2015 IEEE 22nd International Conference on High Performance Computing

Understanding the Performance Benefit of Asynchronous Data Transfers in OpenCL Programs Executing on Media Processors

Nagendra Gulur
Texas Instruments (India)
Bangalore, India
Email: nagendra@ti.com

Suriya Narayanan L
Texas Instruments (India)
Bangalore, India
Email: s-narayananl@ti.com

***Abstract*—In this work, we study the performance benefits of using asynchronous data transfers in OpenCL programs executing on media processors. Asynchronous data transfers are typically implemented by use of *Direct Memory Access* (*DMA*) engines that can be programmed to transfer data from one memory location to another. Asynchronous transfers can free up processing cores from managing data transfers and having to wait for transfer completion. In a typical programming model using asynchronous transfers, the kernel uses a double-buffering scheme wherein data is moved to/from one buffer ("scratch-pad") while the core operates on the other buffer. Intuitively, this model allows the cost of data transfers to be hidden or overlapped with computation.**

This is in contrast with accessing data "through the cache". Here, the core executes loads and stores to access the required data. Due to the inherent spatial and temporal locality of accesses, the cache hierarchy plays a significant role in mitigating the cost/delay associated with frequent off-chip accesses.

In this work, we seek to understand the performance gains expected with use of asynchronous transfers in a typical media processor. To do so, we first develop a simple yet insightful model of performance that helps quantify the benefits of asynchronous transfers over cache-based accesses in such processors. Next, we experimentally evaluate these programming styles on a variety of kernels executing on the Texas Instruments Keystone-II multi-core DSP platform. We observe that asynchronous data transfers can improve performance of image-processing OpenCL programs by as much as $5\times$, with an average improvement of 40%.

***Keywords*-OpenCL; Multi-core; DDR; DMA; Image Processing;**

I. Introduction

OpenCL [1] has emerged as a de-facto standard for writing portable high performance *kernels* that can be compiled and run on a variety of hardware platforms, including CPU cores (such as x86 [2] & AMD [3]), GPU cores (from vendors such as NVIDIA [4], ARM [5], & Imagination [6]), and DSP cores (from vendors such as Texas Instruments [7]). These kernels, written in the C99 [8] language, are invoked from a host application by a well-defined set of APIs defined by the OpenCL standard specification. Typically, these kernels are compute-intensive tasks that operate on large chunks of data with fairly regular processing and communication patterns.

Image processing algorithms fit this programming model and a variety of different image processing tasks have been implemented and accelerated using this standard [9]. Most low-level image processing functions involve transforming the input image data in various ways, such as filtering, scaling, rotating, computing differences between a pair of images, and so on. These kernels typically stride through the input images in a regular access pattern. Typically, due to the spatial locality of accesses, hardware caches achieve high cache hit rates.

In this work, we consider media processors. Media processors typically comprise one or more specialized cores with video/vision/imaging-oriented instruction-sets, and efficient data streaming interfaces. While these cores support complex functional units, they generally avoid sophisticated features of general-purpose cores such as out-of-order execution, branch prediction and multiple hardware thread contexts. The memory hierarchy may comprise one or more levels of caches or scratch-pads backed by an off-chip main memory. They are generally provisioned with sophisticated *DMA* engines to ensure efficient streaming of data between the off-chip memory and internal scratch-pads. In such systems, cache based accesses (abbreviated *CB* in the rest of the paper) suffer from two limitations:

- Cache misses result in severe performance penalty due to off-chip main memory accesses.
- In multi-core configurations, cache misses suffer significant inter-core interference at shared resources such as the main memory and the shared on-chip data transfer network.

A typical off-chip memory access can take hundreds of processor cycles (see Section IV for a detailed description of our evaluation methodology). In the absence of hardware support to hide this penalty, much of this time is spent stalling the requesting core. While a single core's requests may have spatial locality in the off-chip memory, when multiple cores contend for main memory, this spatial locality is often destroyed causing large inefficiencies in off-chip

978-1-4673-8488-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HiPC.2015.14

memory accesses[1]. Thus we seek to evaluate the performance benefits of the alternative to cache-based accesses: the use of *DMA* engines to move data to/from main memory thus freeing up the processor. These *asynchronous transfer* versions (abbreviated *AT* in the rest of the paper) can potentially hide expensive off-chip memory access penalties by allowing useful computations to proceed in parallel with background memory transfers. In such an implementation, there are one or more *local scratch-pad* memories that are typical small but fast SRAM memories that the processor can access far more quickly than off-chip DRAM. The *DMA* engines are programmed to transfer data from off-chip DRAM into these scratch-pad memories.

The effectiveness of this overlap of transfers with computation depends on the severity of the memory access penalty in the *CB* version and how well this penalty can be hidden in the *AT* version. In this work, we examine this question by thoroughly evaluating the benefits of the *AT* versions across several directed micro-benchmarks as well as image processing kernels.

A. Our Contributions

In this paper, we make the following contributions:

- We develop and validate an analytical model of performance that estimates the benefit of asynchronous data transfers in streaming workloads executing on media processors. This model provides a framework for understanding performance issues in *CB* and *AT* versions as well as for identifying the impact of various parameters (workload as well as underlying hardware) and their interactions on performance. The model is demonstrated to achieve high accuracy (average error 6%) in estimating the speed-up of the *AT* versions over *CB* versions of kernels.
- Using the model, we derive an expression for the optimal *DMA* transfer size.
- Using the Texas Instruments KeyStone II multi-DSP SOC platform [10], we evaluate the performance improvement due to asynchronous data transfers using both directed micro-benchmarks as well as a variety of image processing kernels. We find that asynchronous data transfers can improve performance by as much as $5\times$ with an average improvement of 40% over cache-based kernels.

