A Highly Parallel Reuse Distance Analysis Algorithm on GPUs

Huimin Cui
SKL Computer Architecture,
Institute of Computing Technology, CAS
Beijing, China
cuihm@ict.ac.cn

Qing Yi
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, USA
qingyi@cs.utsa.edu

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW, Australia
jingling@cse.unsw.edu.au

Lei Wang
SKL Computer Architecture,
Institute of Computing Technology, CAS
Beijing, China
wlei@ict.ac.cn

Yang Yang
SKL Computer Architecture,
Institute of Computing Technology, CAS
Beijing, China
yangyang@ict.ac.cn

Xiaobing Feng
SKL Computer Architecture,
Institute of Computing Technology, CAS
Beijing, China
fxb@ict.ac.cn

***Abstract*—Reuse distance analysis is a runtime approach that has been widely used to accurately model the memory system behavior of applications. However, traditional reuse distance analysis algorithms use tree-based data structures and are hard to parallelize, missing the tremendous computing power of modern architectures such as the emerging GPUs. This paper presents a highly-parallel reuse distance analysis algorithm (HP-RDA) to speedup the process using the SPMD execution model of GPUs. In particular, we propose a hybrid data structure of hash table and local arrays to flatten the traditional tree representation of memory access traces. Further, we use a probabilistic model to correct any loss of precision from a straightforward parallelization of the original sequential algorithm. Our experimental results show that using an NVIDIA GPU, our algorithm achieves a factor of 20 speedup over the traditional sequential algorithm with less than 1% loss in precision.**

I. Introduction

On modern architectures, the performance of applications critically depends on their memory access behavior, e.g., whether they demonstrate a degree of locality and whether their patterns of data reuses can be adequately exploited by the cache hierarchy of modern computers. To accurately model the data reuse patterns of applications, existing research has resorted to the concept of reuse distance (also known as stack reuse distance), which is defined as the number of *distinct* data items accessed between two successive references to the same data [9], [2]. Reuse distance analysis can be used to directly predict various aspects of the memory system performance of an application, e.g. the hit ratio when running on a fully-associative LRU cache [5], the whole-program locality [9], the locality phases [25], and the miss ratios across different program inputs [34]. It can also be used to guide various optimizations, e.g. to generate cache hints [6], to guide loop tiling [31], [30], and to reorder code and data to improve locality [17].

To demonstrate the process of reuse distance analysis, Figure 1 shows a small sequence of memory accesses (also called a memory trace) which presumably can be dynamically generated online while executing some user application. In the example trace, the distance between the two accesses of b is 5, as five distinct elements, c, g, e, f, and a, have been accessed in between. To analyze the whole trace, the reuse distance between each pair of consecutive accesses to the same data must be computed, therefore with a worst-case complexity of $O(N^2)$, where N is the length of the memory trace. The result of reuse distance analysis is typically formulated as a histogram of the percentages of memory accesses with reuse distances falling inside various ranges between 0 and ∞.

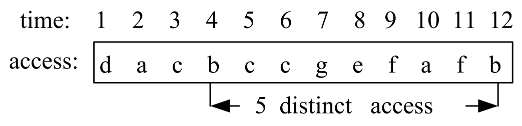

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
5 distinct access

Figure 1: Reuse distance example [9].

Computing a full reuse distance histogram is therefore quite expensive [23]. One option is to perform the analysis offline by collecting the sequence of all memory references made by a user application and then analyzing the whole trace afterwards. However, the space required for storing the whole trace could be overwhelming for long-running applications. As a result, existing research mostly adopted online analysis where fragments of memory traces are collected and forwarded for analysis while running the application. Existing research has exploited a number of tree-based data structures,e.g., m-ary tree, AVL-tree [20], and splay tree [26], to implement the algorithm efficiently. Ding et. all [9] also leveraged approximate reuse distance analysis to shorten the analysis period. However, even among the most efficient implementations, analyzing every memory reference of a program while evaluating the code slows down the program execution by at least 1-2 orders of magnitude [26].

The emerging massively parallel Graphics Processing Units (GPUs) offer an opportunity to accelerate this process. In particular, applications can be evaluated on a conventional multi-core CPU while a separate GPU is dedicated to analyzing the dynamically collected memory traces from running the application. However, the traditional tree-based reuse distance analysis algorithms are fundamentally sequential and hard to parallelize on GPUs, as a global shared tree data structure needs to be modified when analyzing each memory reference in the trace (for more details, see Section II). To parallelize such algorithms, a key challenge is to eliminate artificial dependences introduced by the global shared tree data structure. Further, to enable massive parallelism, the conventional reuse distance analysis algorithm needs to be reformulated so that thousands of sub-tasks can be used to operate on different portions of a memory trace simultaneously.

This paper presents a new parallel reuse distance analysis algorithm, the HP-RDA (Highly Parallel Reuse Distance Analysis) algorithm, to overcome the above challenges while using GPUs to dramatically promote the efficiency of existing sequential algorithms. In particular, we propose a hybrid data structure of hash table and local arrays to flatten the traditional tree representation of memory access traces. Then, we use a probabilistic model to correct any loss of precision from a straightforward parallelization of the original sequential algorithm. Out experimental results show that using an NVIDIA GPU, our algorithm achieves a factor of 20 speedup over the traditional sequential algorithm with less than 1% loss in precision.

The rest of the paper is organized as follows. Section II introduces the traditional sequential reuse analysis algorithm proposed by Ding et al [9]. Section III presents our highly-parallel algorithm, the hybrid data structure of hash table and local arrays, and the probabilistic model to correct the final results. Section IV discusses implementation details on GPUs. Section V presents experimental results. Section VI discusses related work, and Section VII concludes.

II. Background

A reuse distance analysis algorithm takes as input a sequence of memory addresses accessed during program execution, measures the reuse distance between each pair of consecutive accesses to the same address, and then reports the collected data. The sequence of memory addresses is typically called a memory trace, and a balanced tree, such as the data structure shown in Figure 2, is often used to dynamically organize the memory references for efficient lookup of the access history. For example, consider the balanced tree T in Figure 2, which is constructed after processing the first 11 memory accesses in Figure 1. Here each memory address that has been processed corresponds to a unique tree node with three fields: the memory address x, the time step t that x was last accessed, and the size (number of nodes) w of the sub-tree beneath the current node. At time step 12, the memory address b is accessed. The new reuse distance for b can be computed as the number of existing nodes $(x_j, t_j, w_j) \in T$ s.t. $x_j \neq b$ and $t_j > t_b$, where t_b = 4 is the latest access time of b in the existing balanced tree in Figure 2. A hash table can be used to dynamically map each memory address to its latest access time in T.

