Creating memory bandwidth contention with best intentions

George John Chiramel

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Creating memory bandwidth contention with best intentions

George John Chiramel

Heterogeneous System Architecture (HSA) is a computing system architecture that integrates central processing unit (CPU) and graphics processing unit (GPU) with a shared off-chip main memory. On one hand, sharing the memory reduces the communication latency between CPU and GPU but on the other hand, sharing can lead to contention for shared resources. The programs which execute concurrently on the GPU and CPU cores, share the off-chip memory bandwidth. This sharing can result in contention for bandwidth between CPU programs and GPU kernels. The CPU programs can steal bandwidth from GPU kernels which can lead to performance degradation. Since, memory bandwidth is important for the performance of GPU workloads, it is essential to measure the sensitivity of GPU kernels to bandwidth contention from CPU programs.

This thesis describes the design and implementation of a program called Bandwidth Bandit which can steal memory bandwidth from co-running programs. The Bandit which was designed to execute on a CPU, can steal the bandwidth from programs co-running on a CPU or GPU core. The Bandit was used to measure the sensitivity of three GPU applications with different values of bandwidth demand. The results showed that all the three GPU kernels experienced substantial slowdown when subjected to off-chip memory contention due to the co-running CPU application.
Acknowledgments

I sincerely thank Erik Berg from Ericsson for providing support, guidance and confidence throughout the thesis. I would like to thank David Black-Schaffer from Uppsala University for all the guidance and stimulating discussions. I thank Catrin Granbom from Ericsson for the support and motivation. Further, I thank my parents and my wife, Priya for providing motivation and confidence. I would also like to thank Mr. Jonas Carlsson, fellow thesis worker at Ericsson, for all the interesting discussions. Finally I thank my friend Sathya Krishnan for his unconditional support.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>1</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>2</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2. Background Information</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Latency</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Memory level parallelism</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Memory bandwidth</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Memory hierarchy</td>
<td>7</td>
</tr>
<tr>
<td>2.5 Cache Memory</td>
<td>8</td>
</tr>
<tr>
<td>2.5.1 Set associative cache</td>
<td>8</td>
</tr>
<tr>
<td>2.5.2 Cache Replacement</td>
<td>10</td>
</tr>
<tr>
<td>2.5.3 Types of cache misses</td>
<td>10</td>
</tr>
<tr>
<td>2.6 Virtual Memory</td>
<td>10</td>
</tr>
<tr>
<td>2.7 Looking deeper into memory hierarchy</td>
<td>11</td>
</tr>
<tr>
<td>3. Design and Verification of Bandwidth Bandit</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Bandwidth Bandit Method</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Bandwidth Bandit program</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Requirements of Bandit program</td>
<td>15</td>
</tr>
<tr>
<td>3.3.1 A deep dive into bandwidth-MLP relation</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2 A deep dive into conflict misses</td>
<td>17</td>
</tr>
<tr>
<td>3.4 Design of the Bandit program</td>
<td>19</td>
</tr>
<tr>
<td>3.4.1 Detailed design to construct a conflict set</td>
<td>19</td>
</tr>
<tr>
<td>3.4.2 Detailed design to create remaining m-1 conflict sets</td>
<td>20</td>
</tr>
<tr>
<td>3.4.3 Detailed design to traverse through m conflict sets</td>
<td>20</td>
</tr>
<tr>
<td>3.4.4 Intrinsic MLP</td>
<td>23</td>
</tr>
<tr>
<td>3.4.5 Measuring the bandwidth received</td>
<td>23</td>
</tr>
<tr>
<td>3.5 Verification of the Bandit program</td>
<td>23</td>
</tr>
<tr>
<td>3.5.1 Bandit can generate varying amounts of off-chip memory bandwidth</td>
<td>24</td>
</tr>
<tr>
<td>3.5.2 Bandit can steal off-chip memory bandwidth from a target program</td>
<td>25</td>
</tr>
<tr>
<td>3.5.3 More bandit threads can steal more bandwidth from the target program</td>
<td>26</td>
</tr>
<tr>
<td>4. Results</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Target heterogeneous multi-core</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>4.2</td>
<td>Measuring the slowdown of scalar multiplication of vector kernel</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Applying bandwidth bandit method</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparing the slowdown of Black Scholes and Scan kernels</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Target applications</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Applying Bandwidth Bandit method</td>
</tr>
<tr>
<td>5.</td>
<td>Conclusion</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
</tr>
</tbody>
</table>
1. Introduction

A multi-core processor integrates multiple processing units into a single chip where each processing unit is called a core. When all the cores of the multi-core are multiple copies of the same core, it is called as a homogeneous multi-core processor also known as the chip multiprocessor (CMP). When the multi-core processor has cores with different instruction set architecture (ISA), it is known as the heterogeneous multi-core processor. In both these types of multi-core, the main memory lies outside the chip and all the cores share the main memory. Figure 1(a) shows a CMP and 1(b) shows a heterogeneous multi-core.

The off-chip memory bandwidth is critical to the performance of programs which execute on the multi-core. Sharing the bandwidth can lead to contention which can cause negative effects on the performance of co-running applications. Bandwidth contention from an application can cause degradation of the performance of a co-running application. In this thesis, the sensitivity of an application to bandwidth contention is defined as the slowdown experienced by the application due to a specified amount of contention.

Previous studies [1, 2] have shown that, the impact on performance due to memory bandwidth contention can be significant. In another study [3], it was shown that when programs with the same baseline bandwidth consumption are subjected to the same amount of contention, the slowdown experienced can vary quite significantly. While those studies [1, 2, 3] were conducted on CMP, a recent study [12] was conducted on the heterogeneous multi-core processors composed of CPU and GPU cores. Their observation was that, certain GPU applications experienced significant slowdown due to the contention from certain CPU applications.

The importance of measuring the sensitivity of GPU applications to bandwidth contention from CPU applications is shown in figure 2 [12]. In the figure, normalized instructions per cycle (IPC) of GPU applications are shown on the Y-axis. KM, MM and PVR are the GPU...
applications and mcf, omnetpp, perlbench are the CPU applications with memory intensities from most till least. The normalized IPC of each GPU application in the presence of one of the CPU applications is plotted. To understand the effect of contention from CPU application, the IPC of GPU application with no CPU application is also plotted.

The GPU application PVR, has slowed down by 20% due to contention from mcf, 10% due to contention from omnetpp and 0% due to perlbench. This means that the same GPU application PVR has different sensitivity to contention from different CPU applications. The CPU application mcf has caused 10% slowdown for KM, 20% for PVR, but has no effect on MM. This means that the same CPU application mcf causes different slowdown for different GPU applications.

The variation in the effect of different CPU applications on the same co-running GPU application and the variation in the effect of same CPU application on different GPU applications, show the unpredictable nature of the slowdown of GPU applications due to the contention caused by co-running CPU applications. Since the slowdown experienced by a GPU application due to bandwidth contention could be significant, it is important to understand the sensitivity of a given application. Therefore, it is essential to measure the sensitivity of GPU applications to the contention from co-running CPU applications in the laboratory.

Eklov et al. [3] have proposed the Bandwidth Bandit method to measure sensitivity to bandwidth contention. Bandwidth Bandit method involves co running the target application with an application called Bandit which generates memory bandwidth contention. The bandit generates contention by consuming bandwidth from the shared off chip memory bandwidth. The execution time of the target and bandwidth consumed by the bandit can be recorded. The slowdown experienced by the target for various values of bandwidth of the bandit is calculated. In this way, sensitivity of the target can be measured as a function of the amount of contention.