II. Motivation

In this section, we discuss the motivation for evaluating asynchronous transfers. For illustration, we use the straightforward vector-summation kernel whose code snippet is shown in Figure 1. The kernel sums up the elements of a vector and as such this cache-based implementation enjoys a high cache hit rate (assuming 64B cache blocks in the L1D, and 4B elements, its L1D hit rate is 93.75%). The execution timeline of such a cache-based kernel is depicted by the top portion of Figure 3. The kernel alternates between short bursts of computation (denoted 'C') and memory accesses (denoted 'M') with a total execution time that is a sum of the total computation carried out plus the total memory access time. High cache miss rates and/or high off-chip memory access times can cause the kernel to consume a large number of processor cycles.

```
#define GLOBAL_DATA_SIZE  262144
__kernel __attribute__ ((reqd_work_group_size(1,1,1)))
void fvec_sum(__global float *A, __global float *B)
{
    float sum=0;
    for (i=0;i<GLOBAL_DATA_SIZE;i++)
    {
      sum+=A[i];
    }
    B[0]=sum;
}
```

Figure 1: Vector Summation Kernel - Cache-based Implementation

Asynchronous transfers can potentially hide these exposed memory access penalties. The *async_work_group_copy* [11] OpenCL API initiates asynchronous data transfer between local and global memories by programming an on-chip DMA engine and returns control back to kernel execution. The function is executed commonly across a *work-group* and is expected to be used to transfer data for all the *work-items* in the *work-group*. The kernel can continue to perform useful computation and subsequently synchronize with the data transfer via the *wait_group_events* [12] API. If the durations of computation and data transfer match, then nearly all of the data transfer overhead can be hidden behind computation and thereby the overall kernel execution time can be reduced. This is shown in the bottom portion of Figure 3. The blocks marked with an 'S' denote the time spent in transfer control (DMA setup and completion). These blocks constitute overheads of asynchronous implementations.

Since computation proceeds concurrently with data transfers, it necessitates use of multiple buffers – one set of buffers for computation to access, and a second set of buffers that are used for data transfers - *double-buffering* scheme [13]. As shown in Figure 3, the asynchronous version has to be "primed" for computation by fetching an initial chunk of data from global memory into a local scratch-pad. Once the initial chunk has been fetched, the transfer of the next chunk is initiated followed by the computation on the available chunk. Once the computation is completed, it waits for the next chunk to become available before continuing. There is a similar "epilog" at the end.

[1]This is due to the interleaving of requests from multiple cores resulting in adversely affecting the row-buffer locality in memory.

```
#define CHUNK 4096
#define GLOBAL_DATA_SIZE 26214
#define NUM_CHUNKS  GLOBAL_DATA_SIZE/CHUNK – 1
__kernel __attribute__ ((reqd_work_group_size(1,1,1)))
void fvec_sum_async(__global float *A, __global float *B)
{
        local float scratch[2*CHUNK];
        float sum=0;
        int i,j, offset_for_copy;
        event_t copy_complete;
        // prolog
        copy_complete = async_work_group_copy(scratch, A, CHUNK, 0);
        wait_group_events(1, &copy_complete);
        for (j=0;j<NUM_CHUNKS;j++)
        {
          offset_for_copy = (j%2) ? 0 : CHUNK;
          copy_complete =
             async_work_group_copy (scratch+offset_for_copy, A+(j+1)*CHUNK, CHUNK, 0);
          if (j%2) for (i=0;i<CHUNK;i++)  sum+=scratch[i+CHUNK];
          else for (i=0;i<CHUNK;i++)  sum+=scratch[i];
          wait_group_events (1, &copy_complete); // wait for transfer to be complete
        }
        // epilog loop goes here .. Not shown
        B[0]=sum;
}
```

Figure 2: Vector Summation Kernel - Asynchronous Implementation

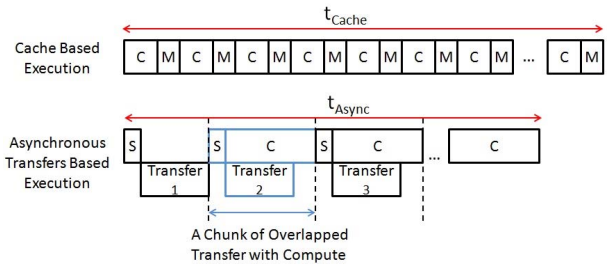

Figure 3: Timelines of Cache and Async Kernels

In Figure 2, it may be observed that each iteration of the outermost loop (loop control variable j), one half of the *scratch* buffer is used for computation and the other half for transfers.

As is evident, the chunk size of computation/transfer plays a key role in the overall efficiency of an asynchronous implementation. Too small a chunk size incurs the transfer setup overhead a large number of times. Too large a chunk size leads to large sequential overheads in the prolog and epilog portions of the kernel. We executed both the *CB* and *AT* versions of the vector-summation kernel (*fvec-sum*) and measured the speed-up obtained by the *AT* version at different chunk sizes. While details of our experimental evaluation are presented in Section IV, we note that these experiments were run on one of the eight DSPs present in the Texas Instruments Keystone-II platform [10]. The performance of the *AT* versions (in CPU cycles) at different chunk sizes is plotted in Figure 4. The horizontal line is the cycle count of the *CB* version. At a chunk size of 16KB, the *AT* version achieves the highest speed-up of 48% over