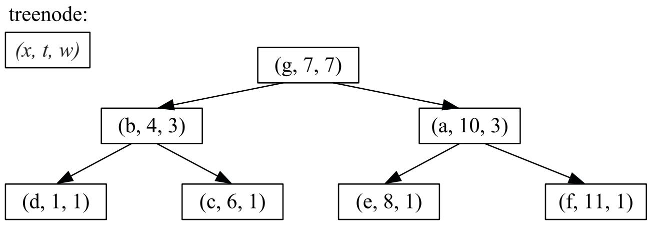


Figure 2: Balanced tree representation of the first 11 memory references shown in Figure 1. Each node corresponds to a distinct memory address x with t being its latest access time and with w number of nodes in the subtree underneath [9]
.

```
Algorithm Reuse_Analysis (ptree, v, t_curr)
1.  //Find v's latest access time.
    t_v = FindHash(hashmap, v);
2. //Compute the distance for v, and delete the previous
    //access to v from ptree.
   dist = Compute_Dist_Delete(ptree, t_v);
3. //Insert new access to v under current time step tcurr,
    //and rotate the tree to keep balance.
    ptree.Insert_Rotate(newnode(v, t_curr));
    return dist;
end algorithm

subroutine Compute_Dist_Delete (ptree, t_v)
    node = ptree.GetRoot();
    //Traverse the tree, finding all nodes whose latest
    //access time is later than t_v.
    while node is not null do
        if  (node.t > t_v) then
            dist = dist + node.right.w
            node = node.left;
        else if (node.t < t_v) then
            node = node.right;
        else
            dist = dist + node.right.w;
            ptree.Delete(node);
            break;
        end if
    end while
    return dist;
end subroutine
```

Figure 3: Sequential reuse distance analysis algorithm [9]

Figure 3 shows the sequential reuse distance analysis algorithm introduced by Ding et al [9]. A similar algorithm was also used by Almasi et al [2]. Here, the routine *Reuse_Analysis* traverses an existing balanced tree $ptree$ in pre-order and computes the reuse distance for each memory reference v in three steps.

1) Find the time step that v was last accessed by looking up a global hash table. Save it to variable t_v.
2) Compute the reuse distance of v as the number of nodes that have later access time than t_v in $ptree$. Erase the previous access to address v in $ptree$ using t_v as a key.
3) Insert a new access to address v into $ptree$ under the current time step. Rotate the tree if necessary to keep it balanced.

Note that since significant modifications are made to $ptree$ when processing each memory access, the algorithm is fundamentally sequential and difficult to parallelize.

III. The HP-RDA Algorithm

The goal of our Highly-Parallel Reuse Distance Analysis algorithm is to reformulate the sequential algorithm so that massive parallelism can be exploited using the SPMD execution model on GPUs. The key insight is the use of a hybrid data structure of hash table and local arrays to flatten the traditional tree representation of memory access traces, and the use of a probabilistic model to correct any loss of precision from a straightforward parallelization of the original sequential algorithm. Section III-F discusses the accuracy of the computed results and the generality of our approach.

A. The Overall Algorithm

The overall HP-RDA algorithm adopts a divide-and-conquer approach. First, to support massive parallelism, we divide the input memory trace into a large number of disjoint smaller traces to be analyzed simultaneously by multiple threads. Then, the results of analyzing the thread-local traces are combined to compute a solution for the original input. Since processing disjoint fragments of the original memory trace in parallel may violate dependence constraints between memory references in the original input trace, a merging step tries to correct such violations by recovering the lost reuse distance information from a set of statistics pre-computed for each thread-local trace. Finally, we use a probabilistic model, presented in Section III-E, to adjust the final solution and reduce error rates.

Figure 4 gives a skeleton of our algorithm, which includes the following four steps.

```
algorithm Parallel_Reuse_Analysis (trace)
1.  //Preprocess the input trace
    trace = Pre_Process(trace);
2.  //Divide trace equally into n small traces.
    local_trace = Divide_Trace(trace, n);
3.  //Analyze the small trace using multi-threads.
    for each thread P(i) do
        local_result[i]  = Thread_Analysis(local_trace[i]);
    end for
4.  //Merge thread local results.
     results = Merge_Results(local_result);
     return result;
end algorithm
```

Figure 4: HP-RDA algorithm.

1) Preprocessing the input: To reduce the required global memory space and to distribute balanced workload among the CUDA threads, this step modifies the input trace so that when a single memory address is repeated by a sequence of neighboring references, all the repetitive references are removed (e.g., a sequence of $aaaa$ would be replaced with a single a). In particular, since the reuse distance of these continuously repeated elements is always zero, the information is easily computed on-the-fly during the trace transformation. Note that this step can also be applied in the sequential algorithm to reduce tree insertion operations.

2) Dividing up the input trace: This step divides an input trace with N memory references equally across n threads, with each thread-level trace containing N/n references. In Figure 4, the local trace to be processed by each thread i is saved in $local\_trace[i]$.

3) Analyzing thread-local traces in parallel: This step is evaluated by a large number of CUDA threads concurrently, with each thread analyzing it's own thread-local trace to compute relevant reuse distances and summarize various statistics of the trace for later processing. Section III-C presents details of this step.

4) Merging thread-local results: This step is evaluated after all the threads have finished evaluating the previous step, so that the thread-local results in $local\_result$ are now ready to be combined into reuse distances for the original input. Note that this step is also parallelized, where reuse distances for different memory addresses are merged concurrently. Because concurrent thread-level analyses may have violated a set of dependence constraints among memory references in the original input, the reuse distances for some memory references may have to be approximated in the merging process. Section III-D presents details of this step and how to further adjust the final results to reduce the loss of precision based on a probabilistic model.

B. Revising Data Structures

The sequential reuse distance analysis algorithm in Section II uses a balanced tree to dynamically store the latest access time of memory addresses processed so far. The tree needs to be modified and rotated when processing each memory reference. Since each tree rotation may operate on a large number of branches, a GPU algorithm operating on such a tree data structure could easily result in all threads being sequentialized waiting to operate on the shared branches. To overcome this difficulty, we designed a hybrid data structure of a hash table and a set of local arrays to support massive parallelism by a large number of CUDA threads.

1) The Hash Table: We use a global hash table to map each memory address (key) in the original input trace to a list summarizing how the address has been accessed within various thread-local traces. In Figure 5(a), *access_info* illustrates the data structure used to summarize the access information of a memory address **v** within a thread-local trace. Here, the tid field remembers the index of the tread-local trace (i.e., the thread id used to analyze the trace), $first$ stores the number of distinct memory references within the trace before encountering the first access of **v**, $latest$ stores the number of distinct memory references after processing the last access of **v** in the trace, and the $next$ pointer is used to organize multiple *access_info* objects (computed from different thread-local traces) into a list.

The sequential algorithm in Section II also uses a hash table to remember the previous access time of each memory address. Our hash table is different in that it contains more information and is used to summarize the statistics computed from a large number of concurrent threads analyzing different portions of the original input trace.

2) Local Arrays: To support concurrent thread-level analysis, we associate with each thread a private array of N entries, where N is the length of the thread-local trace, to keep track of the latest reference of each memory address within the trace. Initially the entire thread-local trace is copied into the private array. While processing each entry $j = 1, ..., N$ of the trace, $\forall i = 1, ..., j-1$, the ith entry within the trace is the latest reference to a memory address *if and only if* $local\_array[i] \neq 0$, where $local\_array[i]$ is ith entry of the private array.

Figure 5(b) illustrates the content of a local array when processing the 13th entry a thread-local trace. In particular, when $index = 13$, the memory address a is encountered. Since $local\_array[2] = a$, the previous latest access of a was encountered when processing the 2nd entry of the trace. Since the local array contains 5 non-zero entries, e, g, c, b, and d, between its 2nd and the 13th entries, the resulting reuse distance for a is 5. After recording the reuse distance, we modify $local\_array[13]$ with value a since entry 13 has now become the latest access to a. Then, we reset $local\_array[2]$ to zero before proceeding to process the next address in the local trace.