In the paper [3], which proposed the Bandwidth Bandit method, the evaluation was done by measuring the sensitivity of a CPU application to contention from a co-running CPU application on a CMP. The goal of this thesis is to apply the bandwidth bandit method to measure the sensitivity of a target GPU kernel to the contention from co-running CPU applications on a heterogeneous multicore processor. The target GPU application was chosen as a vector scalar multiplication program written in OpenCL. The target heterogeneous
multicore processor was the Accelerated Processing Unit (APU) namely AMD A8-6410. The APU is composed of CPU and GPU cores. This thesis report discusses the following

- Design and verification of Bandwidth Bandit
- Results from applying the bandwidth bandit method on the APU

2. Background Information

2.1 Latency

Latency [5] is defined as the time elapsed between stimulation and response. Memory access latency is defined as the time elapsed between the issue and completion of request to the memory.

2.2 Memory level parallelism

Memory level parallelism (MLP) [11] at any instant \( t \) is defined as the number of useful off-chip memory accesses outstanding at \( t \). Figure 3 shows the timeline of execution of compute and memory instructions. The latency of memory access is 200 cycles. From the figure 3, MLP can be defined as follows

\[
MLP(t) = \begin{cases} 
1 & ; \quad 80 < t < 90 \\
2 & ; \quad 90 < t < 280 \\
1 & ; \quad 280 < t < 290 \\
1 & ; \quad 370 < t < 570
\end{cases}
\]

Figure 3 - The timeline shows progress of compute and off chip memory access instructions. Off chip memory access latency is 200 cycles. The green line named Compute, shows the progress of compute instructions. The red line named Memory, shows progress of memory instructions. The lines named M1, M2 and M3 shows the progress of three different off-chip memory accesses.

2.3 Memory bandwidth

Memory bandwidth observed by a thread can be calculated as the total number of bytes accessed by the thread per unit time. In this thesis, the bandwidth that is of interest is the bandwidth received by a thread from the main memory and not the theoretical bandwidth provided by the memory device. Memory Bandwidth received by a thread can be expressed as a function [3] of MLP and latency as follows
bandwidth = \text{transfer\_size} \times \text{MLP/latency}

The definition above clearly shows that the bandwidth received by a thread is directly proportional to the MLP of the thread. The bandit is designed to execute with a specified MLP value.

The memory requests from the different threads are kept in a queue before they are served by the memory device. This is because the speed of the memory device is less than the speed of the processor by an order of magnitude. The memory access latency observed by the thread includes the time spend waiting in the queues and the latency of the memory device. A memory request will typically encounter multiple queues before it is taken by the memory device. A description about the path followed by a memory request is given later in the report.

2.4 Memory hierarchy

The memory system used in a computer is not a single flat memory device. It is a hierarchy of devices with different capacity, access latency and bandwidth. A typical hierarchy consists of CPU registers, cache memories, main memory, local secondary storage (hard disk), and remote secondary storage (tapes, distributed file system etc.). It is shown in figure 4.

At the top of this hierarchy is the processor registers, followed by the cache memories and remote secondary storage devices are at the bottom of the hierarchy. Although the devices at higher levels are faster and smaller, they cost more per byte. Therefore in a typical memory system, the devices at higher levels have lower capacity than the devices in lower levels.

Figure 4 - Access latency of device increases from the top to the bottom of hierarchy. The price per byte decreases from the top to the bottom. Therefore, in a typical computing system, the capacity of the devices in a memory hierarchy increases from top to bottom.

In order to get an optimal performance from the memory hierarchy, the data accessed by a program should be stored at higher levels. However, due to the lower capacities, only the most frequently accessed data is stored at the higher levels of the hierarchy. When the processor issues a memory access request, it is received by the cache memory and if the request cannot be served by the cache, it is received by the main memory. This thesis describes the design and implementation of a program that tries to steal the main memory
bandwidth from co-running programs. In this context, it is sufficient to know about cache and main memory which will be described later. The other levels of memory hierarchy will not be covered.

2.5 Cache Memory

Cache memory is the second level in the memory hierarchy as shown in the figure 4. The main memory is supposed to store all the data required by a program. There is a big gap between the speed of the processor and the main memory. In order to reduce the average memory access time, cache memories are used. Cache is a small and expensive memory which is one or two orders of magnitude faster than main memory. As of today, the maximum available capacity of a cache is few megabytes whereas the main memory is usually few gigabytes.

The main memory stores data in cells which are identified using memory addresses. Since the capacity of the cache is much smaller than the main memory, it can only store a subset of the data stored in the main memory. As discussed in section 2.4, the memory access request from the processor first reaches the cache before the main memory. If the memory address in the memory access request is found in the cache, it is called a cache hit. If the memory address is not found in the cache, it is called a cache miss. If the request is for read, during a cache hit, the data from the cache is send to the processor. If a cache miss occurs, the data is fetched from the memory present on the next level in the hierarchy.

Most of the multi-cores and uniprocessors have multiple levels of cache memory which are referred to as level 1(L1), level 2(L2) and so on. The capacity of caches increases with increase in the level of the cache. The cache memory in the last level is also referred to as last level cache (LLC). Usually the lower level caches are private to the cores and the LLC is shared. When a miss happens at a certain level cache, the data is fetched from the memory at the higher level. So, in a system with two levels of cache, after a miss from L1, the L2 is searched for the data and after a miss from L2, the data is fetched from the main memory.

The cache does not store individual bytes. Data is stored as a group of bytes whose memory addresses are consecutive in main memory. The unit of storage in the cache is called a cache line which is usually 64B. The cache is organized as a collection of rows. The row is also called as a cache set and can contain one or more cache entries. Each cache entry consists of a tag and a cache line which is also known as the cache block. Direct mapped caches and set associative caches are two common schemes of cache organization. Since design and organization of a set associative cache is important to understand the design of the Bandit program, it is described in the next section.

2.5.1 Set associative cache

A cache is specified as N-way set associative cache if each cache set can hold N cache entries. A 4-way set associative cache is shown in figure 5. The design of a 4-way set associative cache for a 64 bit addressable main memory is described below
Cache dimensions:
Size = 64KB, Length of cache line = 64B, Associativity = 4

Design:
Number of cache entries in cache set = 4
Number of cache sets = Size / (Length of cache line * Associativity) = 256

If the address requested by the processor is of the form $b_{63}b_{62}...b_{1}b_{0}$

Bits $b_{5}b_{4}b_{3}b_{2}b_{1}b_{0}$ are used to address the byte in each cache line

Bits $b_{13}b_{12}...b_{7}b_{6}$ are used to index the cache row

Bits $b_{63}b_{62}...b_{15}b_{14}$ are used as the tag value

When the memory request is received by the cache, it loads the cache set whose index matches with $b_{13}b_{12}...b_{7}b_{6}$. Then the tag value of $b_{63}b_{62}...b_{15}b_{14}$ is matched with the tag value of each of the four cache entries of the selected cache set. If there is a match with the tag of a cache entry, then a cache hit has happened. The data from that cache entry can be read or written based on the type of memory access request being a read or write. If tag from none of the four cache entries matches with $b_{63}b_{62}...b_{15}b_{14}$, then a cache miss has happened. Then the data will be fetched from main memory and placed in the cache, thereby evicting one of the cache lines in the cache set. The data look up is shown in figure 5

Figure 5 – Data look up in a 4-way set associative cache. The memory address consists of index, tag and byte. The cache set is located using the index bits. The tag value is compared against the tag value of each cache entry in the cache set until a match happens. From the cache line, the required bytes are extracted using the byte part of the memory address
2.5.2 Cache Replacement

When a cache miss happens, the cache line containing the requested data is fetched from the main memory. If a program has accessed bytes from a certain cache line now, then there is a great chance that it may access the same bytes again or other bytes from the same cache line in the future. Therefore, the incoming cache line must be placed in an appropriate cache set such that it prevents cache misses in future. If the cache set is already filled with cache entries, one of them should be evicted to make space for the new one. A heuristic known as cache replacement policy is used to choose the entry that is least likely to be used in the future. The chosen cache entry is replaced with a new cache entry containing the new cache line. The aim of the cache replacement policy is to reduce the number of cache misses. A commonly used replacement policy is the least recently used (LRU), which discards the least recently used cache line first. In CPU caches with high associativity, a pseudo LRU scheme is implemented which tries to replace one of the many least recently used cache lines.