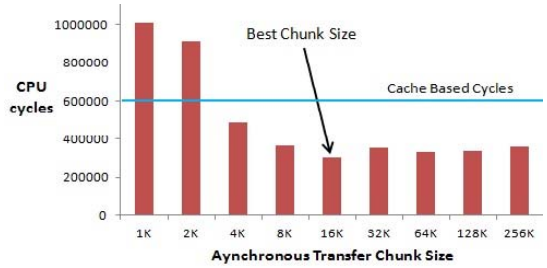

Figure 4: *fvec-sum*: Performance of the *AT* version at different chunk sizes

Parameter	Description
t_C	Computation time of the Kernel
t_M^{Cache}	Exposed memory access time of the *CB* Kernel
t_M^{Async}	Total memory transfer time of the *AT* Kernel
m	Average number of CPU Cycles between consecutive memory accesses
t_S	Asynchronous transfer overhead per each transfer
h_1	L1D Cache hit rate
h_2	L2 Cache hit rate
l_1	L1D Cache hit latency
l_2	L2 Cache hit latency
t_{DDR}	DDR (Off-Chip) memory access time per cache miss
d_e	*DMA* transfer efficiency relative to cache
n	Number of *DMA* chunks
p^*	Average memory access penalty
t^*_{Cache}	Total execution time of the cache-based kernel
t^*_{Async}	Total execution time of the async transfer-based kernel

Table I: Notation used in the analytical models

the *CB* version. To emphasize, this speed-up is achieved in a highly cache-friendly algorithm and thus motivates a deeper analysis of the benefits of using asynchronous transfers in such media processors. Next we develop a performance model to predict the estimated benefits of the *AT* version as a function of key system parameters.

III. Performance Evaluation Model

First we present a speed-up model to understand the extent of improvements achievable with *AT* versions. We then develop a model for estimating good *DMA* chunk sizes in single core kernels in Section III-B. This is followed by a generalization to cover multiple cores in Section III-C. Before we develop these models, we introduce some notation that is defined in Table I. All parameters in units of time are measured in CPU cycles. Parameters marked with a * are evaluated using other parameters listed in the Table.

A. Speed-up Model

It is easy to see that the total execution time of the cache-based kernel can be approximated by: $t_{Cache} \approx t_C + t_M^{Cache}$. This approximation assumes that cache-based loads can not be overlapped with computation. For cores that can (partially) hide the latencies of cache-based loads, it is assumed that the parameter t_M^{Cache} is suitably adjusted to

account for the exposed penalty of such loads. This reflects typical media processor implementations wherein features such as out-of-order execution, branch prediction and hardware multi-threading are omitted in favor of sophisticated computing units. We say that the kernel is *Compute-Bound* if $t_C \geq t_M^{Cache}$, and *Memory-Bound* otherwise.

By use of asynchronous transfers, if the total asynchronous transfer time t_M^{Async} is overlapped with useful computation, then the total execution time t_{Async} can be approximated as: $t_{Async} \approx max(t_C, t_M^{Async})$. This assumes ideal conditions – i.e., no *DMA* control overhead. For illustrating achievable speed-up, this approximation suffices. In the detailed model developed in the next Section, we account for this control overhead. The speed-up S achieved by the *AT* version is given by:

$$S = \frac{t_{Cache}}{t_{Async}} = \frac{t_C + t_M^{Cache}}{max(t_C, t_M^{Async})} \quad (1)$$

Using this Equation, we can develop lower and upper-bounds for the achievable speed-up of the he *AT* version over the *CB* version. In general the achieved speed-up of the *AT* version over the *CB* version depends on the *Compute-Bound* vs *Memory-Bound* nature of both the *AT* and *CB* versions. The four scenarios that arise and the resulting expected speed-ups are listed in Table II. S denotes the expected speed-up in each scenario. These cases are described briefly below.

Both* CB *and* AT *versions are* Compute-Bound*: From Equation 1, since $t_C > t_M^{Cache}$ and $t_C > t_M^{Async}$, it follows that $1 \leq S \leq 2$.

CB *is* Memory-Bound *and* AT *version is* Compute-Bound*:* Using Equation, it can be seen that $S = \frac{t_C + t_M^{Cache}}{t_C} = 1 + \frac{t_M^{Cache}}{t_C} > 2$. Thus arbitrarily large speed-ups become possible depending on the value of $\frac{t_M^{Cache}}{t_C}$.

CB *is* Compute-Bound *and* AT *version is* Memory-Bound*:* In the scenario that a *Compute-Bound* kernel in the cache version turned into a *Memory-Bound* kernel in the asynchronous version (i.e., $t_M^{Async} \geq t_C \geq t_M^{Cache}$), then the achievable speed-up is ≤ 2 and could even be ≤ 1 if $t_M^{Async} \gg t_M^{Cache}$. This is highly unlikely in practice since bulk *DMA* transfers tend to be more efficient than cache accesses. Such a situation might arise due to excessive *DMA* transfers compared to transfers made in the cache version. It suggests that the kernel is not well suited to an *AT* implementation (such as accesses not being contiguous thereby requiring too many small DMAs, or wasteful transfers for example). Among the four scenarios, this is the only scenario in which *AT* versions could be detrimental.

Both* CB *and* AT *versions are* Memory-Bound*: In this scenario (i.e., $t_M^{Cache} \geq t_C, t_M^{Async} \geq t_C$), the expected speed-up depends on the relative values of the two ratios, $\frac{t_C}{t_M^{Async}}$ and $\frac{t_M^{Cache}}{t_M^{Async}}$. The first ratio – $\frac{t_C}{t_M^{Async}}$ is ≤ 1 since the *AT* version is *Memory-Bound*, while the second ratio – $\frac{t_M^{Cache}}{t_M^{Async}}$ is generally ≥ 1 assuming that asynchronous memory transfers are more efficient. We can only conclude that the speed-up is ≥ 1.

Speed-up	***Compute-Bound* *CB* version**	***Memory-Bound* *CB* version**
***Compute-Bound* *AT* version**	$1 \leq S \leq 2$	$S > 2$
***Memory-Bound* *AT* version**	$0 < S \leq 2$ (unlikely scenario)	Generally $S \geq 1$. Depends on actual value of $t_C, t_M^{Cache}, t_M^{Async}$

Table II: Speed-up of *AT* versions

Having set up the context in which to view the relative performances of *AT* and *CB* versions, next we develop a model for estimating good *DMA* chunk sizes using values of t_{Cache} and t_{Async} estimated using key system and workload parameters. First we develop this model assuming a single core executing the kernel followed by an extension to cover multi-core kernels.

