ISSN No. 0976-5697

Volume 7, No. 1, January-February 2016

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Coalesced Redundant Block Synchronization on GPGPU

Suma S

Bharathiar University, Coimbatore,India

Abstract: Speculative parallelization is a mainstream technology used to increase the instruction level parallelism and the thread level speculation. Since the computations are executed faster on GPU's, the application programs can be still executed faster in combination and increase the performance. The execution order is rearranged to increase the performance enhancements through the load/store pair dependencies determined in a different categories to increase the instruction level parallelism. In this paper it is proved that how speculation reduces the memory latencies and increase in instruction execution through the proof with merge sort program using the cuda-memcheck and CUDA-GDB. The profiler GPROF is used for sequential program and the parallel program uses CUDA which uses the mechanism of synchronization through coalesced redundant blocks on GPU.

Keywords-: synchronization, GPGPU, instruction level parallelism, thread level parallelism, performance, redundant blocks, coalescing.

I .INTRODUCTION

Instruction level parallelism has increased tremendously the execution speed and the performance of the computer system. According to Moore's law the numbers of transistors are increasingly over the microprocessor chip. The clock speed of the system is also increasing. The Graphic Processors have tremendous scope in increasing the performance, scalability and increases the granularity of the code also.

Along with the CPU the future microprocessors include both CPU and GPU for the general purpose applications referred as GPGPU's. The fine grained parallelism has good impact on the GPGPUs. The general purpose graphic processors have been serving the industry for general applications also.

All modern high performance processors need to exploit the instruction level parallelism where the instructions are executed. The execution time can be reduced by executing instructions in parallel and avoiding execution while instruction takes its time for the execution. The tolerance to the memory latency is violation of program semantics before the instruction level parallelism if a load action is executed prior to store instruction that writes to the same memory location is determined as true dependence with a preceding store. The load is allowed to execute speculatively before store to increase the performance such that it may be data dependent, the true memory dependence is not violated, the speculation would be successful else it results in the erroneous execution. The best performing technique is memory dependence speculation and synchronization. With this technique the loads are speculated whenever it is needed. When the misspeculations encountered, the information regarding the dependences are recorded in the predictive structure.

The ILP techniques are useful in sending load requests for tolerating the memory latency. The ILP exploitation execution should not violate the program semantics before the instruction is executing.

High performance can be achieved by the dependences on the memory and the speculation with parallelization.[1].

II Speculative Memory Dependences

The Speculative parallelization consists of speculative and nonspeculative threads for the execution of computations. The nonspeculative thread executes normally whereas speculative threads executes parallel part and if the speculation is correct, the execution of the rest of the instruction is avoided else the nonspeculative thread resumes the normal execution of the program squashing the predicted results. It does not change the architectural state of the system and the execution condition resulting in system crash. Hence it is a promising technique to improve ILP with Thread level speculation.

Sreepathi Pai R. Govindarajan Matthew J. Thazhuthaveetil have integrated automatic memory manager into the X10 compiler to eliminates redundant memory transfers.[4]

The speculation/synchronization may be used as a potentially lower complexity for exploiting load/store parallelism.

The true dependences and false dependences are categorized into 4 groups.

1. True positive dependences.
2. True negative dependences.
3. False positive dependences.
4. False negative dependences.

1. True positive dependences: means that there exists truly dependence between the store/load operations and it can be predictable also.
2. The True negative dependences: are the memory dependences between load/store operations where a dependency exists but that cannot be predictable as it is indirectly dependent on the load/store pairs.
3. The false positive dependences: are memory dependences between load/store pair where there exists no dependences between the load/store but the predictions can be made.
4. The false negative dependences: are the dependences of load/store pair such that there exists no dependences and it cannot be speculated also such that they are static in nature.