2.5.3 Types of cache misses

There are three types of cache miss namely compulsory miss, capacity miss and conflict miss. When the memory access request is to a cache line which has never been brought to the cache before, there will be a miss and that is called the compulsory miss. If the program’s working set is larger than the capacity of the cache, then some of the cache lines stored in the cache will be discarded to bring in the cache lines that belong to the working set of the program. The cache misses that are caused due to this condition are called compulsory misses. If the program’s working set has data that belongs to cache lines which belong to the same cache set and the number of such cache lines needed by the program is more than the associativity of the cache, then some of the cache lines in the cache set will be replaced with new cache lines. When the program accesses a cache line that was replaced earlier from the cache set, a miss happens which is called as the conflict miss.

2.6 Virtual Memory

Virtual Memory is a memory management technique used by computing systems to map the memory address values used by a program to the physical memory address values. Virtual memory makes the hierarchical memory appear as a single flat memory to the programs.

The memory address generated by a program is always the virtual address (VA). The operating system (OS) and the memory management unit (MMU) of the CPU participate in the translation of virtual address values into physical address (PA) values. The OS maintains the translation tables for every process in the main memory. The MMU uses those tables to compute the PA.

In most of the implementations of virtual memory, the virtual address space is divided into pages. Each page is a block of contiguous VA values used by the program. Usually, the size of a page is 4 kilobytes. The OS maintains a table called the Page Table (PT) which holds the mapping between the virtual and the physical pages. Each entry in the PT is known as the Page Table Entry (PTE). The VA generated by a thread is used to obtain the address of a PTE. If the physical page is present in the main memory, the PTE will contain the PA of the virtual page. The exact PA is constructed using the PA from the PT and the VA generated by the thread. The VA to PA translation is illustrated in figure 6 [10].

10
If the cache is designed such that the bits from PA is used as the index, and the bits from PA is used as the tag value, then the cache is known as physically indexed and physically tagged (PIPT) cache. The last level caches are usually PIPT caches and this thesis, the bandit is designed with this assumption. From the VA of a word, the bandit finds the PA and then use the PA to find the cache set to which the cache line containing the word.

2.7 Looking deeper into memory hierarchy

The memory hierarchy of a system considered for the design and development of the Bandit program is shown in figure 7 [3]. It consists of caches and the main memory. The system has multilevel caches namely L1, L2, L3 caches where L1, L2 caches are private to each core and L3 which is the LLC is shared among all the cores.

In a multicore processor, the cores operate in parallel and can produce as many memory requests per cycle as the number of the cores. The LLC caches are still slower than the processor. For example, the access latency of L3 cache in Intel Sandy Bridge [15] is approximately 27 CPU cycles. While the LLC is completing a memory access request, there
will be many more memory requests generated by a core waiting to be served by the LLC. A queue of finite capacity, local to each core is present in the system to hold each of these memory requests, until the LLC is ready to take them. These queues are labelled in the figure 7 as local.

As described in section 2.5, when a memory request misses in LLC (L3 in the case of figure 7), the request is send to the integrated memory controller (MC). The MC [14] manages the movement of data in and out of the Dual Inline Memory Modules (DIMM). The DIMM contain the Dynamic Random Access Memory (DRAM) circuits. The main (primary) memory of the system in figure 7 is composed of three memory modules.

The main memory is very much slower than the cores. For example, the access latency of main memory is approximately 180 CPU cycles [15]. During these 180 cycles, multiple cores can generate multiple memory requests which can miss all levels of cache. These memory requests will have to wait until the memory controller is ready to take them. A queue of finite capacity is present in the system to hold each of these memory requests. This single queue will contain the memory requests from all the cores and is labelled as global in the figure 7. In order to steal bandwidth, the bandit program tries to compete with the co-running programs for vacant slots in this global queue. This is explained in a later section.

In figure 7, it can be seen that the MC connects to the DRAM modules through independent memory channels. Each DRAM module [6] consists of several independent banks which could be accessed in parallel if there are no conflicts in address or data bus. Each bank is organized into rows (also called as pages) and columns. It is the banks that store the actual data in the intersection of the rows and columns. The MC translates the PA in the memory request it receives into DRAM address in terms of the channel, bank, row and column. As explained in the previous paragraph, the DRAM is slower than the CPU by hundreds of CPU cycles and therefore there are queues for each bank to hold the memory access instructions waiting to get access to the bank. These queues are inside the memory controller [14] and can be seen in figure 8. Caches are intentionally not shown in figure 8 as it is not important. During the execution of the bandit program, it will compete with the co-running programs for the vacant slots in those bank queues. This is described later in the report.
Figure 8 – Memory controller receives the stream of memory requests from the four cores. The address translator translates the physical address into a DRAM address in terms of the bank id. The memory request then waits in the queue of the corresponding bank until the bank is ready for access.

3. Design and Verification of Bandwidth Bandit

3.1 Bandwidth Bandit Method

In the introduction section of this report, the importance to measure the sensitivity of GPU applications to off-chip memory bandwidth contention generated by the co-running CPU applications in an APU was described. The thesis tries to verify the hypothesis that the bandwidth bandit method can be used to measure the sensitivity of a GPU application to the off-chip memory bandwidth contention due to a co-running CPU application.

Bandwidth Bandit method [3] is a quantitative method to measure the sensitivity of a target application to off-chip memory bandwidth contention. In this thesis, sensitivity of an application to bandwidth contention refers to the slowdown experienced by that application when subjected to a certain amount of bandwidth contention. The bandwidth contention created by an application is denoted by the amount of bandwidth received by the application. Bandwidth Bandit is a special program which when co-run with a target application in a multi-core, generates contention for off-chip memory bandwidth. For the sake of convenience, hereafter the Bandwidth Bandit program is referred to as the Bandit program.

The aim of bandwidth bandit method is to measure the slowdown $S$ experienced by a target program to off-chip memory bandwidth contention $C$. The method is described in the steps given below:

1. Run the target program and record its execution time
2. Start the bandit program to generate off-chip memory contention $C$
3. Co-run the bandit and target program
4. Record the execution time of target program
5. Stop the bandit program and measure the slowdown $S$ for target
6. Repeat the steps 1-4 with different values of contention $C$

In this thesis, the bandwidth bandit method is applied on an APU to measure the sensitivity of GPU kernel to contention from co-running CPU application. The experimental setup considered for measuring the sensitivity of the GPU kernel using bandwidth bandit method is shown in figure 9. The figure illustrates one of the possible configurations of the bandit program. The bandit program is executed on one of the four CPU cores along with the target GPU kernel on the GPU cores. The GPU host program launches the GPU kernel and waits for the kernel to complete the execution. The contention for off-chip memory bandwidth generated by the bandit program is named as $C$. The execution time of GPU kernel executing on the GPU cores is recorded. The slowdown $S$, experienced by GPU kernel is calculated for various values of contention $C$.

![Figure 9](image)

Figure 9 – Experimental set up to measure the sensitivity of a target GPU kernel to off-chip memory bandwidth generated by the bandit program running on the CPU cores

### 3.2 Bandwidth Bandit program

In order to apply the bandwidth bandit method on an APU, the first step is to design the bandwidth bandit. The bandwidth bandit program is the main component of the bandwidth bandit method. The bandwidth bandit is a program which generates contention for off-chip memory bandwidth. Contention for off-chip memory bandwidth is created by generating
demand for off-chip memory bandwidth. The bandwidth bandit method described in the previous section requires a bandit program to generate varying values of contention. Therefore the bandit program should be able to generate demand for off-chip memory bandwidth and there should be a mechanism to vary the demand.