B. Estimating Good DMA *Chunk Sizes*

In this section, our goal is to develop a framework for determining if the *AT* version can out-perform the *CB* version and to determine the optimal transfer sizes in the *AT* version. Note that the purpose of the model is only to provide insight into what the key governing parameters and their interactions are, and not to provide fully-tuned/optimized results for arbitrary kernels.

First, we recall that the total execution of the *CB* version is given by: $t_{Cache} = t_C + t_M^{Cache}$. The total memory penalty t_M^{Cache} is a function of the rate m at which memory accesses occur, their hit rates h_1, h_2 in the caches and the off-chip memory penalty, t_{DDR}. The average memory penalty p per access is given by: $p = h_1 \cdot l_1 + (1-h_1) \cdot h_2 \cdot l_2 + (1-h_1) \cdot (1-h_2) \cdot t_{DDR}$. This is just the weighted average of the access latencies associated with $L1D$, $L2$ and DDR. These are workload-specific values. Since there is a memory access every m cycles, the total memory access penalty, t_M^{Cache}, may be expressed as: $t_M^{Cache} = \frac{t_C \cdot p}{m}$.

Thus we can determine if the kernel is *Compute-Bound* or not by the check: $t_C \geq \frac{t_C \cdot p}{m}$. This simplifies to: $m \geq p$. In other words, a *CB* version is *Compute-Bound* if there are atleast p cycles of computation between successive memory accesses. Thus, using kernel-specific values of these parameters, the model can be used to decide if the kernel is *Compute-Bound* or *Memory-Bound*. The total execution time of the *CB* version may now be expressed as:

$$t_{Cache} = t_C + t_M^{Cache} \quad (2)$$

$$= t_C + \frac{t_C \cdot p}{m} \quad (3)$$

$$= t_C(1 + \frac{p}{m}) \quad (4)$$

For the *AT* version, we assume that there is a non-overlapped initial setup (t_S) and transfer time (t_{Init}) followed by

overlapped transfers and computations (refer Figure 3). The transfer time t_M^{Async} is the total time taken to transfer the same amount of data as that accessed by the *CB* version. If there are n transfers made, and assuming that all transfers are of equal size, we can write: $t_{Init} = \frac{t_M^{Async}}{n}$. That is, the time for the initial non-overlapped transfer is an n^{th} of the total transfer time. Note that, for simplicity of the model, we assume that the scratch-pad memory has the capacity to store the data needed for large transfers.

Owing to increased transfer efficiency of large DMA transfers, t_M^{Async} is expected to be $\leq t_M^{Cache}$. Thus, using the *DMA* transfer efficiency parameter d_e (generally, $0 < d_e < 1$), we estimate t_M^{Async} as: $t_M^{Async} = d_e \cdot t_M^{Cache}$. If there are n transfers made, then the total transfer setup overhead is: $n \cdot t_S$. Thus the total computation is estimated to be: $t_C^{Async} = n \cdot t_S + t_C$ (since the total computation time now includes the cost of the setup overhead). The total execution time of the *AT* version may now be expressed as: $t_{Async} = max(t_C^{Async} + t_{Init}, n \cdot t_S + t_M^{Async} + \frac{t_C}{n})$.

Compute-Bound AT *Version* Here, we assume that the *AT* version is *Compute-Bound*. Thus:

$$t_{Async} = t_C^{Async} + t_{Init} \tag{5}$$
$$= n \cdot t_S + t_C + \frac{t_M^{Async}}{n} \tag{6}$$
$$= n \cdot t_S + t_C + \frac{d_e \cdot t_M^{Cache}}{n} \tag{7}$$
$$= n \cdot t_S + t_C + \frac{d_e \cdot t_C \cdot p}{m \cdot n} \tag{8}$$
$$= n \cdot t_S + t_C(1 + \frac{d_e \cdot p}{m \cdot n}) \tag{9}$$

We would like to seek a condition on n for the *AT* version to be more efficient than the *CB* version. Thus we seek that: $t_{Async} < t_{Cache}$. Substituting for the expressions of t_{Cache} and t_{Async} using Equations 4 and 9, we get:

$$n \cdot t_S + t_C(1 + \frac{d_e \cdot p}{m \cdot n}) < t_C(1 + \frac{p}{m}) \tag{10}$$

This equation represents the condition for t_{Async} to be better (smaller) than t_{Cache} as a function of several parameters.

Solving for n: Observe that Equation 9 is quadratic in n, the number of transfers. If all the other parameters are known, then it is possible to solve for the desired value of n that minimizes t_{Async}[2]. In practice, this ideal value n^* may not be feasible due to implementation aspects (too small a scratch-pad, for example).

In Section V, we show that such an optimal number of transfers indeed exists and that the model estimates n^* accurately. At the optimal n^*, the speed-up S achieved by the *AT* version is given by:

$$S^* = \frac{t_C(1 + \frac{p}{m})}{n^* \cdot t_S + t_C(1 + \frac{d_e \cdot p}{m \cdot n^*})} \tag{11}$$

Memory-Bound AT *Version* We proceed with the development of the condition for the *Memory-Bound AT* version to outperform the *CB* version along similar lines. The total execution time of the *AT* version is now given by:

$$t_{Async} = n \cdot t_S + t_M^{Async} + \frac{t_C}{n} \tag{12}$$
$$= n \cdot t_S + d_e \cdot t_M^{Cache} + \frac{t_C}{n} \tag{13}$$
$$= n \cdot t_S + d_e \cdot \frac{t_C \cdot p}{m} + \frac{t_C}{n} \tag{14}$$
$$= n \cdot t_S + t_C \cdot (\frac{d_e \cdot p}{m} + \frac{1}{n}) \tag{15}$$

Equation 12 follows from the assumption that the *AT* version is *Memory-Bound*. Thus the total execution time is a sum of the total transfer setup time, $n \cdot t_S$, the total transfer time, t_M^{Async} and the time for the last (non-overlapped) chunk of compute, $\frac{t_C}{n}$. Requiring that $t_{Async} < t_{Cache}$ leads to the inequality:

$$n \cdot t_S + t_C \cdot (\frac{d_e \cdot p}{m} + \frac{1}{n}) < t_C(1 + \frac{p}{m}) \tag{16}$$

As in the preceeding Section, this leads to a quadratic in n and the optimal value that minimizes t_{Async} can be determined from the roots of the quadratic.