C. Thread-Level Analysis

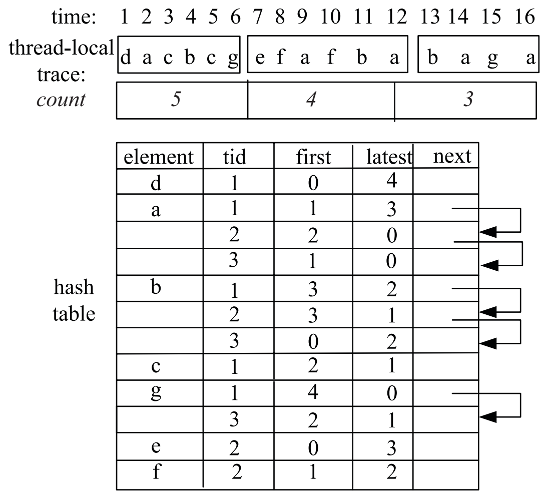
Figure 5(c) summarizes the steps performed by each CUDA thread spawned by our algorithm. The algorithm takes a single input, the local trace to be analyzed by the thread, and compute two sets of information: the reuse distances of the local trace, and the statistics of how each address is accessed within the local trace. The reuse distances are returned as result of the thread-level analysis, and the thread-local statistics are stored inside the global hash table as a set of *access_info* objects (see Figure 5(a)), one for each address referenced inside the local trace. The evaluation includes the following three steps.

1) Step 1: Initialize the local array from the input trace.

2) Step 2: For each index i of the local array and the corresponding memory access v stored in $local\_array[i]$, query the lash table to find out the time step t_v that v was last accessed. Then, set the reuse distance of v to be the number of non-zero elements between t_v and i with $local\_array$. Finally, after saving the reuse distance for v, reset $local\_array[t_v]$ to zero, and enter the additional access information into the hash table,

3) Step 3: Count the number of distinct memory addresses inside the local trace.

Figure 6 illustrates the output of the algorithm after analyzing a trace of 16 memory references. As example, the memory address *b* appears in all three thread-local traces. In the 2nd thread-local trace, three distinct memory addresses, *e, f* and *a*, are accessed before the first access of *b*, and one distinct memory address, *a*, has been accessed after the latest access of *b*.


time: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
thread-local trace: d a c b c g | e f a f b a | b a g a
count 5 | 4 | 3
hash table

element	tid	first	latest	next
d	1	0	4	
a	1	1	3	
	2	2	0	
	3	1	0	
b	1	3	2	
	2	3	1	
	3	0	2	
c	1	2	1	
g	1	4	0	
	3	2	1	
e	2	0	3	
f	2	1	2	

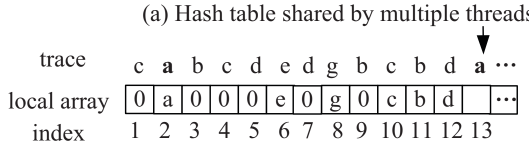
Figure 6: Summary information for thread-local traces (*count* contains the number of distinct memory addresses in each thread-local trace. For each memory address v in the hash table, the *first* column lists the number of other memory addresses accessed before the first access of v in each thread-local trace, and the *latest* column lists the number of addresses accessed after the latest access of v.

D. Merging Thread-Local Results

Figure 7 shows our algorithm for merging reuse distances computed by different threads. The algorithm takes a single input, the collection of reuse distances computed so far, and

```
struct hashnode {
    int64          addr;
    access_info    *tinfo;
    hashnode       *next;
}
struct access_info {
    int32          tid;
    int32          first;
    int32          latest;
    access_info    *next;
}
```

(a) Hash table shared by multiple threads


trace
c **a** b c d e d g b c b d **a** ···
local array
0 a 0 0 0 e 0 g 0 c b d ···
index
1 2 3 4 5 6 7 8 9 10 11 12 13

(b) Thread-private local array. It is dynamically updated as the trace being traversed. Above example shows the array status before the pointed **a** is processed. When we traverse to trace[i], the j-th (0<j<i) array element is defined as:
0: not the latest access to trace[j] by now
trace[j]: the latest access to trace[j] by now

```
algorithm Thread_Analysis (trace)
//Global hash table, and global array count holding number of
//distinct elements for each thread
global hash, count[];
1. //Initialize local_array from the input trace
   local_array = Copy_To_Local(trace);
2. //process each memory access in sequential order
   for each index i of local_array do
      v = local_array[i];
      $t_v$ = Get_Last_Access(hash, threadID, v);
      //Compute reuse distance for v
      dist = Compute_Reuse_Distance(local_array, $t_v$, i);
      local_results = Store_Result(dist, v);
      //Reset v's previous access in the local array
      if ($t_v$ != 0)          local_array[$t_v$] = 0;
      //Update v's access_info
      Record_Access_Info(hash, threadID, v, i);
   end for
   //Count the number of distinct elements in the trace
3. count[threadID] = Count_Distinct_Elements(local_array);
   return local_result;
end algorithm
```

(c) Algorithm for thread-level analysis

Figure 5: Hybrid hash table and local arrays, together with the algorithm for thread-level analysis.

extends the collection by considering situations where a pair of references to the same memory address span across multiple thread-local traces. Such information is recovered from the set of *access_info* objects stored in the global hash table, illustrated in Figure 5(a) and discussed in Section III-B1.

In particular, for each memory address v stored in the hash table, the algorithm traverses the *access_info* objects of v in increasing order of their thread *id*s (i.e., in the original ordering of the corresponding thread-local traces). If consecutive *access_info* objects are created by two distinct threads with ids i and j respectively, where $i < j$, the reuse distance between the last reference of v in the ith local trace and the first reference of v in the jth local trace can be computed as:

$$dist(v) = Get\_Access\_Info(v,i).last + \sum_{k=i+1}^{j-1}(count[k]) + Get\_Access\_Info(v,j).first \tag{1}$$

where $Get\_Access\_Info(v,i)$ returns the $access\_info$ object of memory address v created by thread i, and *count[k]* contains the number of distinct memory addresses processed by thread k. Figure 8 shows an example of merging the thread-level results.

While Formula 1 can be used to extensively recover reuse distances omitted by thread-level analysis, the result could still remain imprecise and thus require the invocation of another routine, $Adjust\_Distance$ in Figure 7, to further recover dependence constraints that may have been violated by the thread-level analysis. Details of the adjustment is discussed in Section III-E.

```
algorithm Merge_Results (local_result)
global hash, count[]
   for each memory address v in hash do
      dist = 0;
      //process each thread-local trace in increasing order,
      //compute dist using Formula 1.
      for i = 1 to num-of-threads do
         if (there exists a node in v's access_info, whose tid=i) then
            dist = dist + Get_Access_Info(v, i).first;
            dist = Adjust_Distance(dist);
            result = Store_Result(dist, v);
            dist = Get_Access_Info(v, i).last;
         else
            dist += count[i];
         end if
      end for
   end for
  //Add local result into result.
  result = Add_Results(result, local_result);
  return result;
end algorithm
```

Figure 7: Algorithm for merging thread-level results.