Since the intention is to measure the sensitivity of the target program due to memory contention, the bandit program should not fight with the target program for the capacity of shared cache. This was a requirement for the bandit program as specified in the original paper [3], where the bandit was designed for multi-core CPU system. The last level cache is shared among CPU cores in a multi-core CPU but not between the CPU and GPU cores in an APU. Although the goal of this thesis is to use the bandit on an APU, Ericsson has plans to use it on multi-core CPU based systems in the future. Therefore the bandit [3] is designed not to use a significant amount of the shared cache memory.

From the above two paragraphs, the desired features of the bandit program can be formulated as follows

1. Bandit should generate demand for off-chip memory bandwidth
2. The user should be able to control the bandwidth demand generated by the bandit
3. The bandit should not use significant part of the last level cache

### 3.3 Requirements of Bandit program

The desired features of the bandit program are used to derive the requirements for the bandit program. The off-chip memory bandwidth demand generated by a program is same as the off-chip memory bandwidth consumed or received by a program. A thread consumes off-chip memory bandwidth when its memory access instructions are served by the off-chip memory. However, the memory hierarchy which consists of the caches and off-chip memory is transparent to the programs that are executed. A program cannot directly choose to bypass the cache memory to read or write from the off-chip memory. Therefore, as described in section 2.5, the bandit program should generate memory access instructions that misses from all the levels of cache and eventually served by the off-chip memory.

The bandwidth received by a program is directly proportional to the MLP of the program as described in section 2.3

\[
\text{Bandwidth} \propto \text{MLP}
\]

If the MLP of a program can be controlled, the bandwidth demand generated by the program can be controlled. Therefore, the bandit program should be designed in such a way that its MLP can be changed based on the user’s input at the start of the program execution.

Based on sections 2.2 and 2.5, the MLP of a program can be defined as the number of cache misses from the LLC. From the relation between bandwidth and MLP given above, it can be concluded that the bandwidth received by a program is directly proportional to the number of cache misses from last level cache.

\[
\text{Bandwidth} \propto \text{No: of LLC misses}
\]
Therefore, the bandit should generate cache misses based on the MLP value given by the user’s input.

As described in section 2.5.3, cache misses caused by a program can be classified into capacity misses and conflict misses. A capacity miss happens when the program’s working set is larger than the capacity of the cache. The bandit program cannot choose to create capacity misses because it will have to use the cache capacity completely thereby failing to satisfy the feature number three. A conflict miss can happen when the program’s working set has accesses to a number of cache lines mapped to a particular cache set. In order to cause one conflict miss only one cache set is used from the whole cache. If a program can cause \( m \) concurrent conflict misses by using \( m \) cache sets, it has an MLP of \( m \). A typical cache has several thousands of cache sets and the desired MLP value is at most 25. Therefore, it can be concluded that if the bandit program generates conflict misses, it will not use up a significant amount of shared cache.

With this, the requirements for the design of bandit program can be formulated as follows

1. The bandit program should create conflict misses
2. The bandit program should create \( m \) conflict misses concurrently

### 3.3.1 A deep dive into bandwidth-MLP relation

This section describes the bandwidth-MLP relation from the point of view of the memory hierarchy described in section 2.6. The bandwidth-MLP relation described in section 3.3 gives an impression that bandwidth received/consumed by a program increases forever, with the increase in the MLP value generated by the program. However, this is not true as the bandwidth of the memory system is limited. There are limitations for the MLP values that can be achieved by a program.

The MLP of a program at any instant is the number of in-flight memory requests of the program and can be a different value at different points of the memory hierarchy. For example, with reference to figure 7, MLP of a program at the local level is the number of in-flight memory requests in the local queue corresponding to the core. Similarly, MLP of a program at global level is the number of in-flight memory requests of the program in the global queue (GQ) and the MLP at the MC level is the number of in-flight requests in the queues for the DRAM banks as shown in figure 8.

At any instant, there is a per core limit called as local limit, on the number of in-flight requests that can be placed in the GQ. Thus, there is a limit [3] on the MLP that a single thread can achieve and thereby a limit exists on the amount of bandwidth that can be consumed by a single thread. Therefore, more threads can be executed in parallel to achieve higher MLP and thereby higher bandwidth. However, there is a limit on the MLP that can be achieved by multiple threads due to the finite length of the global queue. This limit is called the global limit.

As described in section 2.6, the queues in the memory hierarchy have a finite number of slots to store the memory access instructions. Some queues in the hierarchy are shared among the co-running programs. There will be contention for the slots in such shared queues among the co-running programs. In the bandwidth bandit method, the bandit program generates contention for off-chip bandwidth by stealing the slots in the queues it shares with the co-
running programs. In a homogenous multi-core system, the GQ and queues for the DRAM banks in MC are shared among the co-running programs whereas in an APU, the CPU and GPU cores share the queues for the DRAM banks in MC.

The quantitative method to reason about the MLP of a program and the contention it generates for the slots in shared queues is given below. In this example the MLP at the global level is considered. If the bandit program with MLP value $m_b$ is co-run with a target program with MLP value $m_t$, and the length of GQ is $l_g$, then the contention for GQ slots by the bandit can be can be expressed as follows

\[ m_b + m_t \leq l_g \] - No contention for GQ slots

\[ m_b + m_t > l_g \text{ and } m_b > m_t \] - Bandit program creates contention for GQ slots

This understanding of the impact of the bandit on the memory hierarchy is the motivation behind the hypothesis of this thesis, which is to apply the bandwidth bandit method on an APU to measure the sensitivity of a GPU kernel to the contention generated by the co-running CPU program. Although the GQ is not shared among the CPU and GPU cores in an APU, the above method can be applied to reason about the contention for slots in other shared queues.

3.3.2 A deep dive into conflict misses

The bandit program should create conflict misses because the off-chip memory can be accessed without consuming a significant amount of the cache memory. This is illustrated in the following example.

Consider an 8MB, 16way associative cache with 64B long cache lines. These dimensions are close to the dimensions of cache memories found in a typical modern day multicore chip. In order to achieve an MLP = 24 using conflict misses, the bandit needs to steal 24 cache sets out of the 8K cache sets. The amount of cache used up by the bandit is less than 0.3%.

In order to create a conflict miss, the bandit program should access a stream of memory addresses which map to the same cache set, in a continuous loop. The number of memory addresses in the stream depends on the associativity of the cache. In order to ensure that the access to each memory address in the stream results in a cache misses, the number of memory addresses can be chosen as twice the associativity of the last level cache.

The idea in the previous paragraph is illustrated through an example here. Consider the memory system given in figure 10. The main memory addresses namely $\{(0000)_{2}, (1000)_{2}, (2000)_{2}, (3000)_{2}\}$ are among the many addresses mapped to the cache set with index value $\{(000)_{2}\}$. Let the sequence of cache lines containing the sequence of memory addresses be C0, C1, C2, and C3, all of which are mapped to the cache set with index $\{(000)_{2}\}$. 
Let the sequence of addresses in memory access instructions generated by the program be (0000)$_2$, (1000)$_2$, (2000)$_2$, (3000)$_2$, (0000)$_2$, (1000)$_2$, (2000)$_2$, (3000)$_2$ ... At any instant of time, the cache set can hold a maximum of two cache lines as the cache is two-way associative. In figure 1, the contents of the cache set are represented after the completion of each memory access instruction in the sequence. In the figure, the event of accessing the main memory address is represented using a green cell containing the address being accessed. In the beginning, the contents of the cache set is empty (marked in blue). Then the program accesses a list of main memory addresses (0000)$_2$, (1000)$_2$, (2000)$_2$, (3000)$_2$ in that order. The cells pointed by the arrow represent the contents of the cache set after the memory access is completed. Each of those memory accesses will result in a compulsory miss and the corresponding cache line is brought to the cache set. Consider that the cache follows a LRU replacement policy. Therefore, when the memory address (2000)$_2$ is accessed, the cache line C0 will be replaced by C2 and when the memory address (3000)$_2$ is accessed, C1 will be replaced by C3. After accessing (3000)$_2$, the program continues the sequence where the next main memory address to be accessed is (0000)$_2$. This time the cache set contains C2, C3 and memory access causes a cache miss. This is a conflict miss and for each further memory access a conflict miss occurs as shown in the figure 11 below. The memory access instructions which cause conflict misses are marked in red and the compulsory misses are marked in green.