Consider an instruction set for 2 digit BCD no and unpack the BCD no. to store them in two different memory locations. The instruction set for the same problem is

```
LDA 3200H
ANI F0H
RRC
RRC
RRC
RRC
STA 3201H
LDA 3200H
ANI OFH
STA 3201H
HLT
```

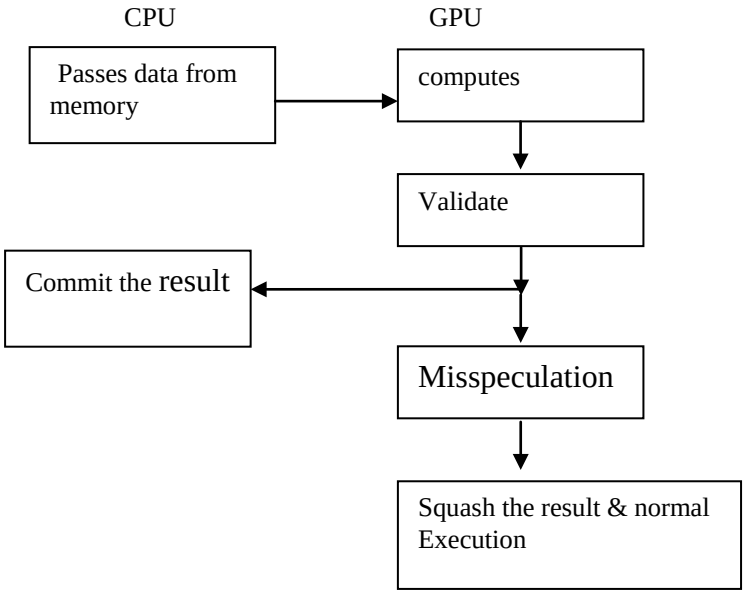
The above instruction set can be speculatively parallelized and executed as

Instruction set With no speculation	Reordered instruction with true positive dependences	
	Nonspeculative	Speculative
LDA 3200H	LDA 3200H	ANI 0FH
ANI F0H	ANI F0H	STA 3201
RRC	RRC	
RRC	RRC	
RRC	RRC	
RRC	RRC	
STA 3201H	STA 3201H	
LDA 3200H		
ANI OFH		
STA 3201H		
HLT		
11 cycles	7 cycles	2 cycles
(a)	(b)	(c)

The above instruction set in (a) which consists of load/store pair with sequential execution takes 11 cycles without speculation. From (b) it is evident that the nonspeculative thread initiates speculative threads, passes state information and data to speculative thread for executing the other part of the instruction set. The speculative threads executes the second part of the instruction set computes the result and before the nonspeculative thread completes its task, the result is updated in the specified memory location saving 2 cycles of fetch and store instruction. Suppose the values computed by the speculative threads results in misspeculation, the result is squashed and nonspeculative thread executes the instruction ANI 0FH only again it saves 1 cycle of execution. According to this instruction set, ANI FOH is treated as the true positive dependent instruction. RRC is treated as true negative dependent instruction as it alone cannot be predicted.

II.I Coalesced Block Synchronization

The speculative execution exploits the instruction level parallelism and thread level speculation. It can also be implemented using both the hardware and software techniques. The parallelizable problems which are embarrassingly parallel can be easily adapted to the architecture which uses CPU and GPU together to enhance parallelism. The diagram of GPGPU for enhancing instruction level parallelism and thread level speculation is provided [6,7]. A model which illustrates with speculative memory latency technique's that the executions of the computations are carried out through the following steps.



The execution model consists of 4 processor cores CPU and GPU. The Intel Inspiron n5050 consists of quad cores processors where CPU can compute the execution of task in both the cores.

We consider CPU as the non speculative and GPU as the speculative computation of the tasks. once the execution of program starts, the process of execution of instructions is that the CPU to transfer data from its memory to the GPU memory for executing the parallel part of the program and the serial part of the program gets executed on CPU only. The results from the GPU are committed back to CPU memory.

Before the execution of the task, the compiler profiles the instructions such that it separates the instructions which are categorized as above. The instructions like false negative dependent, true negative dependent are executed in a different execution order. This reduces many memory fetch load instruction cycles and reduces memory latency and redundancies also. The redundant computations are the computations that are repeated instructions executed repeatedly in an instructions set.