The speed-up S^* obtained by the *AT* version at the optimal value n^* is given by:

$$S^* = \frac{t_C(1 + \frac{p}{m})}{n^* \cdot t_S + t_C \cdot (\frac{d_e \cdot p}{m} + \frac{1}{n^*})} \tag{17}$$

Putting it together: In summary, the speed-up model uses workload-specific parameters ($t_C, h_1, h_2, l_1, l_2, t_{DDR}$, and m) and platform-specific parameters (t_S and d_e) to classify the workload as *Compute-Bound* or *Memory-Bound* in both the *CB* and *AT* versions. After classification, the execution times t_{Cache} and t_{Async} are appropriately estimated. The model estimates an optimal value of t_{Async} by identifying the best value of n, the number of *DMA* transfers initiated in the *AT* version. This enables the estimation of the best speed-up of the *AT* version over the *CB* version.

C. Extension to Multi-Core

We develop a simple extension to model multi-core configurations. Assuming that kernel computation and memory accesses are uniformly distributed across all the available C cores, the work performed by each kernel scales by a factor of $\frac{1}{C}$. However, the use of shared memory and on-chip network resources causes congestion and reduced efficiency of memory accesses. While a single-core kernel is likely to obtain higher row-buffer hit rates and fewer bank conflicts in the DDR main memory, the multi-core kernel is likely

[2]It is noted that finding the best n^* is equivalent to finding the best *DMA* transfer chunk size.

Name	Description	Parameters
fvec-sum	Summation of a floating-point vector	256K elements
cvec-sum	Summation of a vector of characters	256K elements
fvec-copy	Copy a floating-point vector	256K elements
cvec-copy	Copy a vector of characters	256K elements

Table III: Listing of Micro-kernels

to suffer from greater row-buffer misses and bank conflicts. Thus while the computation per core reduces from t_C to $\frac{t_C}{C}$, the average memory access penalties tend to worsen. This is taken into account in the model by suitably adjusting the off-chip memory access timing parameter t_{DDR}, as well as the *DMA* transfer efficiency factor d_e.

As our results indicate, this simple extension provides a reasonably good approximation of the behavior observed in multi-core systems.

IV. Experimental Evaluation

We evaluated the performance benefit of asynchronous data transfers using both micro-kernels as well as realistic image processing kernels on the Texas Instruments Keystone-II multi-core platform [10]. We describe the details of our workloads, hardware platform and performance metrics below.

A. Workloads

We used two types of workloads for evaluation: i) directed micro-kernels that emphasize the impact of asynchronous data transfers, and ii) image processing kernels that represent realistic workloads. Directed micro-kernels are listed in Table III and the image processing kernels in Table IV. Each kernel is implemented in two flavors – a cache-based implementation (denoted by a *cache* subscript) as well as an asynchronous data transfer based implementation (denoted by a *async* subscript).

1) Micro-Kernels: The first two micro-kernels – *fvec-sum* and *cvec-sum* are cache-friendly kernels dominated by load operations in the cache-based versions. In the *async* version, these kernels leverage the *async_work_group_copy* API to move chunks of data into a local scratchpad using the double-buffering technique. The next two kernels – *fvec-copy* and *cvec-copy* have equal amounts of loads and stores and thus reveal the performance benefits of using asynchronous transfers.

As predicted by our model in Section III, the chunk size of transfers used in the *async* version matters – small sizes incur too much *DMA* control overhead while very large sizes suffer from the serial overhead of the initial transfer whose time can not be hidden. Unless otherwise stated, we use a chunk size of 16KB per *async_work_group_copy* call.

2) Image Processing Kernels: Image processing kernels listed in Table IV are low-level primitives used by vision and imaging applications. In general, they exhibit cache-friendly behavior (with the exception of the histogram kernels which perform writes to an output array using pixel values to compute the array index). They also have higher compute-to-memory ratios as compared to the micro-kernels.

Name	Description	Parameters
Image_Diff_S	Image Difference	Img. Size: 640X360
Image_Diff_B	Image Difference	Img. Size: 1920X1080
Max_S	Find max. pixel value	Img. Size: 640X360
Max_B	Find max. pixel value	Img. Size: 1920X1080
Histogram_S	Histogram of pixel values	Img. Size: 640X360
Histogram_B	Histogram of pixel values	Img. Size: 1920X1080
Gaussian_S	Gaussian filter over image	Img. Size: 640X360
Gaussian_B	Gaussian filter over image	Img. Size: 1920X1080
Image_Resize_B	Image Resize	Input Image: 1920X1080 Output Image: 960X540
Imaging_App	Sample Application: downsizing, noise filter, histogram	Input Image: 1920X1080

Table IV: Listing of Image Processing Kernels

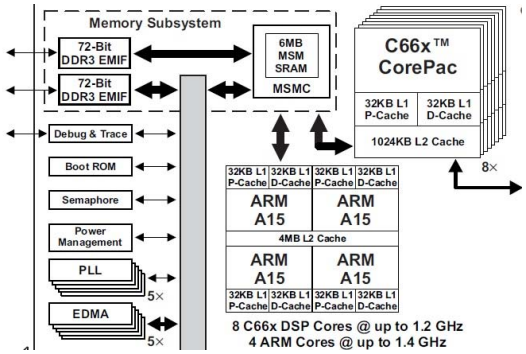

Figure 5: Overview of TI Keystone-II Multi-Core SoC

For completeness, we developed an imaging application that performs a series of computations: image downsizing (from 1920×1080 to 640×360), noise reduction via a 3×3 Gaussian filter and finally a 256-bins histogram computation. The application is implemented as three OpenCL kernels that are invoked in a pipeline.