Figure 10 – Main memory and two way associative cache. Memory addresses (0000)$_2$, (1000)$_2$, (2000)$_2$, (3000)$_2$ are mapped to cache set at index (000)$_2$
The set of array elements corresponding to the indices in conflict set are given below.

In order to run multiple bandit threads based on the user input, more contention can be generated. Hence the bandit program is designed to run multiple bandit threads based on the user input. The basic design of the bandit program is as follows:

1. The bandit program takes the desired MLP value \( m \) as input
2. The bandit program creates \( m \) conflict sets
3. The bandit traverses through the \( m \) conflict sets concurrently

3.4 Design of the Bandit program

The bandit program steals off-chip memory bandwidth from co-running programs by consuming bandwidth from the shared off-chip memory bandwidth. This is done by increasing the MLP value of the bandit program. However, as described in section 3.3, the amount of MLP that can be achieved by a thread is limited by the local limit. This poses a limit on the contention that can be introduced by a single thread. If more bandit threads are run in parallel [3], more contention can be generated. Hence the bandit program is designed to run multiple bandit threads based on the user input. The basic design of the bandit program is as follows:

1. The bandit program takes the desired MLP value \( m \) as input
2. The bandit program creates \( m \) conflict sets
3. The bandit traverses through the \( m \) conflict sets concurrently

3.4.1 Detailed design to construct a conflict set

In order to achieve an MLP = \( m \), the bandit needs to create \( m \) conflict sets. This is done by constructing one conflict set at first and then use the elements of the first conflict set to construct the remaining \( m-1 \) conflict sets. This section describes the methods used to construct the first conflict set. This is done during the initialization of the bandit program. Hereafter the bandit program will be referred to as the bandit for brevity. The main data structure used in the bandit is an array called \( data \). The steps used by the bandit to find a conflict set are given below.

1. Choose a random array element \( data[\text{rand}] \)
2. Find the virtual memory address of \( data[\text{rand}] \)
3. Find the physical memory address from the virtual address
4. Calculate the cache index of \( data[\text{rand}] \)
5. Traverse the array \( data \) to find other elements which has the same cache index
6. The list of array indices of all those elements are stored in \( indexList \)

The set of array elements corresponding to the indices in \( indexList \) forms a conflict set. The length of conflict set is chosen to be twice the associativity of the LLC in order to ensure that...
a conflict miss happen while accessing every single element of the set. The LLC is a physically indexed and physically tagged cache and therefore, in order to find the cache index of an array element, it is required to find its physical address. In order to translate a virtual memory address into the corresponding physical memory address, the bandit uses a set of interfaces called pagemap[8], which is provided by the Linux kernel. The value of cache index is calculated using the physical address and the specifications of cache such as size and associativity values.

Once the first conflict set is created, in order to create the other \( m-1 \) conflict sets, this method need not be repeated. Those conflict sets are constructed in a method described in the next section. In order to create the first conflict set, this thesis did not choose to use the method used by Eklov et al. [3] as it involved the creation of large pages. Although the above method causes more TLB misses and generates lesser off-chip memory traffic when compared to the large pages, this method increases the portability of the bandit.

### 3.4.2 Detailed design to create remaining \( m-1 \) conflict sets

The bandit specifies the allocation of the data array in such a way that the starting address of the array is aligned to the size of the virtual memory page used in the system. Thus, if the size of each array element is 8 bytes and the page size is 4KB, then \( \text{data}[0] \) to \( \text{data}[511] \) will span an entire physical page. If the length of a cache line is 64B then \( \text{data}[0] \) to \( \text{data}[7] \) will span an entire cache line. Thus, if \( \text{data}[0] \) belongs to a cache line whose cache index is 701, then \( \text{data}[8] \) will belong to the cache line with cache index 702, \( \text{data}[16] \) will belong to the cache line with cache index 703 and so on. Once the first conflict set is constructed using the method given in the section above, the remaining \( m-1 \) conflict sets are constructed using the method given in this section. If the first conflict set is \( \{ \text{data}[10], \text{data}[600], \text{data}[1090], \text{data}[1560] \} \) then the next conflict set is \( \{ \text{data}[18], \text{data}[608], \text{data}[1098], \text{data}[1568] \} \) and so on.

### 3.4.3 Detailed design to traverse through \( m \) conflict sets

This is the last step in the design of the bandit which describes how the demand for off-chip memory is generated by traversing the conflict sets. After the \( m \) conflict sets are constructed by the bandit, it creates a links between the elements of each conflict set in a circular manner. This is illustrated using an example in figure 12, by using the two conflict sets from the example in the previous section. The first conflict set is \( \{ \text{data}[10], \text{data}[600], \text{data}[1090], \text{data}[1560] \} \) and the second conflict set is \( \{ \text{data}[18], \text{data}[608], \text{data}[1098], \text{data}[1568] \} \).

A link between \( k^{th} \) element and the \( (k+1)^{th} \) element in the conflict set is created by storing the index of the \( (k+1)^{th} \) element as the data in the \( k^{th} \) element. In the figure 12, the link between \( \text{data}[10], \text{data}[600] \) is created by storing the index value 600 in \( \text{data}[10] \). The links are made circular by storing the index of the first element in the memory location of the last element. In the figure 12, the dotted arrows in red colour show the links between the elements of the first conflict set and the dotted arrows in blue colour show the links between the elements of the second conflict set. The last element in each list is linked to the first element in the same list thereby making the list circular in nature.
The bandit creates conflict misses by traversing the elements in a conflict set as explained in section 3.3. In order to achieve an MLP value of \( m \), the \( m \) conflict sets are traversed in a concurrent manner. This done by using the pointer chasing microbenchmark whose sample code is shown in listing 1. The sample code in listing one is based on the data array in figure 12. The \( \text{ix}[i] \) array is initialized with the first element of each conflict set. The sample conflict sets used in the previous paragraph can be used for explaining the working of the microbenchmark. The first conflict set is \{data[10], data[600], data[1090], data[1560] \} and the second conflict set is \{data[18], data[608], data[1098], data[1568] \}. The initial state of the array \( \text{ix}[i] \) will be \( \text{ix}[0] = \text{data}[10] \) and \( \text{ix}[1] = \text{data}[18] \). When the execution reaches the end of the for loop given in the listing 1, number of in-flight off-chip memory requests will be 2. The next iteration of the loop can only begin when \( \text{data}[10] \) is received and immediately the memory request for the next element in conflict set ie. \( \text{data}[600] \) will be dispatched. Therefore, while the loop is in execution, bandit program will always have an MLP value of 2. The value of \text{DESIRED\_MLP} can be set based on the user input and in this way the bandit’s MLP can be changed. The bandit program traverses the \text{DESIRED\_MLP} number of \text{conflict sets} concurrently to consume bandwidth. This method of creating conflict misses by chasing the pointers in a loop with data dependencies is taken from the original paper of Eklov et al. [3]. If the user specifies more than one bandit thread, then each thread creates a separate \text{data} array, constructs \( m \) conflict lists and traverses the list to generate off-chip memory bandwidth contention.
Figure 1 – The data array with the indices chosen for the conflict sets. The conflict set 1 is \{data[10], data[600], data[1090], data[1560]\} and conflict set 2 is \{data[18], data[608], data[1098], data[1568]\}. The red dotted arrows trace the links between the conflict set 1 and blue dotted arrows trace the links between the conflict set 2.