The execution order of instructions is arranged in such a way that the CPU and GPU executes instructions to reduce memory latencies. If the execution results in a correct speculated value the results are committed else the results are squashed and CPU executes the entire set of instructions where it speculated normally. But invalidations also results in less number of memory cycles compared to normal execution.

This can be verified running a sorting program a mergesort with both sequential and parallelized program on both CPU and GPU which works as a co processor and speculative threads. The CUDA-DBG debugger tool is used to check for the memory as well as leakages and dependences. The parallel version of the merge sort uses the array of elements sorted in using odd and even merging where odd indexed elements are grouped together and even indexed elements are grouped together. Then the sorting takes place on the GPU. The GPU is using the technique of allocating the parallel threads in the form of blocks where each block is coalesced as a redundant block to compute odd merge and even merge respectively. The sequential version of the program for mergesort is given below

as well as the parallel version of the odd-even mergesort with CUDA is given below. The programs are run with the CUDA-dbg debugger and the results of memcheck, dependencies are provided in the results.

The sequential algorithm for merge sort is

```
Algorithm mergesort (int k[],int l,int h)
    {     int m;
          If (l<h)
     {
          m=(l+h)/2;
          mergesort (k,l,m);
         mergesort (k,m+1,h);
         Smerge (k,l,m,h);      }
    }

Algorithm Smerge(int k[],int l,int m,int h)
{
    int b,c,d,s[20];
       b=l; c=l; d=m+1;
  while ((b<=m)&&(d<=h))
  {    if ( s[b] < s[d])
       s[b++]=k[c++];
       else
      s[b++]=k[d++];
  }
  while(b<=m)  s[b++]=k[c++];
  while(d<=h) s[b++]=k[d++];
for(b=l;b=k-1;b++) k[b++]=s[b++];
}
```

The parallel merge algorithms is

```
Algorithm pmerge()
{

Allocate blocks and threads.
Even indexed and odd indexed elements are grouped
 The blocks are sorted.
}
```

The parallel merge sort also uses Fourier coefficients when considering the geometric computations. Since the odd and even merge results in 2 periodic functions for odd and even functions.

If f is even $f(x)=a0+\sum an \cos n\pi/p\ x$

Where $a0=i/p\int f(x)dx$ $an=2/p\int f(x)\cos n\pi/2\ x\ dx$

If f is odd $f(x)=\sum bn\sin n\pi/p\ x$

Where $bn=2/p\int f(x)\sin n\pi/px\ dx$.

Results:

```
suma@suma-Inspiron-N5050:~$ gprof merge gmon.outFlat
profile:
Each sample counts as 0.01 seconds.
 no time accumulated
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        3     0.00     0.00  merge
  0.00      0.00     0.00        1     0.00     0.00  mergesort
 %         the percentage of the total running time of the time
program used by this function.
cumulative a running sum of the number of seconds accounted
seconds   for by this function and those listed above it.
 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this   listing.
calls      the number of times this function was invoked, if  this
function is profiled, else blank.
   self      the average number of milliseconds spent in this ms/call
function per call, if this function is profiled, else blank.
 total     the average number of milliseconds spent in this ms/call
function and its descendents per call, if this   function is profiled,
else blank.
name       the name of the function.  This is the minor sort  for this
listing. The index shows the location of
 the function in the gprof listing. If the index is  in parenthesis it
shows where it would appear in  the gprof listing if it were to be
printed
 Call graph (explanation follows)
granularity: each sample hit covers 4 byte(s) no time propagated
index % time    self  children    called     name
                0.00    0.00       3/3          mergesort [2]
[1]     0.0    0.00    0.00       3         merge [1]
-----------------------------------------------
                                   6           mergesort [2]
                0.00    0.00       1/1          main [6]
[2]     0.0    0.00    0.00       1+6       mergesort [2]
                0.00    0.00       3/3          merge [1]
                                   6           mergesort [2] ----------------------
----------------------
This table describes the call tree of the program, and was sorted
by  the total amount of time spent in each function and its
children.
Each entry in this table consists of several lines.  The line with
the index number at the left hand margin lists the current
function.
 The lines above it list the functions that called this function,  and
the lines below it list the functions this one called.  This line lists:
index       A unique number given to each element of the table.
Index numbers are sorted numerically. The index number is
printed next to every function name so
it is easier to look up where the function in the table.
     % time This is the percentage of the `total' time that was spent
in this function and its children.  Note that due to different
viewpoints, functions excluded by options, etc, these numbers
will NOT add up to 100%.    self           This is the total amount
of time spent in this function.
 children  This is the total amount of time propagated into this
function by its children.
 called      This is the number of times the function was called. If
the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.
```