B. Evaluation Platform

We evaluated the performance of the *cache* and *async* versions on the Texas Instruments Keystone-II multi-core SOC [10]. A diagrammatic overview of the architecture is presented in Figure 5.

The SOC comprises 4 ARM Cortex-A15 cores and 8 C66x [14] DSP cores. In our study, we use only the DSP cores as OpenCL targets. These DSP cores are 8-way VLIW (Very Long Instruction Word) cores operating at a peak of 1.228GHz. Each core has a 2-level cache hierarchy [15] comprising separate level-1 instruction and data caches and a unified level-2 cache. The level-2 memory is configurable – it can be used as a full cache or a full scratch-pad or as a combination of both. Across the DSP cores, the caches are not coherent. Thus to avoid false-sharing, the caches are configured for *write-through*. At the main memory con-

Platform	Cores, Freq.	Caches	Off-Chip
TI K2H (DSP)	8 cores, 1.228 GHz	L1:32KB cache L2: 1MB Used as: 1MB local, or 768KB cache	DDR3 64-bit 1333MHz

Table V: Hardware Configuration

troller, such writes are implemented by atomic *read-modify-write* operations on the memory. The cores access off-chip DDR3 memory via two 64-bit data channels operating at a peak of $800MHz$ (i.e., $1600M$ transfers per second). In our evaluation, we used only one 64-bit data channel operating at a speed of $666MHz$ (i.e., $1333M$ transfers per second). Key parameters of this platform are summarized in Table V [3].

The DMA engine [16] on this platform is capable of 1D and 2D transfers, and supports both pipelined and concurrent transfers between on-chip and off-chip memories.

The DSP has inherently limited hardware parallelism (it does not support multiple hardware threads of execution and has SIMD width of upto 8) and thus benefits from kernels that perform more work in each *work-item* as compared to kernels that are written for highly parallel architectures such as GPUs wherein each *work-item* performs a smaller amount of work. Further, as generally observed on a variety of platforms, dispatching more *work-groups* than the available compute units increases dispatching and scheduling overheads. Thus we restrict our attention to two types of kernel compositions, both of which map efficiently onto the underlying hardware platform: one, single-core kernels that are comprised of just one *work-group* and one *work-item*; and two, multi-core kernels that are comprised of 8 *work-groups* and one *work-item* per *work-group*. Having just one *work-item* per *work-group* exposes significant pipelining, loop optimization and VLIW scheduling opportunities to the compiler. Single-core kernels reveal the benefits of asynchronous transfers when main memory is dedicated to just one core (lower access penalties), while multi-core kernels reveal the benefits of asynchronous transfers when main memory is shared across all the 8 cores.

C. Parameters used by the Model

The parameters t_C, m, h_1, h_2 and t_{DDR} required by the model are obtained by turning on hardware profiling while the kernel executes. We remark that t_{DDR} is kernel-dependent since the memory penalty depends on the congestion at the main memory, the spatial locality of accesses, as well as other factors such as bank conflicts and the frequency of read-to-write turn-arounds. The $L1D$ access penalty l_1 is set to 0 and $L2$ access penalty l_2 set to 5 cycles. The transfer setup overhead t_S is set to 870 CPU cycles and the *DMA* efficiency factor d_e set to 0.17 based on experimental observation across a range of DMA transfer sizes.

[3]Additionally, the chip has $6MB$ on-chip memory that is shared by all the cores (shown as MSM SRAM in the Figure). In our experimental study, we did not use this on-chip resource.

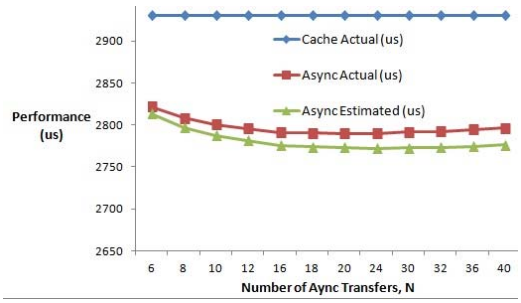

Figure 6: Validation of the Model on the *Compute-Bound* Kernel *Max_B*

D. Metrics

For each kernel workload W described in Section IV-A, we collect cycle counts of execution in both the *cache* and *async* versions, denoted W_{cache} and W_{async} respectively. The performance improvement is measured as the percentage improvement of W_{async} over W_{cache} (given by: $\left(\frac{W_{cache}-W_{async}}{W_{cache}}\right) \cdot 100$). To ensure consistent results, the caches are flushed before kernel execution and results are averaged over 5 runs for each workload. All the results reported in Sections V-B and V-C are actual results obtained on the evaluated platform.

V. Results and Discussion

We first present a validation of the performance model and its batch-size estimation technique. Next we present detailed results obtained on the evaluation platform using only one of the 8 DSPs followed by the results obtained on the 8-core version.

A. Model Validation

We validated the performance model for its accuracy in estimating efficient chunk sizes in both *Compute-Bound* and *Memory-Bound* kernels. The *Max_B* is a *Compute-Bound* kernel (i.e., $t_C > t_M^{Cache}$), and as per Table II, the *AT* version can not achieve speed-up in excess of 2. Figure 6 plots the observed (actual) execution time of the *CB* version, the observed (actual) execution time of the *AT* version and the model-estimated execution time of the *AT* version at different numbers of asynchronous transfers (model parameter n). The model deviates from the observed performance by less than 1%. The best speed-up achieved (actual) is $\approx 1.05\times$ which occurs at $n = 24$. The model's prediction is in agreement with this result.