Listing 1 – Pointer chasing microbenchmark. The microbenchmark traverses a maximum of 2 conflict sets. This results in one memory access per conflict set. The accesses are made concurrent using a loop-carried data dependency. Using this microbenchmark, the MLP of program can be controlled. This example can be extended to desired value of MLP.

```c
#define CACHE_ASSOCIATIVITY 2
#define DESIRED_MLP 2

for(unsigned long long int i = 0; i < NUM_ITER;++i)
{
    switch(DESIRED_MLP)
    {
        case 2:
            ix[1] = data[ix[1]];
            break;
        case 1:
        {
            ix[0] = data[ix[0]];
            break;
        }
    }
}
```
3.4.4 Intrinsic MLP

With reference to the listing 1, from the perspective of the bandit program, its MLP is equal to DESIRED_MLP. However the MLP observed from GQ could be same or a different value depending on the local limit and the number of vacant slots in the GQ. Similarly the MLP of the bandit program from the queues in the DRAM banks could be another value. Hence, in order to distinguish among the MLP values at different levels, the MLP from the perspective of bandit program can be called as the intrinsic MLP. Intrinsic MLP is important because, it is by specifying the values of the intrinsic MLP and the number of bandit threads, the user can control the amount of off-chip memory bandwidth received by the bandit. Although the off-chip memory bandwidth cannot be precisely controlled, the control mechanism is sufficient for the purpose of measuring the sensitivity of a target application.

3.4.5 Measuring the bandwidth received

The user provides the values for the intrinsic MLP and the number of bandit threads as input at the start-up of the bandit program. Since the bandwidth consumed by the bandit program cannot be precisely controlled by those values, it will be useful to know the bandwidth received by the bandit program while it is running. The bandwidth value can be used as a feedback to modify the input parameters until the bandit consumes the desired amount of off-chip memory bandwidth. Therefore, the bandit is designed to report the bandwidth received in periodic intervals. Here the bandwidth refers to the bandwidth observed by the thread and not the theoretical bandwidth of the memory system. The bandwidth received by the bandit program is measured by the program itself. In order to reduce the dependency on the platform, the bandit program does not use hardware performance counters to measure the bandwidth. The bandit program records the time elapsed between the issuing a memory request and receiving the data. Since the bandit knows its intrinsic MLP, it can calculate the bandwidth received as follows

\[
\text{Bandwidth} = \frac{\text{intrinsic MLP} \times \text{SIZE_OF_CACHE_LINE}}{\text{Time elapsed}}
\]

3.5 Verification of the Bandit program

This section describes the verification of the features of the bandit program which are given in section 3.2. The test cases were executed on a quad core AMD A8-6410 (family 16h) APU machine. The APU chip which is codenamed as AMD Beema has the following cache configuration

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>32KB/32KB, 2/8 way, instruction/data, private</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2 MB, 16 way, unified, shared</td>
</tr>
</tbody>
</table>

The MC has two channels of which only one channel is connected to a memory device. The memory device is a dual ranked 8GB, 1.35V, 11-11-11, DDR3L-1600 CL11 SDRAM. In this machine, when there is a miss in the L1-D cache, the memory request is send to the shared L2 cache. Each core has a load –store (LS) unit which contains a queue that can track up to eight
in-flight L1 cache misses. The memory request corresponding to the load instruction leaves the queue when the load has been completed and the data is delivered [16].

The features of the bandit program given in section 3.2 are verified using test cases and the results are presented in plots which show the bandwidth consumed vs intrinsic MLP of the bandit or the target program. The bandit program takes the number of bandit threads and the intrinsic MLP value as inputs.

3.5.1 Bandit can generate varying amounts of off-chip memory bandwidth

The aim of this test case is to verify that the bandit can generate demand for off-chip memory bandwidth and the user can control it in a convenient manner. In this test case, the intrinsic MLP of the bandit is varied and the bandwidth generated by the bandit program is recorded. As mentioned before, the intrinsic MLP is the user input at the start-up of the bandit program and the bandit program can measure the bandwidth it receives. This experiment is similar to the experiments done by Eklov et al. [3]. The result of the experiment is shown in figure 13.

![Bandwidth vs intrinsic MLP](image)

**Figure 13 – The bandwidth received by the bandit vs intrinsic MLP of bandit program**

The observations from the result shown in figure 13 are

- The bandit can generate a demand for off-chip memory bandwidth
- When intrinsic MLP ≤ 8, the bandwidth consumed is directly proportional to the intrinsic MLP. This is because the queue in LS unit can track up to eight memory requests.
- When intrinsic MLP > 8 the bandwidth does not increase in proportion to MLP. This is because the access latency of the memory requests now includes the time spending waiting to get access to the queue in LS unit.
The data point for MLP = 9 appears strange because there is no increase in the bandwidth when MLP is increased from eight to nine. This is the only case where the bandwidth has not increased when the MLP is increased. Something strange is happening with the memory system when the MLP = 8.

- When MLP > 11 the percentage of increase of bandwidth is very small (< = 4%)

This test verifies the following

- The bandit can generate demand for off-chip memory bandwidth
- The user can control the bandwidth consumed by the bandit by specifying the intrinsic MLP. Although this is not a high precision control, it can be used to vary the bandwidth consumed by the bandit program

3.5.2 Bandit can steal off-chip memory bandwidth from a target program

The aim of this test case is to verify that the bandit that was designed during this thesis is suitable to be used as the bandit program in the bandwidth bandit method. The role of bandit program in the bandwidth bandit method is described in section 3.1. In order to verify that bandit developed during the thesis is suitable for the bandwidth bandit method, it is essential to verify that the bandit program can steal bandwidth from co-running applications. This is done by applying the bandwidth bandit method using the bandit and a target whose bandwidth consumption characteristics are known earlier. When the target is co-run with the bandit, the target should receive lesser bandwidth than what it received when it was run without the bandit. This test case also aims to verify that the bandit is effective in stealing bandwidth from co-running target applications of varying MLP values.

Since the bandwidth vs intrinsic MLP characteristics of the bandit is already known from section 3.5.1, it can be used as the target application. The test case involves co-running the bandit with a fixed intrinsic MLP = 24 and another instance of bandit as the target. During the test the intrinsic MLP of the bandit used as target was varied from 1-24. The target measures the bandwidth it receives for every value of intrinsic MLP. Here both the target and bandit are single threaded programs. The results of the test case are shown in the figure 14 below.
In figure 14, the blue curve labelled as Tgt_B0 represents the bandwidth received by the target when it is executed without the bandit program and the red curve labelled as Tgt_B1 represents the bandwidth received by the target when it is co-run with the bandit program. During the test, the intrinsic MLP of the bandit program was fixed and the intrinsic MLP of the target was varied. It can be observed that red curve, labelled as Tgt_B1, lies significantly lower than the blue curve, labelled as Tgt_B0. This means that when the target program is co-run with the bandit, it receives significantly lower bandwidth than what the target received when it was executed without the bandit program. This is because the bandit program stole the bandwidth from the target. Therefore, from the figure 14, it can be concluded that the bandit program can be used to steal bandwidth from a co-running target program and it is suitable for the bandwidth bandit method. From figure 14, it can be observed that when co-run with the bandit program, the target has received lower bandwidth for every value of MLP. This means that the bandit program has been able to steal bandwidth from the target programs with a wide range of MLP. It can also be observed that when the target program generates higher bandwidth demand (MLP > 12), the amount of bandwidth stolen by the bandit is also higher. This is in accordance with previous studies [1], [2], where it was observed that programs with higher baseline bandwidth demand are generally more sensitive to off-chip memory bandwidth contention than programs with lower baseline bandwidth demand.