E. A Probabilistic Model For Adjusting Solutions

Figure 8 shows an example where using the algorithm steps so far, an incorrect reuse distance value, 6, would be returned for a memory reference, g, instead of its correct reuse distance which should be 4. The loss of precision comes from missing the overlap of distinct memory addresses inside the 2nd and 3rd thread-local traces (a and b exist in both thread-local traces). Since a straightforward

element	merging	note
a	3+2=5	Get_Access_Info(a,1).last+Get_Access_Info(a,2).first
	0+1=1	Get_Access_Info(a,2).last+Get_Access_Info(a,3).first
b	2+3=5	Get_Access_Info(b,1).last+Get_Access_Info(b,2).first
	1+0=1	Get_Access_Info(b,2).last+Get_Access_Info(b,3).first
c	—	—
d	—	—
e	—	—
f	—	—
g	4+2=6	Get_Access_Info(g,1).last+count[2] +Get_Access_Info(g,3).first

Figure 8: Example for merging thread-level results.

merging step ignores such overlap, the reuse distance ($dist$) computed from Formula 1 could be larger than the actual value ($dist_{real}$). We correct the result using the following formula:

$$dist_{real} = dist * ef \qquad if \quad dist > M \tag{2}$$

where M is the number of distinct memory addresses in the entire original trace t, and ef is called as an *effective factor* of t. Intuitively, for an input trace t, ef represents the probability that there is no overlap between two arbitrary thread-local traces taken from t. Below, we discuss how to compute ef.

Given a trace t of N references to M memory addresses, assume t has been equally divided into *n* thread-level traces, t_1, ..., t_n, each with a set of local memory references S_1, S_2, ... S_n, respectively. We define the *effective factor* ef of t as:

$$ef = \frac{M}{\Sigma_{i=1}^{n} count[i]} \tag{3}$$

where $count[i]$ is the number of distinct memory addresses in S_i. Note that the number of distinct memory addresses in ($\bigcup_{i=1}^{n} S_i$) is M. Hence the effective factor (ef) of t is computed as the inverse of the average number of times that an arbitrary memory address may appear simultaneously across different thread-local traces. (i.e., inverse of the likelihood that two arbitrary thread-local traces may overlap). The value of ef ranges from 0 to 1. Its upper bound (ef_{max}=1) can be found when $\forall(i,j), S_i \cap S_j = \emptyset$; i.e., there is no overlap among the thread local traces. Its lower bound ($\lim_{n\to\infty} ef_{min} = 0$) can be found when there is a thread-local trace S_k that subsumes all the other local traces; i.e.,

$$ef_{min} = \frac{count[k]}{\Sigma_{i=1}^{n} count[i]} \tag{4}$$

$$when \quad \exists k \in (1..n), s.t. \quad \forall j \neq k, S_j \subset S_k$$

F. Correctness And Generality

Some of the reuse distances computed by our HP-RDA algorithm could be slightly different from those computed by the sequential algorithm in Section II. However, after applying the probabilistic model in Section III-E, the loss of precision is negligible in a majority of practical situations, as confirmed by our experimental results.

While our algorithm applies only to the reuse distance analysis algorithm, the overall approach, including the redesign of the data structures and the adjustment of solutions based on a probabilistic model, can be leveraged for other trace analysis problems, e.g., pattern seeking in a given trace (used in biological areas for protein analysis) [22], trace-based cache/memory bank simulator [29], [19], etc.

IV. Algorithm Implementation on GPU

We have specialized our implementation of the HP-RDA algorithm for the NVIDIA GPGPU architecture by accommodating varying architectural features such as the two layer thread block and thread level parallelism, the hierarchical memory system, among others.

A. Two-levels Of Parallelization

CUDA supports parallelism at two levels, the thread block level and the thread level. Accordingly, we first divide the input trace into multiple thread-block-level traces and then divide each thread-block-level trace into many thread-level traces. Figure 9 illustrates the distribution of tasks among the threads. Note that the workload being distributed include not only the analysis of thread-local traces but also the merging of local results at both the thread and thread-block levels. Further, the workload assigned to each thread varies at different algorithmic steps. At the thread-level analysis step, each thread processes a thread-local trace. At the thread-level merging step, each thread processes a group of distinct elements from the hash table of its parent thread block. Finally, at the thread-block-level merging step, each thread processes a group of distinct elements across all the hash tables from different thread blocks.

B. Pool-Based Dynamic Memory Allocation

Our HP-RDA algorithm needs to dynamically allocate memory to store a large amount of data, e.g., the hash table nodes and the summary information of thread-local threads. However, CUDA does not support *malloc*. To resolve this issue, we implemented a lightweight pool-based dynamic memory allocator on GPU and provided support for efficient lock-free concurrent insertions to the hash table using the approach by Hong et. al [11]. Our implementation allocates a large block of global memory, which acts as two memory pools, for each thread block. Each memory pool uses a *free-space-ptr* pointer to keep track of the free spaces. Every time *malloc* is invoked from a CUDA kernel, the *free-space-ptr* is atomically incremented using the *atomicAdd()* operation provided by GPU. One of the memory pools is dedicated to hold hash nodes, enabling us to directly count the number of distinct memory addresses in each thread-block-local trace from the memory pool. The other pool is used for other miscellaneous data, including the *access_info*

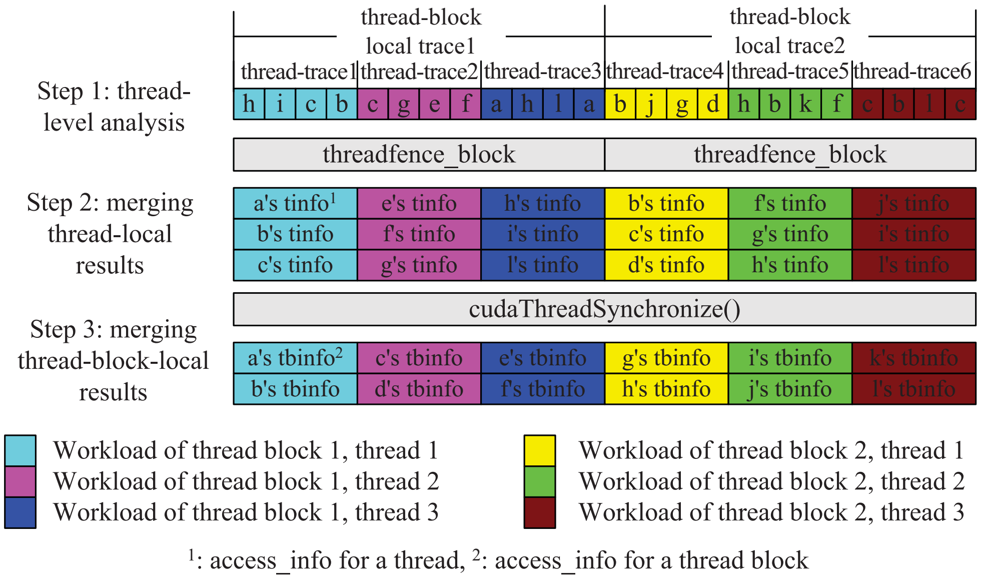

thread-block local trace1
thread-block local trace2
thread-trace1
thread-trace2
thread-trace3
thread-trace4
thread-trace5
thread-trace6
Step 1: thread-level analysis
h i c b c g e f a h l a b j g d h b k f c b l c
threadfence_block
threadfence_block
Step 2: merging thread-local results
a's tinfo[1] e's tinfo h's tinfo b's tinfo f's tinfo j's tinfo
b's tinfo f's tinfo i's tinfo c's tinfo g's tinfo i's tinfo
c's tinfo g's tinfo l's tinfo d's tinfo h's tinfo l's tinfo
cudaThreadSynchronize()
Step 3: merging thread-block-local results
a's tbinfo[2] c's tbinfo e's tbinfo g's tbinfo i's tbinfo k's tbinfo
b's tbinfo d's tbinfo f's tbinfo h's tbinfo j's tbinfo l's tbinfo
Workload of thread block 1, thread 1
Workload of thread block 1, thread 2
Workload of thread block 1, thread 3
Workload of thread block 2, thread 1
Workload of thread block 2, thread 2
Workload of thread block 2, thread 3
[1]: access_info for a thread, [2]: access_info for a thread block

Figure 9: Workload distribution for HP-RDA. Each color represents the workload for a thread, varying with steps. And the grey horizontal strips represent synchronization operations.

for each memory reference, the *next* pointer used to link the access information together (as shown in Figure 5(a)), and some other temporary variables. The memory pools are be released until the reuse distance analysis for the whole input trace has been completed.

