Data-centric Combinatorial Optimization of Parallel Code

Hao Luo¹ Guoyang Chen² Pengcheng Li¹ Chen Ding¹ Xipeng Shen²

¹ University of Rochester ² North Carolina State University
{hluo,pli,cding}@cs.rochester.edu {gchen11,xshen5}@ncsu.edu

Abstract

Memory performance is one essential factor for tapping into the full potential of the massive parallelism of GPU. It has motivated some recent efforts in GPU cache modeling. This paper presents a new data-centric way to model the performance of a system with heterogeneous memory resources. The new model is composable, meaning it can predict the performance difference due to placing data differently by profiling the execution just once.

Categories and Subject Descriptors C.4 [Performance of Systems]: Modeling Techniques

Keywords Locality metrics, Locality modeling, Footprint

1. Introduction

Memory performance is essential to parallel performance. This paper is concerned with highly coupled parallelism on the current GPU architecture. While the degree of parallelism on a GPU chip is not overly large compared to traditional massively parallel processing (MPP), the memory system is very different. The speed of access is far greater when the datum is on-chip. When it is not, the aggregate bandwidth is far lower, since all data reside on one module of main memory.

Indeed, memory is one of the performance bottlenecks on GPUs and as a result, it has the most complex design. It includes shared memory and various types of cache including L1/L2, texture and constant caches. The memory complexity continues to increase as GPU architects try to maximize the throughput of highly threaded executions.

A programmer now has the burden of dealing with many parameters. She chooses what data to place in cache and how to configure the cache (e.g. capacity of L1). Each parameter may have a large impact on performance. Exhaustive testing, either through direct execution or simulation, would take too long. Even after it is done, the result offers little insight about a program or its machine.

In this paper, we take the approach of modeling, of both a parallel program and the GPU hardware. Modeling is fast. It profiles a program execution just once and then calculates the performance when we examine all parameters. It picks the best values after examining all possible values of parameters. This method is based on low-overhead modeling with one-time profiling rather than costly, repetitive testing.

To build such model, we formulate and solve a fundamental problem on multi-core and many-core processors — the heterogeneity problem. Heterogeneity exists in software and hardware:

- *Data nonuniformity*. Nonuniform locality can be found between different data. For example, in scientific computing, arrays in programs may be accessed in irregular patterns, resulting in worse locality than those regularly accessed arrays.
- *Memory heterogeneity*. On-chip memory has different types. For example, L1 data cache and texture cache on GPUs. When compared with each other, L1 cache has larger capacity and texture cache has longer access latency.

Different heterogeneities interact. Nonuniform data compete for the cache space and the individual occupancy varies depending on the interaction with its peers. The current GPU programming model provides programmers control over this interaction through selectively placing data in different memory. It is this capability that brings great potential and challenge for optimization.

We present a data-centric, compositional analysis to model the locality of nonuniform data. The analysis is grounded on a single theory based on the footprint of data access. The footprint gives the average size of the data used in a time period. In this paper, we make an extension to model the interaction between data nonuniformity and memory heterogeneity.

2. Data-centric Models of Parallel Code

2.1 Background: Locality Defined by Footprint

An *execution trace* is a sequence of memory accesses and each access is represented by a memory address. A subsequence of consecutive accesses in the trace is called an *execution window*.

The locality of an execution window is the working-set size (WSS), which is the amount of distinct data accessed in the window [2]. WSS varies depending on the window. The *footprint*, $fp(w)$, is defined as the average WSS for all windows of length w [3–5].

For example, consider the execution trace `abcca`. When the window length $w = 1$, each element is a window. WSS is 1 for each window, and the average WSS $fp(1) = 5/5 = 1$. There are 4 windows of length $w = 2$, their working-set sizes are 2, 2, 1, and 2, and the average, $fp(2) = 7/4$. Similarly, we have $fp(3) = 7/3$, $fp(4) = 6/2 = 3$, and $fp(5) = 3/1 = 3$. For any sequence of memory accesses, the footprint $fp(w)$ is a unique function defined for $w > 0$.

Based on footprint, Xiang et al. formulated a higher-order theory of locality (HOTL) [5]. The theory gives a relation between the footprint function and cache miss ratio. The HOTL theory stipulates that *the average increase of the working-set size, i.e. the miss ratio, is the increase of the average working-set size, i.e. the derivative of the footprint*:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

PPoPP '16 March 12–16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03...\$15.00
DOI: <http://dx.doi.org/10.1145/2851141.2851182>

$$mr(c) = mr(fp(w)) = fp(w+1) - fp(w)$$

where $c = fp(w)$ is the size of a fully associative LRU cache.

2.2 Parallel Footprint to Model Data Nonuniformity

2.2.1 Time-preserving Decomposition

In footprint analysis, an execution trace is a sequence of memory accesses, each a memory address. In this paper, we first propose a way to *decompose* an execution trace into *components*. A component is a subsequence of a trace. For our purpose, we consider only hierarchical decomposition, where the components are either disjoint or nested. Minimal components are all disjoint, and any other component is a union of multiple minimal components.