3.5.3 More bandit threads can steal more bandwidth from the target program

As described in section 3.3.1, there is a limit to the amount of bandwidth that can be stolen by a single thread. This is observed in the figures 13 and 14 where the bandwidth levels off when MLP > 8. As described in section 3.4, more bandit threads should be able to steal more bandwidth from the target. Therefore, it is essential to verify that the bandit designed in this
thesis can steal more bandwidth with more threads. The aim of this test case is to verify that
the bandit program with two threads can steal more bandwidth from the target.

The bandit program is designed in such way that the user can specify the number of threads
at the start-up of the program. During the test, each thread is pinned to a single core and the
MLP of the target is varied from 1, 2.. 24. The bandwidth received by the target is recorded
when it is co-run with bandit program. The result is shown in figure 15.

![Target bandwidth vs intrinsic MLP](image)

Figure 15 – The curve labelled as Tgt_B0 represents the bandwidth received by the target program without the
bandit. The curves labelled as Tgt_B1 and Tgt_B2 represent the bandwidth received by the target when co-run
with one and two bandits respectively.

In the figure 15, the blue curve labelled as Tgt_B0 represents the bandwidth received by the
target program when it is not co-executed with the bandit program. The red curve labelled as
Tgt_B1 and green curve Tgt_B2 represent the bandwidth received by the target program
when co-executed with one and two bandits respectively. It can be observed that the curves
Tgt_B1 and Tgt_B2 lie below the curve Tgt_B0 and that the Tgt_B2 lies below Tgt_B1. This
means when the target program is co-executed with two bandit threads, it receives lesser
bandwidth than the bandwidth it received when co-executed with one bandit thread. This is
because the bandit program with two threads steals more bandwidth from the co-running
target than the bandit program with one thread. Therefore it is verified that the bandit
program developed during this thesis can steal more bandwidth using more threads.

4. Results

As described in the introduction section of the report, the goal of this thesis is to apply the
bandwidth bandit method to measure the slowdown of a target GPU kernel to the off-chip
memory contention from co-running CPU applications on a heterogeneous multicore
This section presents the results after applying the bandwidth bandit method to measure the sensitivity of a target GPU kernel on a heterogeneous multi-core processor.

4.1 Target heterogeneous multi-core

The bandwidth bandit program developed during the thesis was evaluated on a real hardware and not a simulator. The target heterogeneous multi-core processor used for the experiments is the AMD APU A8-6410. This hardware was chosen because, it contains CPU and GPU cores, which share the off-chip memory bandwidth. The programs which execute on the CPU cores can create contention for the off-chip memory bandwidth and the GPU programs can be affected due to the contention. Besides, Ericsson had an interest in evaluating the AMD APUs.

The APU consists of four x86 based CPU cores and an on-chip AMD Radeon R5 GPU. The R5 GPU contains two compute units based on the AMD Graphics Core Next (GCN) [17] architecture. The internal structure of the target platform is shown in figure 16.

![Diagram of APU](image)

Figure 16 – The APU consists of CPU cores and GPU compute units. The Northbridge provides the interface between the CPU, GPU cores to the main memory. The memory requests from GPU compute units pass through the Graphics Memory Controller and the DRAM Controller. The memory requests from CPU cores pass through the DRAM controller. The DRAM controller is the interface to the DRAM.
4.2 Measuring the slowdown of scalar multiplication of vector kernel

The aim of this experiment is to apply the bandwidth bandit method to measure the slowdown of a target GPU kernel to the off-chip memory contention from co-running CPU applications on the target platform. The target GPU kernel performs the multiplication of a vector by a scalar constant. It can be mathematically expressed as \( B = k \times A \) where \( A \) and \( B \) are vectors and \( k \) is a scalar constant. The target application consists of a host program written in C++ and a kernel program that is written in OpenCL C. The host program is executed on the CPU and during its execution it launches the kernel program onto the on-chip GPU compute units. The host program can specify the number of GPU threads to execute the kernel program. The execution on the GPU compute units follows the Single Program Multiple Data (SPMD) model. Each GPU thread executes the same kernel program but operate on different data. The input vector \( A \) is partitioned equally among all the GPU threads. Each thread performs the scalar multiplication on its data partition. All threads can perform their work in parallel because there are no data dependencies among the threads.

Consider a vector of size 16 and GPU threads as eight, then the operation of the kernel program can be illustrated as shown in figure 17. The vector \( A = [A_0, A_1 ... A_{15}] \) and the scalar constant is \( k \). The number of GPU threads specified for the operation is eight. The vector \( A \) is partitioned among the eight threads equally where each thread performs the scalar multiplication. The GPU threads are denoted as T0, T1 ... T7 where T0 performs \( k \times A_0 \), T1 performs \( k \times A_1 \), T2 performs \( k \times A_2 \), and so on.

\[
k \times \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \\ A_8 \\ A_9 \\ A_{10} \\ A_{11} \\ A_{12} \\ A_{13} \\ A_{14} \\ A_{15} \end{bmatrix} = \begin{bmatrix} k \times A_0 \\ k \times A_1 \\ k \times A_2 \\ k \times A_3 \\ k \times A_4 \\ k \times A_5 \\ k \times A_6 \\ k \times A_7 \\ k \times A_8 \\ k \times A_9 \\ k \times A_{10} \\ k \times A_{11} \\ k \times A_{12} \\ k \times A_{13} \\ k \times A_{14} \\ k \times A_{15} \end{bmatrix}\]

\{ T0 \}
\{ T1 \}
\{ T2 \}
\{ T3 \}
\{ T4 \}
\{ T5 \}
\{ T6 \}
\{ T7 \}

Figure 17 – The figure illustrates the operation of the target kernel program. \([A_0, A_1 ... A_{15}]\) is a vector and \( k \) is a scalar constant. T0, T1 ... T7 denote the GPU threads. The vector is divided into eight equal sized partitions. Each thread concurrently performs the scalar multiplication on its vector partition.

The execution of every GPU thread is sensitive to latency because there is no computation to overlap with the off-chip memory operations. As soon as the off-chip memory requests from
the GPU thread are dispatched, the thread has nothing to do and it stalls. Although the GPU tries to hide the memory access latency of the dispatched memory requests, by scheduling a different thread which is ready to execute, soon the new thread will also stall. Therefore, this GPU kernel is highly sensitive to memory access latency. When subjected to off-chip memory contention the target GPU kernel is expected to experience substantial slowdown.

4.2.1 Applying bandwidth bandit method

The aim of this experiment is to measure the sensitivity of the target kernel to the off-chip memory contention caused by the co-running bandwidth bandit. This is done by measuring the execution time of the target kernel when co-executed with a bandit that generates a certain bandwidth demand. The bandwidth received by the bandwidth bandit is a measure of the off-chip memory contention it generates.

The experiment is conducted by following the procedure of bandwidth bandit method as given in section 3.1. The bandit is executed on the CPU cores and the target kernel is executed on the GPU cores of the APU. The bandit and the target program measures the bandwidth received by each of them. The bandit measures the bandwidth it receives as described in section 3.4.5 and the target host program uses OpenCL profiling events to measure the execution time of the kernel program on the GPU.