C. Memory Utilization

Inside the NVIDIA GPU, each CUDA thread can access a private local memory, a shared memory owned by all threads of the same thread block, and a global memory shared by all threads running on the same GPU [1]. Our implementation allocates the dynamic memory pools on the global memory (GM). Therefore the hash table, together with the summary information of all thread-local threads, are all located on the GM. The frequently used *free-space-ptr* variables, the total count of distinct elements for each thread-local trace, and the histograms holding the analysis result of each thread-local trace, are all allocated on the shared memory for fast access. The thread local arrays and temporary variables are placed in the thread-local memory.

V. Performance Evaluation

Through experimental evaluation, we compare our HP-RDA algorithm with the sequential reuse distance analysis algorithm by Ding et. al [9] and seek to confirm two conclusions: (1) our HP-RDA algorithm offers significant performance advantages over the conventional sequential reuse distance analysis approach, and (2) the potential loss of precision using our parallel algorithm on GPUs is minor and likely negligible.

A. Experimental Methodology

Binary Instrumentation. Both the sequential reuse distance analysis algorithm and our HP-RDA algorithm [9] are implemented as plugins for the PIN binary instrumentation system [16]. The instrumented application collects all the memory addresses referenced, combines them into fragments of memory traces, and then feed these fragments of traces to its plugin for processing.

Platform. We ran our experiments on a Intel 2.13GHz quad-core Xeon E5506 with 32KB L1 DCache, 32KB ICache, and 4MB L2 cache. The machine comes with a NVIDIA Fermi Tesla C2050 GPGPU from NVIDIA. The GPGPU has a 3GB global memory and 14 Streaming Multi-processors(SMs), each containing 32 Streaming Processors(SPs). Each SM has 32768 registers and a 48KB local scratchpad memory shared by all active threads of the SM. We evaluated the sequential reuse distance analysis algorithm on the host CPU and evaluated our HP-RDA algorithm on the GPGPU. Additionally, to isolate the algorithmic advantage of our HP-RDA algorithm from the extra degrees of parallelism offered by GPUs, we also evaluated an OpenMP implementation of our divide-and-conquer algorithm on the host CPU. The instrumented application and the collection of memory traces were always evaluated on the host CPU.

Workloads. We evaluated both the sequential reuse distance analysis and our HP-HDA algorithms using selected programs from a set of CPU SPEC2000 benchmarks, using the *test* inputs of the benchmarks.

B. Algorithm Efficiency

Figure 10 presents the speedup that our HP-RDA algorithm achieved over the sequential reuse distance analysis algorithm, using both a GPU implementation and an OpenMP implementation on CPUs. Here the performance statistics include the time spent analyzing the traces using both algorithms but omit the cost of instrumentation and trace collection. From Figure 10, when running on a GPU platform, our HP-RDA algorithm can achieve up to a factor of 33 speedup, with a factor of 19.6 speedup in average. When implemented using OpenMP on the host CPU, our algorithm shows a factor of 3 speedup in average.

Figure 11 presents speedups of evaluating the whole instrumented application when using our HP-RDA over using the sequential algorithm. Here the performance statistics include both the time spent analyzing the traces and the cost of instrumentation and trace collection. As shown in the figure, even when amortized over the cost of instrumenting and evaluating the user application, our HP-RDA algorithm can achieve up to a factor of 22 speedup, with a factor of 15.1 speedup in average, when evaluated on the GPU. Essentially, our GPU parallelized HP-RDA algorithm can reduce the time required for reuse distance analysis from hours to minutes, significantly reducing the wait time for such analysis and thus improving the productivity of developers. The speedup of its OpenMP implementation on the host CPU is 1.85 in average when 4 threads are used.

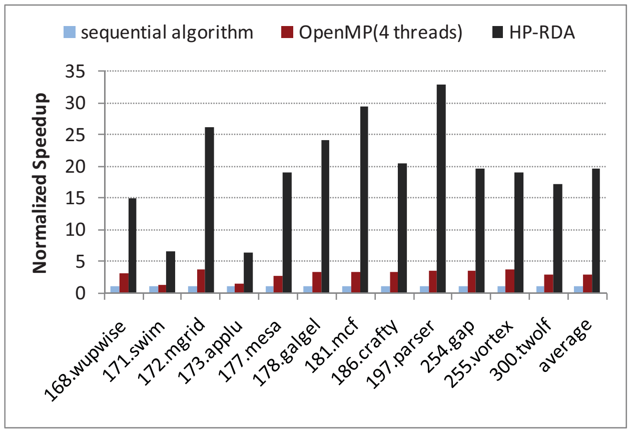


Figure 10: Normalized speedup of HP-RDA over the sequential algorithm. Performance statistics include data transfer between CPU and GPU but omit time spent in instrumentation and trace collection.

C. Accuracy Of Results

To estimate the accuracy of results computed by our HP-RDA algorithm, Figure 12 shows the resulting histograms generated by our HP-RDA (with and without the result correction step) and the sequential algorithms for 6 floating point SPEC benchmarks. Figure 13 shows the same comparison for 6 integer SPEC benchmarks. In Figure 12, three benchmarks, 168.wupwise, 173.applu, and 178.galgel, show visible differences in the results computed by our

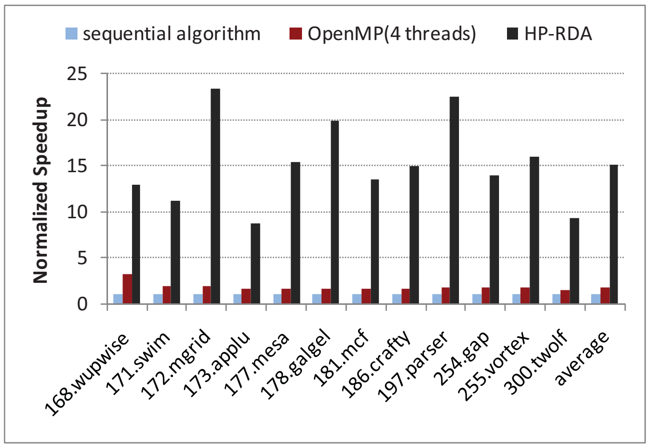


Figure 11: Normalized speedup of evaluating the whole instrumented application with HP-RDA over using sequential algorithm. Performance statistics include data transfer between CPU and GPU and the cost of instrumentation and trace collection.

HP-RDA algorithm when the final solution adjustment step (see Figure 4) is omitted. However, when including the adjustment step, the difference is reduced to a negligible degree. Similar behavior can be observed for the three integer benchmarks, 181.mcf, 197.parser, and 255.vortex, in Figure 13.

To quantify the degree of differences between the results computed by our HP-RDA and the sequential algorithms, we use H bins to divide each of the three histograms computed by the varying algorithms. For each $i = 0, ..., H$, if the sequential algorithm places a_i values inside the ith bin of its histogram, and our HP-RDA algorithm places b_i values to the corresponding bin, we compute an average error rate e as:

$$e = \Sigma_{k=1}^{H}(h_k)/H \qquad h_k = \mid a_k - b_k \mid$$

Figure 14 shows the average error rates of HP-RDA with and without the final solution adjustment step. In particular, the adjustment step was able to reduce the average error rate from 1.2% to 0.37% in average and under 1% in all cases.