Fig. 1 system performance when the sequential program is working.

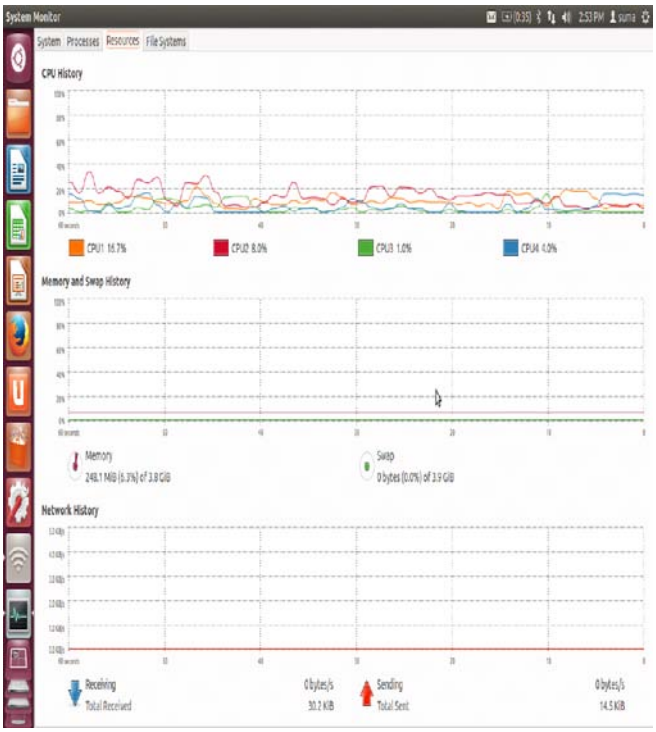


Fig.2 System Performance for the parallel program.

REFERENCES

[1] "Speculative Execution in high performance computer Architectures" David Kaeli and pen-chung yew chapman & hall/CRC Chapter13,14

[2] NVIDIA. CUDA: Compute Unified Device Architecture. URL http://developer.nvidia.com/cuda.

[3] NVIDIA. NVIDIA CUDA C Programming Guide version 4.0. 2011.

[4] "Automatic CPU-GPU communication management and optimization". T. B. Jablin, P. Prabhu, et al. In PLDI, 2011.

[5] "Fast and Efficient Automatic Memory Management for GPUs using Compiler-Assisted Runtime Coherence Scheme" Sreepathi Pai R. Govindarajan Matthew J. Thazhuthaveetil PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA 2012 ACM 978-1-4503-1182-3/12/09.

[6] " Coalesced Speculative Prefetching and Inter thread Data Dependences" Suma S ,N.P. Gopalan, IEEE international Conference on Computer Communication and Informatics (ICCCI 2014) Sri Shakthi Engineering college,Coimbatore,India Jan 3-5 2014. CFP1408R-CDR/ISBN978-1-4799-2352-6/14©2014IEEE.

[7] "Nvidia cuda software and gpu parallel computing architecture," D. Kirk in ISMM '07: Proceedings of the 6th international symposium on Memory management. New York, NY, USA: ACM, 2007, pp. 103–104