A similar validation is performed on the *Memory-Bound* kernel, *fvec_copy*, reported in Figure 7. As per Table II, the *AT* version can result in arbitrarily large speed-up. Indeed,

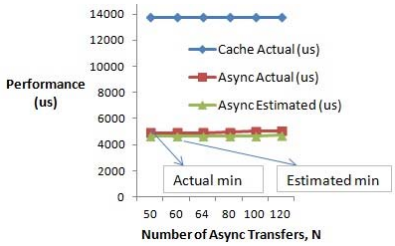

Figure 7: Validation of the Model on the *Memory-Bound* Kernel *fvec_copy*

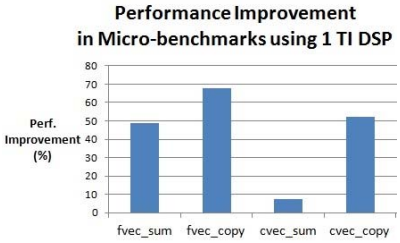

Figure 8: Performance of Asynchronous Transfers in Micro-Benchmarks on Single-Core DSP Platform

we observe the best speed-up of $\approx 2.5\times$ achieved by the *AT* version at $n = 50$[4]. Further, the model has less than 7% error in estimating overall execution time, t_{Async}, compared to the observed execution time. Similar results are observed in other workloads (not shown due to space constraints). On average, the model has an error of 6% in estimating the speed-up obtained by the *AT* versions over *CB* versions.

These results indicate that the model has sufficiently high accuracy to permit its use for a quick evaluation of the benefits of using asynchronous transfers.

It may be noted, however, that the results presented in the next two Sections are actual experimental results obtained on the evaluation platform.

B. Single-Core Results

For illustration, we discuss the performance benefits observed on micro-benchmarks first followed by a discussion of the performance improvements observed in image processing kernels.

1) Micro-benchmarks: Figure 8 plots the performance improvement obtained by the *AT* version of each micro-benchmarks over the corresponding *CB* version when the benchmarks were mapped to run on only one DSP. These single-core versions reveal the opportunities for performance improvement when the entire available main memory bandwidth is made available to a single core. Both the floating-point kernels gain significantly – nearly 50% or more. The character data type kernel – *cvec-sum* – gains the least ($\approx 7\%$) owing to its very high L1D and L2 cache hit rates reducing the opportunities for saving exposed memory transfer penalties[5]. Overall, despite the high cache hit rates of these kernels, the *AT* versions provide significant performance improvement by virtue of the efficient *DMA* transfers.

2) Image processing kernels: Performance improvements obtained by the *AT* versions in case of image processing kernels are plotted in Figure 9. We observe an average

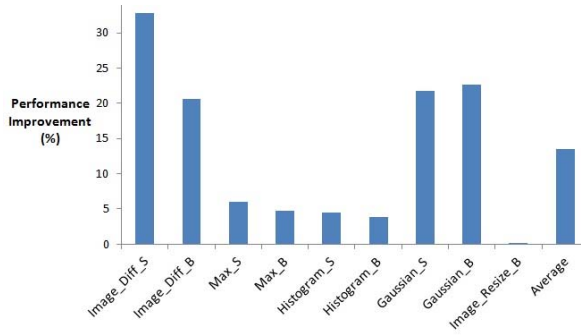

Figure 9: Performance of Asynchronous Transfers in Imaging Kernels on Single-Core DSP Platform

performance improvement of 14% in these workloads with the *Gaussian_B* kernel showing a $3X$ speed-up.

It can also be observed that the speed-ups are generally higher in workloads that load more data – such as the *Image_Diff* and *Gaussian* kernels which load two input image pixels per output pixel. This shows that asynchronous transfers are more effective in kernels that have higher memory bandwidth demand.

C. Multi-Core Results

In this Section, we evaluate the improvements seen in *AT* versions of OpenCL kernels mapped to leverage all of the eight DSP cores available on the TI Keystone-II device. Here, *work-groups* are mapped across these cores and can thus leverage the concurrency of 8 DSP cores. The cores however share the off-chip DDR memory via a single 64-bit channel and thus frequent off-chip accesses can potentially create contention at the main memory.

Figure 10 plots the performance improvements seen by the micro-benchmarks (the baselines used here are the corresponding cache versions of kernels that run on all 8 cores). The kernels that perform writes (*fvec_copy* and *cvec_copy*) show improvements of over $3.5\times$ over cache versions. These are mainly due to the benefits of removing the frequent interference between cache loads and writes initiated by each core in the *CB* version. Since the caches are write-through, frequent writes fill up write buffers quickly and subsequent writes/loads cause cores to stall. Further,

[4]The actual and model-estimated execution times look nearly flat due to very little variation in execution time with varying n.

[5]On the otherhand, *cvec-copy* sees much higher benefit owing to the overhead of frequent writes that leads to a performance bottleneck in the *CB* version.

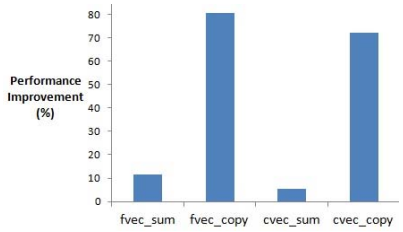

Figure 10: Performance of Asynchronous Transfers in Micro-Benchmarks on 8-Core DSP Platform

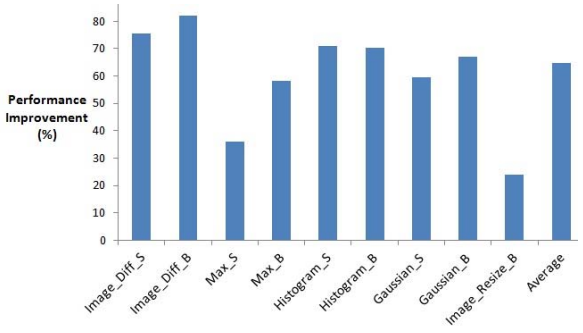

Figure 11: Performance of Asynchronous Transfers in Imaging Kernels on 8-Core DSP Platform

at the shared main memory, the reads and writes from different cores conflict causing increased row-buffer misses and associated access penalties. The *AT* versions reduce this interference substantially by allowing each *DMA* request to be efficiently serviced through the memory hierarchy.