Note that even without the probabilistic model based solution adjustment step, the error rate is relatively small. Therefore, we provide an extra option *–disable-adjusting* to turn off the adjustment step to further reduce the runtime overhead of our algorithm when desired by developers.

D. Performance Breakdown

Figure 15 breaks down the performance statistics of our HP-RDA algorithm into four components based on the time spent in local thread-level analysis, merging thread-level results, merging thread-block-level results, and final adjustment of solutions. Here for most benchmarks, thread-level local analysis is the most time-consuming. This is not surprising, since each thread needs to create the hash table,

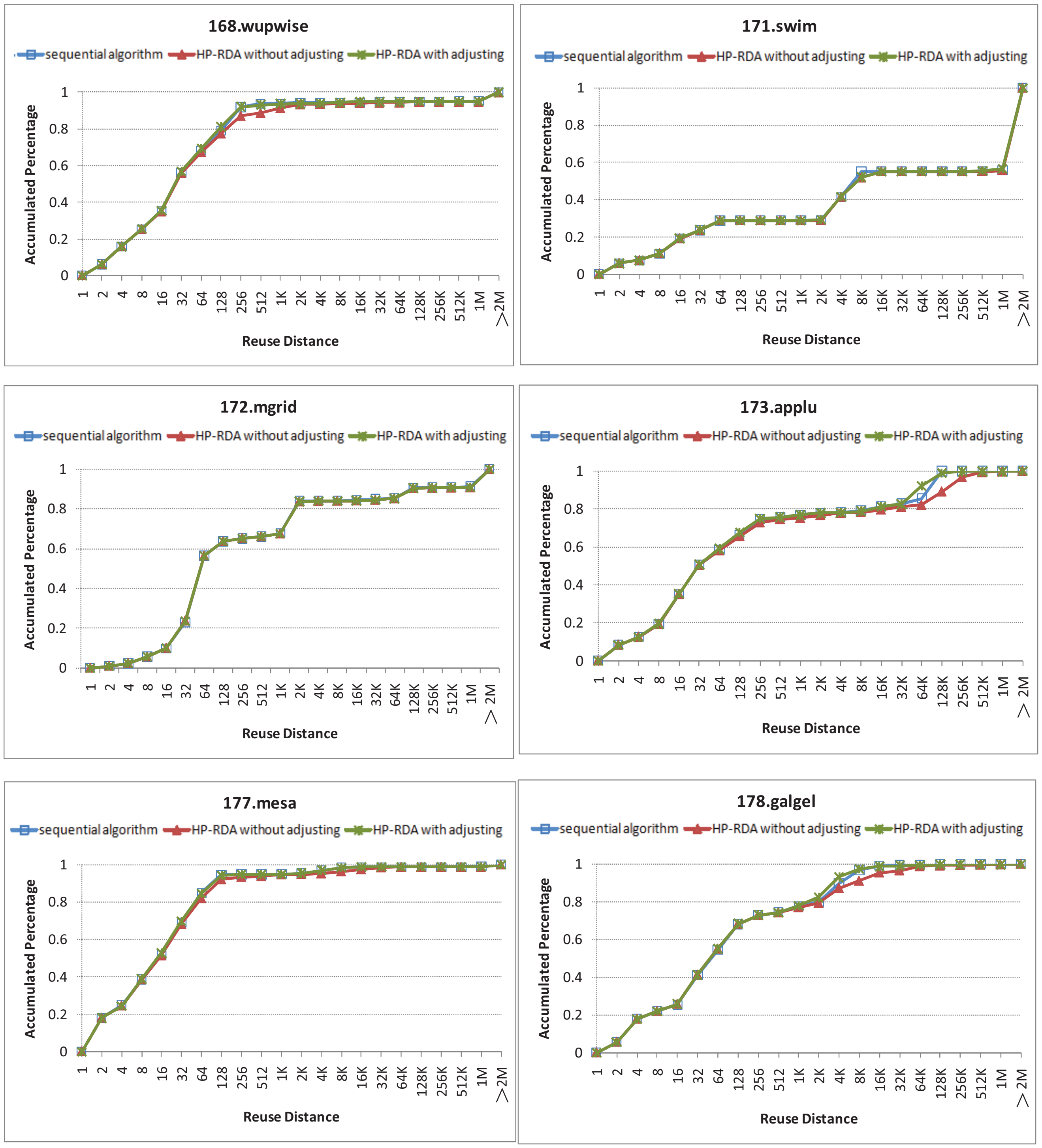


Figure 12: Reuse distance histograms computed for 6 benchmarks in CFP2000 by sequential algorithm and HP-RDA with and without the final adjustment step. X axis is reuse distance, Y axis is the fraction of references with reuse distance less than or equal to x.

maintain the hash nodes, scan the local array, and compute reuse distances inside its local trace. All these operations incur frequent global memory references and thus are among the most significant sources of execution time. A single exception is the 171.swim benchmark, which references a large number of memory accesses without much reuse inside