We say a decomposition is *time preserving (TP)* if every access in each component preserves the time information it had in the complete sequence. For example, the trace `abc abc abc...` accesses three distinct data elements, and it can be decomposed into three per-element components, `a__ a__ a__...`, `_b_ _b_ _b_...` and `__c __c __c...`, where “_” represents a placeholder indicating that in the original trace there was an access to another component.

We define the *parallel footprint* $pfp(w)$ for a TP sequence. The working-set size (WSS) is still the amount of distinct data accessed in a time window, but the placeholders do not contribute to the working set. The parallel footprint $pfp(w)$ is the average WSS for all window length w .

As an example, consider the component `a__ a__ a__...`. For unit length windows, the working-set size (WSS) is 1 in every three windows and 0 in others, so we have $pfp_a(1) = 1/3$. In a regular footprint, we always have $fp(1) = 1$, for all traces. For a parallel footprint, $pfp(1)$ can be any value between 0 and 1 (including 0 and 1).

2.2.2 Data-centric Analysis

The parallel footprint supports data-centric analysis. We aim to maximize performance by choosing a cache memory for each array in the program to be placed on. To achieve that, we decompose a program’s execution trace into components such that each component consists of the addresses from an array. We measure the components’ parallel footprints and then compute the aggregate footprint of multiple arrays by simple addition.

Table 1 shows an example of data-centric analysis. The full sequence consists of accesses to three data items. The top three rows show the time-preserving decomposition on the left and the parallel footprint on the right, for each datum.

Table 1. Footprint composition: the footprint of a trace is computed by adding the parallel footprints of its component traces.

	traces	$pfp(w)$		
		$w=1$	$w=2$	$w \geq 3$
single-datum	<code>a__a__a__...</code>	1/3	2/3	1
locality	<code>_b_ _b_ _b_...</code>	1/3	2/3	1
(measured)	<code>__c __c __c...</code>	1/3	2/3	1
data group locality (composed)	<code>ab_ab_ab...</code>	2/3	4/3	2
	<code>_bc_bc_bc...</code>	2/3	4/3	2
	<code>_ca_ca_ca...</code>	2/3	4/3	2
	<code>abcabcabc...</code>	1	2	3

The benefit of parallel footprints is shown in the bottom four rows. For any subsequence that accesses two or more data items, we can compute the aggregate footprint directly, without having to analyze the sequence again. If we were to measure the footprint for the composed sequences, shown in the lower left of the table, we would obtain the same result as we do from calculation. Suppose a

program accesses a set of arrays, the parallel footprint of any subset of the arrays, S , is the aggregate parallel footprint of the members of S . That is,

$$pfp_S(w) = \sum_{i \in S} pfp_i(w)$$

2.2.3 Properties of Parallel Footprints

The main property of TP decomposition is as follows: *If the components do not access common data, the sum of their parallel footprints is the same as the parallel footprint of the combined sequence.* The example in Table 1 shows this property for three components whose accesses are uniformly interleaved. The property actually holds for *arbitrarily interleaved components*:

- The components may contain any number of accesses. For example, one component may contain 100 times more accesses than another component.
- The ordering of the components may be arbitrary. For example, most accesses in one component may predate all accesses in another component.

TP decomposition can be used for accesses timed by physical clock (e.g. CPU cycles or instruction count). At each tick of a physical clock, there may be no access, one access or multiple accesses. When there are multiple accesses at the same time, a component is no longer a sequence since not all accesses are totally ordered. But the data-centric, compositional analysis in Section 2.2.2 can be applied without changes.

2.3 Data-centric Optimization

Based on the notion of TP decomposition, we have developed a technique for analyzing per-array locality for CUDA programs. We integrated the analysis into a data placement engine PORPLE [1]. It enhanced PORPLE’s cache performance model and brought 10% extra speedup to a set of 5 GPU programs on average.

3. Conclusion

In this paper, we presented a data-centric locality analysis. The analysis models how data with nonuniform locality interact and compete for cache space. It is a useful extension to footprint theory in optimizing the data placement for GPU programs.

Acknowledgments

We are grateful to Jacob Brock and the anonymous reviewers, for their valuable feedback and comments. The research is supported by the National Science Foundation (Contract No. CNS-1319617, CCF-1116104, CCF-0963759), IBM CAS Faculty Fellowship, the National Natural Science Foundation of China (Contract No. 61328201) and a grant from Huawei Technologies.

References

- [1] G. Chen, B. Wu, D. Li, and X. Shen. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of MICRO*, 2014.
- [2] P. J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5):323–333, 1968.
- [3] C. Ding and T. Chilimbi. All-window profiling of concurrent executions. In *Proceedings of PPOPP*, 2008. *Poster paper*.
- [4] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of PPOPP*, pages 91–102, 2011.
- [5] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*, pages 343–356, 2013.