSTM model and record the RMSE for a number of different configurations. But we tuned the LSTM model more finely which is why we can see some of ad hoc values in tables 3 and 4. We can see that our observation of longer lookback leads to better RMSE holds up here as well. Here we prefer the solution with 3 lookback on 220 blocks. Note that the solution with 256 blocks and 8 lookback is comparable to if we adjust the RMSE for the additional unused data points. But since the earlier has a lower lookback, we prefer it over the latter.

5 Conclusions and Future Work

Even though we had a very small dataset, our model managed to perform well. The high values of RMSE is not something that deters us since our purpose is to find data dependency between kernels that might occur as a result of these indirect

Lookback \ Blocks	128	256	512
4	436748.12	493503.58	548773.77
8	49054.5	50828.75	**34289.67**
12	30842.73	64119.81	52505.42

Table 1. RNN RMSE on Train data

Lookback \ Blocks	128	256	512
4	301290.21	423956.36	406741.53
8	41620.78	43696.74	**28968.93**
12	22818.1	67755.01	49233.47

Table 2. RNN RMSE on Test data

Lookback \ Blocks	128	200	220	256
3	350968.9	346503.4	**299282**	289597
8	37256.3	59766.6	41899.5	32688.9
10	56655.3	42484	38053.4	50439.3

Table 3. LSTM RMSE on Train data

Lookback \ Blocks	128	200	220	256
3	158036.5	200604.3	**61028.9**	69049.8
8	23048.8	54652.5	29783.4	21352.7
10	37845.6	38706.6	20984.4	34171.6

Table 4. LSTM RMSE on Test data

accesses. So as long as we are accurate within a range, we can predict with some level of confidence whether two kernels might share data dependency or not.

We are in the process of generating more traces with preferably a large number of indirect accesses to evaluate our model on a different dataset. In the event that our current model do not perform well given the new data, we propose to use the context information of program counter (PC) value along with the delta to train our models, similar to the approach of Hashemi et. al. [7] as shown in Figure 3b, PC may be useful to capture context better and allow us to lower the lookback factor even more.

References

[1] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. 2017. Wireframe: Supporting Data-Dependent Parallelism through Dependency Graph Execution in GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 600–611. https://doi.org/10.1145/3123939.3123976

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[3] Ronald D. Blanton, Xin Li, Ken Mai, Diana Marculescu, Radu Marculescu, Jeyanandh Paramesh, Jeff Schneider, and Donald E. Thomas. 2015. Statistical Learning in Chip (SLIC). In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (Austin, TX, USA) *(ICCAD '15)*. IEEE Press, 664–669.

[4] Mark J Charney and Anthony P Reeves. 1995. *Generalized correlation-based hardware prefetching*. Technical Report. Technical Report EE-CEG-95-1, Cornell University.

[5] Neal C. Crago, Mark Stephenson, and Stephen W. Keckler. 2018. Exposing Memory Access Patterns to Improve Instruction and Memory Efficiency in GPUs. *ACM Trans. Archit. Code Optim.* 15, 4, Article 45 (Oct. 2018), 23 pages. https://doi.org/10.1145/3280851

[6] J Gindele. 1977. Buffer block prefetching method. *IBM Technical Disclosure Bulletin* 20, 2 (1977), 696–697.

[7] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 1919–1928. http://proceedings.mlr.press/v80/hashemi18a.html

[8] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[9] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *2008 International Symposium on Computer Architecture*. 39–50. https://doi.org/10.1109/ISCA.2008.21

[10] D.A. Jimenez and C. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 197–206. https://doi.org/10.1109/HPCA.2001.903263

[11] Norman P. Jouppi. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 364–373. https://doi.org/10.1145/325096.325162

[12] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-Block Prediction & Dead-Block Correlating Prefetchers. *SIGARCH Comput. Archit. News* 29, 2 (May 2001), 144–154. https://doi.org/10.1145/384285.379259

[13] Subbarao Palacharla and Richard E Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*. 24–33.

[14] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic Locality and Context-Based Prefetching Using Reinforcement Learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 285–297. https://doi.org/10.1145/2749469.2749473

[15] Yuan Zeng and Xiaochen Guo. 2017. Long Short Term Memory Based Hardware Prefetcher: A Case Study. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia) *(MEMSYS '17)*. Association for Computing Machinery, New York, NY, USA, 305–311. https://doi.org/10.1145/3132402.3132405