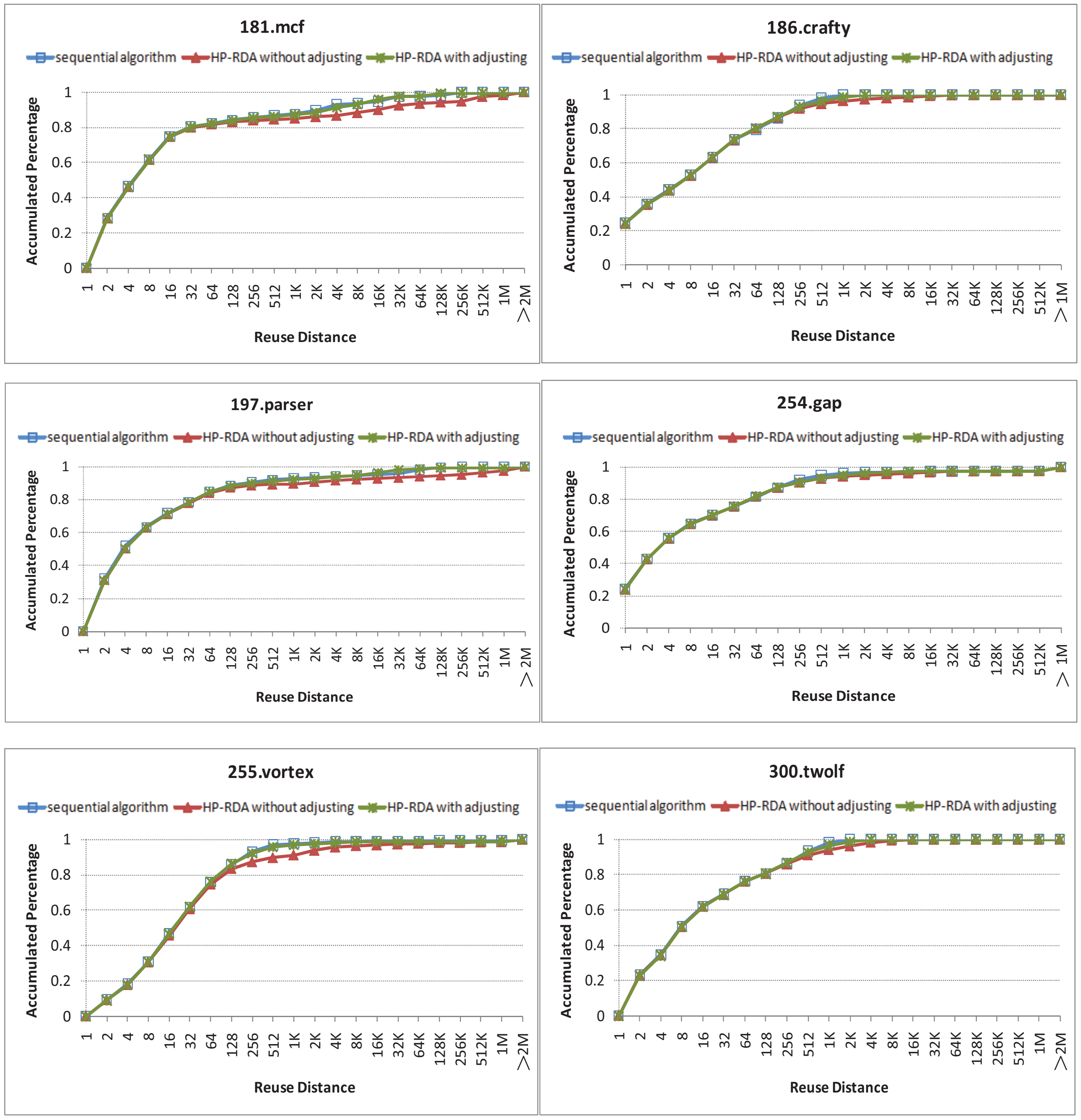


Figure 13: Reuse distance histograms computed for 6 benchmarks in CINT2000 by sequential algorithm and HP-RDA with and without the final adjustment step. X axis is reuse distance, Y axis is the fraction of references with reuse distance less than or equal to x.

thread-level local traces (shown in Figure 13).

E. Tuning Parameters

Our HP-RDA algorithm is parameterized by the following architecture-specific parameters which we manually tuned to achieve satisfactory performance on GPUs.

- **trace size, denoted as** N**,** is the size of memory traces to be transfered from CPU to GPU one at a time; i.e., the size of the memory trace that the instrumented application each time uses to invoke the reuse distance analysis plugins. This parameter is constrained by the global memory size of GPU. We determined its value

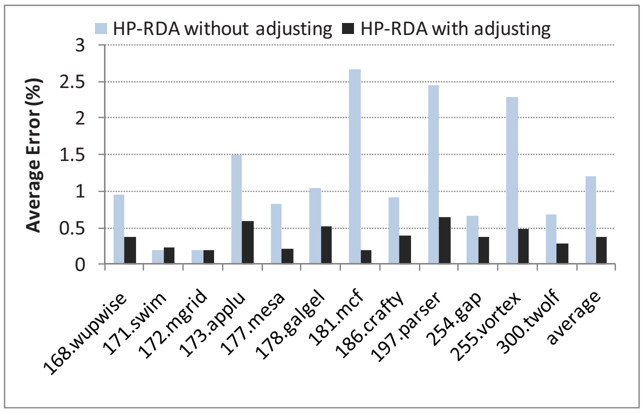


Figure 14: Average error rates of HP-RDA with and without solution adjustment, comparing with sequential algorithm.

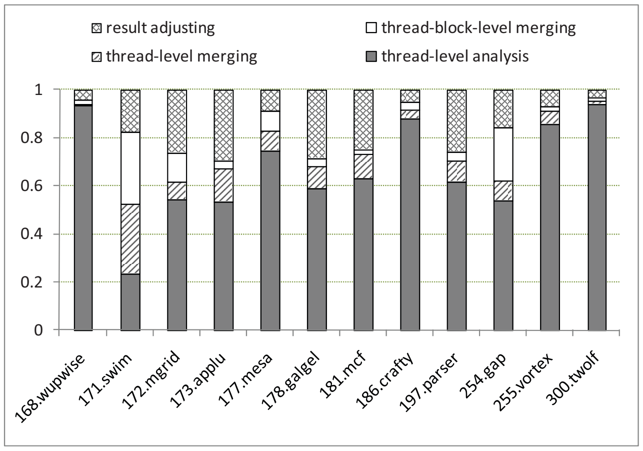


Figure 15: Performance breakdown of HP-RDA.

under the constraint $36 * N < 3GB$ and set its value to be 0x4000000 for convenience.

- **Hash table size, denoted as** ***HS*****.** To efficiently hash the memory addresses from the input trace, we set the hash table size to be 2*N, where N is the size of the input memory trace.
- **Memory pool size, denoted as** ***MS*****.** Our algorithm implementation on the GPU uses two memory pools for dynamic objects allocation. Distinct elements are allocated in the memory pool for hash nodes, and their repeated appearances in the other pool. We allocate the total memory pool to hold N elements, and the memory consumption is estimated as 20*N bytes, where N is the size of the input memory trace.
- **Size of thread-level trace, denoted as** ***ST*****.** This parameter determines the workload of each CUDA thread. When *ST* is too small, the accuracy of the algorithm could be reduced. We determined the value of this parameter together with the CUDA threads organization. Suppose the number of CUDA thread blocks is NTB, and the number of threads per block as NB, the constraint $NTB * NB * ST = N$ is enforced. Further, we fixed NB to be 64, as a larger value could cause the GPU shared memory to be exhausted. This parameter is sensitive only to the underlying GPU architecture and is independent of the user application being analyzed.

Figures 16 and 17 show the accuracy and execution time under different configurations. It can be observed that the accuracy of results decreases when ST becomes smaller. In particular, when ST = 16384, the analysis result is very close to that of the sequential algorithm. Combining Figures 16 and 17, we can set ST to be 2048, which was the configuration used for our evaluation.

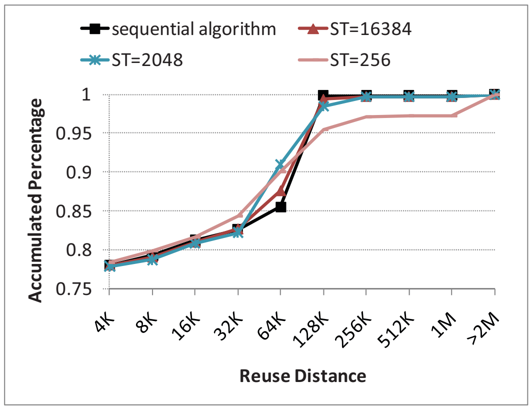


Figure 16: Accuracy varying with ST .

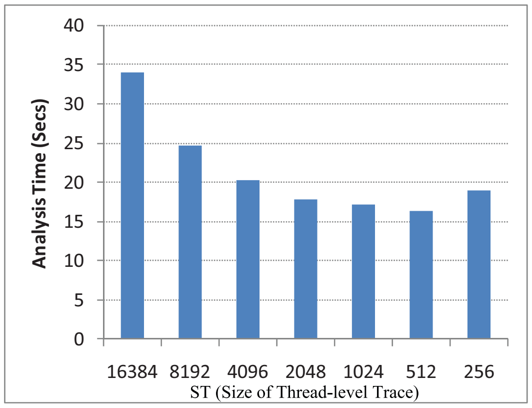


Figure 17: Analysis time varying with ST.

VI. Related Work

A. Reuse Distance Analysis Algorithms

Reuse distance analysis was originally introduced by Mattson Et. al [18] under the name *stack distance*, and has been widely-used to model the reuse of data in caches on modern architectures [8], [9] and to understand the memory system behavior of applications. Ding et. al used reuse distance to predict whole-program locality of applications by revealing their global reuse patterns [9]. Shen et. al used the analysis to identify program locality phases [25]. Zhong, Shen, and Ding then further developed locality analysis to generate a parameterized model to predict cache miss rates across different program inputs [34].