The number of GPU threads is chosen in such a way that the GPU kernel does not saturate the off-chip memory bandwidth of the system. This means that the aggregate bandwidth demand of the bandit program and the target kernel is less than the memory bandwidth capacity of the memory system. The target kernel is expected to experience substantial slowdown due to off-chip memory contention from the bandit even when the aggregate bandwidth demand is less than the bandwidth capacity of the memory system. This is because the target kernel is maximally sensitive to the memory access latency.

The amount of off-chip memory bandwidth consumed by the bandwidth bandit is varied by varying the MLP and the number of threads. The bandwidth measurements are made after pinning the bandwidth bandit and the target host program to specific CPU cores. The results are shown below in figure 18.
As expected when the target GPU kernel was subjected to a range of off-chip memory contention (0.47 – 4.82 GB/s), it experienced substantial slowdown (8.4% - 109.49%). It can be seen that when the bandwidth received by the bandit increased, the slowdown experienced by the target kernel mostly increased. This is because, when the bandit consumed more bandwidth, it increased the contention for the shared resources. When the contention for shared resources increases, the memory access latency for the target GPU kernel increases. This causes the threads of the target GPU kernel to wait for a longer time to get the response for its off-chip memory requests. This leads to increasing slowdown values when subjected to increasing amounts of contention.

However, the rate of increase in slowdown of the target kernel is lesser than the rate of increase in the contention. This is because the GPU always tries to hide the memory access latency by using a large number of threads. In particular, during this experiment, the target GPU kernel used 32000 threads and thus the increase in access latency was ameliorated.

The blue bars show the slowdown experienced by the target when co-executed with a single threaded bandit. The bandit’s bandwidth was increased by increasing the MLP of the program. The bandwidth received by the bandit is a measure of the contention generated by the bandit. It can be seen that the slowdown experienced by the target increased with increase in contention. However, there was a limit on the contention that could be generated by a single threaded bandit. When this limit was reached, sensitivity measurement was continued by increasing the number of bandit threads to two.

The red bars show the slowdown experienced by the target when co-executed with a bandit program that had two threads. The bandit’s bandwidth was increased by increasing the MLP of each thread. It can be seen that the slowdown values in this region is much higher than the
slowdown values in the blue region. For the same value of MLP, the bandit program generates much larger contention with two threads than with one thread. Since the target GPU kernel is maximally sensitive to latency, it experienced greater slowdown values when subjected to more contention.

From figure 18, it can be seen that when the bandwidth received by the bandit has increased from 3.27 GB/s to 4.82 GB/s, the slowdown increased from 99.6% to 109.49%. That means when the bandit’s bandwidth increased by 47.4%, the slowdown increased by only 0.1%. Therefore, it can be concluded that in this region, bandit can consume more from shared bandwidth without reducing the bandwidth available to target. This could be because of the partitioning [21] of the main memory capacity of the APU between the CPU and GPU programs. The data accessed by the bandit program and the GPU program are in different partitions. This means that there is a limit to the MLP that can be stolen from target GPU program.

4.3 Comparing the slowdown of Black Scholes and Scan kernels

In the previous experiment, the bandwidth bandit method was applied on a target GPU kernel which had low performance and low bandwidth demand. In this experiment, the bandwidth bandit method is applied on those target kernels with higher performance and bandwidth demand which are taken from the AMD APP SDK v3.0 [18]. It is more relevant to measure the slowdown of these kernels with the bandit developed during the thesis because they try to solve more standard data parallel problems and are optimized for high performance execution on the GPUs.

4.3.1 Target applications

The aim of this experiment is to compare the slowdown experienced by two different GPU applications when subjected to the same amount of off-chip memory contention caused by co-running CPU applications. In order to estimate the effects of contention on the target GPU programs, it is essential to know their relevant characteristics. The GPU applications chosen for this experiment and their characteristics are given in the table 1 below.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Application</th>
<th>MPKI</th>
<th>BW/C (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLK</td>
<td>Black Scholes</td>
<td>1.6</td>
<td>67.6</td>
</tr>
<tr>
<td>SLA</td>
<td>Scan of Large Arrays</td>
<td>2.7</td>
<td>45.0</td>
</tr>
</tbody>
</table>

Table 1 [19], [20] – Application characteristics. MPKI – L2 cache misses per kilo-instructions and BW/C – The ratio of attained bandwidth to peak system bandwidth

The performance of a GPU application is defined as the number of operations per second. It can be estimated based on the analytical model given in this paper [19]. In this paper the performance $P$ of a GPU application is expressed as a function of its L2 cache misses per kilo-instructions ($MPKI$) and its bandwidth ($BW$).

$$P \propto \frac{BW}{MPKI}$$
The paper says that BLK and SLA were tested with this model [19] and the mean absolute error was only 4.2%. Therefore, this model can be used to compare the performance of BLK and SLA.

On applying the model to calculate the performance of BLK and SLA using the values from table 1,

\[ P_{BLK} = k \times 42.25 \quad \text{and} \quad P_{SLA} = k \times 16.67 \]

where \( k \) is the proportionality constant.

From the performance values it is evident that BLK performs more number of operations per second than the SLA. However, BLK also has a higher bandwidth demand than SLA and this should make BLK more sensitive to off-chip memory bandwidth contention.

4.3.2 Applying Bandwidth Bandit method

The bandwidth bandit method is applied separately on BLK and SLA as described in section 4.2.1. The inputs to BLK and SLA are configured such that they deliver the best performance. The slowdown experienced by the target is recorded for different values of bandit bandwidth. The experiment is repeated with two bandit threads. During this experiment, it was observed that the slowdown of BLK and SLA increased with increase in the MLP of the bandwidth bandit program. In order to compare the sensitivity to contention, the average slowdown was calculated and plotted. The results are as shown in figure 19.

![Figure 19](image_url)

Figure 19 – The Y-axis shows the percentage average slowdown experienced by the target GPU program. The blue and red bar represent the Black Scholes and Scan kernel respectively. The groups namely Bandit_1 and Bandit_2 represent the cases when the co-running bandit program had one and two threads respectively.
From the figure 19, it can be seen that BLK experiences higher slowdown than SLA. Although BLK is a higher performance kernel than the SLA, the contention from the CPU application brings larger slowdown to BLK. The reason for the larger slowdown of BLK is due to its higher bandwidth demand than SLA. The higher bandwidth demand makes the execution time more sensitive to off-chip memory bandwidth contention caused by the co-running CPU application. This result is very similar to the results from the previous studies [1], [2] on the sensitivity of CPU applications from co-running CPU applications. Their main results were that the CPU applications with higher baseline bandwidth demand are generally more sensitive to off-chip memory bandwidth contention.

5. Conclusion

In a heterogeneous multicore processor, the CPU and GPU cores share the off-chip memory bandwidth. When a GPU program is co-executed with a CPU program, there could be contention for the shared memory bandwidth. The performance of a target GPU program can be affected due to the contention generated by the co-executing CPU program. In order to improve the performance of the GPU program when subjected to contention, it is important to understand the sensitivity of the target GPU program to memory contention.

Eklov et al. [3] have already shown that in a homogeneous multicore processor, the bandwidth bandit method can be used to measure the sensitivity of a CPU program to memory contention caused by co-running CPU programs. In this thesis, a bandwidth bandit program was designed which can execute on the CPU cores and create contention for the memory bandwidth. This bandwidth bandit program was used to study the effects of contention on target GPU programs that had different values of bandwidth demand. This was done by applying the bandwidth bandit method on those target programs on a real heterogeneous multicore processor. The results after applying bandwidth bandit method showed that the target GPU kernels suffered substantial slowdown. The conclusion from this thesis is that, in a heterogeneous multicore processor, the bandwidth bandit method can be used to measure the sensitivity of a target GPU program to contention caused by co-running CPU programs.
Bibliography


[14] B.Jacob, S.W.Ng, D.Wang, " Memory Systems: Cache, DRAM, Disk", Elsevier