Compiler researchers also leveraged reuse distance analysis to guide memory-aware optimizations. Beyls et. al used reuse distances to generate cache hints for load/store instructions [6]. Li et. al used reuse distances to evaluate the potential benefits of register allocation for array elements on scalar processors [15]. Marin et. al used reuse distances to identify significant memory access patterns causing cache misses and to provide insight for improving data reuse [17]. Zhong et. al defined a k-distance analysis to guide array regrouping and structure splitting [35].

As computer architectures evolving towards multicores, Ding and Chilimbi focused on statistical modeling of multithreaded reuse distances by combining data sharing and thread interleaving information with per-thread reuse distance analysis [7]. Schuff et. al used the sampling method to go beyond statistical modeling and to track interactions between threads [23]. For multi-processor programs, existing research has focused on modeling destructive interferences among separate processes contending for limited cache resources [14], [28].

B. Accelerating Reuse Distance Analysis

Researchers have used a number of data structures, including m-ary tree [4], blocked hashing [4], AVL-tree [20], and splay tree [26], to promote the efficiency of reuse distance analysis. Ding et. al also leveraged approximate reuse distance analysis to reduce the analysis cost [9]. However, even under the best implementations, analyzing every memory reference of a program slows down its evaluation by at least 1-2 orders of magnitude [26].

A common approach to accelerating reuse distance analysis is through sampling, which randomly selects a number of instructions and a trigger to control the start of the analysis. Zhong and Chang [33] presented a sampling-based method which organizes data accesses into a tree and then separates the analysis into a “sampling period” and a “hibernation period”. The tree is modified only during sampling period and is read to compute reuse distances only during hibernation. Schuff et. al used sampling in their multicore reuse distance analysis algorithm [23]. While sampling accelerates reuse distance analysis by controlling the trace generation, our focus is on parallelizing the trace analysis to take advantage of the massive number of parallel processing nodes in GPUs.

C. Using GPU to Accelerate Irregular Applications

Most irregular applications require customized optimization when porting to GPUs. Prabhu et. al [21] presented a linear-algebraic encoding approach for higher-order control-flow analysis. Solomon and Thulasiraman [27] analyzed the performance of porting two irregular applications, matrix parenthesization and breadth first search, to GPUs. Joseph et. al [13] presented a parallel implementation of the Particle in cell (PIC) algorithm on GPUs.

While hash tables are widely-used on traditional CPUs, their implementations on GPUs are not straightforward. Hong et. al [11] presented a hash table implementation on GPU together with a lightweight memory allocator. Zhang et. al [32] presented a hash table implementation on GPU both with and without atomic operations supported. Amossen and Pagh [3] introduced a new data layout, BATMAP, to accelerate item set mining for set intersections.

Many researchers have parallelized tree searching or traversal algorithms, e.g., the k-D tree traversal algorithm [12], decision trees and forests [24], and B+ tree search [10], on GPUs. However, we are not aware of any other existing algorithm for implementing balanced tree creation and rotations on GPUs.

VII. Conclusion

This paper presents a highly-parallel reuse distance analysis algorithm (HP-RDA), which reformulates the sequential algorithm so that massive parallelism can be exploited using the SPMD execution model on GPUs. We have used a hybrid data structure of hash table and local arrays to flatten the traditional tree representation of memory access traces, and have used a probabilistic model to correct any loss of precision from a straightforward parallelization of the original sequential algorithm. The HP-RDA algorithm is evaluated on a Fermi platform, and our experimental results show that up to a factor of 20 speedup can be achieved comparing with a sequential implementation of the algorithm, with less than 1% average error.

Acknowledgements

This research is supported in part by the Chinese National Basic Research Grant 2011CB302504, the Innovation Research Group of NSFC 60921002, the National Science and Technology Major Projects of China 2009ZX01036-001-002 and 2011ZX01028-001-002, the National Natural Science Foundations of China under Grant 60970024, the US National Science Foundation under Grants 0833203 and 0747357, the US Department of Energy under Grant DE-SC001770, and the Australian Research Council under Grant DP110104628.

References

[1] Nvidia cuda programming guide 2.0. http://www.nvidia.com/object/cuda_develop.html.

[2] G. Almasi, C. Cascaval, and D. A. Padua. Calculating stack distances efficiently. In *MSP*, Berlin, Germany, 2002.

[3] R. Amossen and R. Pagh. A new data layout for set intersection on gpus. In *IPDPS*, Anchorage, Alaska, USA, 2011.

[4] B. T. Bennett and V. J. Kruskal. Lru stack processing. Technical report, IBM Journal of Research and Development, 1975.

[5] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *IASTED*, pages 617–662, 2001.

[6] K. Beyls and E. D'Hollander. Generating cache hints for improved program effciency. *Journal of Systems Architecture*, pages 223–250, 2005.

[7] C. Ding and T. Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical report, Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[8] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *IPDPS*, 2001.

[9] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, San Diego, California, USA, 2003.

[10] J. Fix, A. Wilkes, and K. Skadron. Accelerating braided b+ tree searches on a gpu with cuda. In *A4MMC*, 2011.

[11] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *PACT*, 2010.

[12] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree gpu raytracing. In *I3D*, 2007.

[13] R. Joseph, G. Ravunnikutty, S. Ranka, E. D'Azevedo, and S. Klasky. Efficient gpu implementation for particle in cell algorithm. In *IPDPS*, Anchorage, Alaska, USA, 2011.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, Washington, DC, USA, 2004.

[15] Z. Li, J. Gu, and G. Lee. An evaluation of the potential benefits of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjuction with the HPCA-2*, San Jose, California, 1996.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[17] G. Marin and J. Mellor-Crummey. Pinpointing and exploiting opportunities for enhancing data reuse. In *ISPASS*, 2008.

[18] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. Technical report, IBM Sysmte Journal, 9(2), 1970.

[19] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multiprocessor server environment. In *WMPI*, New Yor, NY, USA, 2004.

[20] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical report, Lawrence Berkeley Laboratory,, 1981.

[21] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. Eigencfa: Accelerating flow analysis with gpus. In *POPL*, Austin, USA, 2011.

[22] B. Rehm. Bioinformatic tools for dna/protein sequence analysis, functional assignment of genes and protein classification. *CHEMISTRY AND MATERIALS SCIENCE*, 2001.

[23] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *PACT*, Vienna, Austria, 2010.

[24] T. Sharp. Implementing decision trees and forests on a gpu. In *ECCV*, 2008.

[25] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, New York, USA, 2004.

[26] D. D. Sleator and R. E. Tarjan. Self adjusting binary search trees. *Journal of the ACM*, 1985.

[27] S. Solomon and P. Thulasiraman. Performance study of mapping irregular computations on gpus. In *IPDPSW*, Atlanta, GA, USA, 2010.

[28] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *ICS*, 2001.

[29] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation a survey. *ACM Computing Surveys*, 1997.

[30] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Boston, 2000.

[31] J. Xue and C. Huang. Reuse-driven tiling for improving data locality. *International Journal of Parallel Programming*, pages 671–696, 1998.

[32] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Gpu-accelerated text mining. In *EPHAM*, Seattle, Washington, USA, 2009.

[33] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *ISMM*, New York, NY, USA, 2008.

[34] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *PACT*, 2003.

[35] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, New York, NY, USA, 2004.