Figure 11 plots the performance improvements seen in image processing kernels (baselines being the correspoding cache versions that run on all 8 cores). Substantial performance improvements are observed in all of the kernels with average improvement of 65%. In all the kernels except *Max_S* and *Image_Resize_B*, the *AT* versions improve performance by over $2X$. Referring to our speed-up discussion in Section III-A and Table II, we can infer that these kernels are *Memory-Bound* in the cache version. Thus the *AT* implementations of these kernels have successfully hidden the memory penalty enabling speed-ups ≥ 2. While the *Image_Resize_B* kernel hardly achieves any improvement in the single-core configuration (refer Figure 9), it is able to gain over 20% improvement in performance in the multi-core configuration.

On the end-to-end imaging application, the *AT* version showed a 24% improvement in performance over the *Compute-Bound* version.

D. Discussion

Low-level image processing kernels such as those considered in this work perform small amounts of computation per pixel. Such kernels can easily become *Memory-Bound* in the cache versions. For instance, among multi-core kernels, the *Image_Diff_B* kernel, which computes (absolute) differences between pairs of pixel values, is *Memory-Bound*. On the otherhand, the *Max_S* kernel that loads just one image and finds the maximum pixel value in the image is *Compute-Bound*. Thus the nature of the cache-based kernel can be quite sensitive to the specifics of the kernel implementation and can quickly change from *Compute-Bound* to *Memory-Bound*.

It should be noted that the benefits of *AT* versions are a result of the characteristics of the underlying processor architecture – single-threaded cores that are inherently limited in their ability to hide long latency stalls arising from off-chip memory accesses. On such cores, it is important to reduce stall penalty and background transfers provide a very effective means of achieving this objective. Further, multi-core configurations comprising such cores can get adversely affected by the interference effects caused on shared resources. In general, the *AT* versions provide a more robust performance behavior and are less sensitive to memory penalties. The *AT* versions leverage spatial locality in row-buffer accesses and incur fewer bank conflicts in off-chip accesses resulting in more efficient bulk data transfers. Thus these techniques have a much bigger benefit when applied to multi-core kernels as the results in Section V-C suggest. While the average improvement of *AT* versions is $< 15\%$ in single-core, the average improvement is greater than 65% in multi-core as reported in Figures 9 and 11.

On the other hand, architectural features such as support for multiple hardware threads, low-latency thread switching, memory prefetching, and out-of-order execution can alleviate the severity of cache-based off-chip memory accesses. A study of the benefits and challenges associated with leveraging these features in hiding the observed memory latency in OpenCL kernels is beyond the scope of this paper.

VI. Related Literature

Use of *DMA* controllers and double-buffering schemes is not new. Works in [17], [18] and [19] discuss the use of these mechanisms in systems with explicitly managed scratch-pad resources. Our work aims to demonstrate the significant performance benefit of these mechanisms in the context of the popular programming model. The performance model proposed in our work reveals the key governing parameters affecting overall performance and provides a quantitative framework for analysis of OpenCL workloads.

The work in *ROS-DMA* [13] proposes a model to estimate good *DMA* batch sizes, but does not provide a comparison with cache-based implementations. It is also restricted to a single core and as such does not address efficiency issues in multi-core architectures.

Several works have modeled the performance of GPU architectures [20]–[22]. These prior models are focused on identifying and modeling critical architectural elements in the context of throughput-oriented architectures. Further,

they do not compare the benefits of asynchronous transfers over cache-based kernels.

The works in *DBDB* [23] and *PolyMage* [24] are compiler techniques for obtaining optimized executables for different hardware platforms. Our model could be integrated into these works to let the compiler automatically generate *AT* versions where appropriate.

VII. Conclusion

In this paper, we examined the performance benefits of the use of the *async_work_group_copy* and *wait_group_events* OpenCL APIs to initiate background memory transfers using *DMA* engines in a typical media processor platform. We developed a performance model to assess the impact of various key governing parameters and showed how large speed-ups are possible when a memory-bound kernel in the cache-based version turns into a compute-bound kernel in the asynchronous-transfers version due to highly efficient DMA transfers. We experimentally observed speed-ups of upto $5\times$ and average performance improvement of 40% in image processing kernels running on an 8-core DSP platform.

References

[1] Khronos, "The open standard for parallel programming of heterogeneous systems," 2009. [Online]. Available: https://www.khronos.org/opencl/

[2] Intel, "OpenCL Technology." [Online]. Available: https://software.intel.com/en-us/intel-opencl

[3] AMD, "OpenCL Zone." [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/

[4] NVIDIA, "OpenCL." [Online]. Available: https://developer.nvidia.com/opencl

[5] ARM, "ARM Mali OpenCL SDK." [Online]. Available: http://malideveloper.arm.com/develop-for-mali/sdks/mali-opencl-sdk/

[6] Imagination, "Imaginations PowerVR Series6 is the first mobile GPU to pass OpenCL 1.2 conformance with Khronos." [Online]. Available: http://www.imgtec.com/news/detail.asp?ID=858

[7] TI, "OpenCL." [Online]. Available: http://processors.wiki.ti.com/index.php/OpenCL

[8] ISO, "ISO C Standard 1999," Tech. Rep., 1999. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[9] M. Akhloufi and A. Campagna, "OpenCLIPP: OpenCL Integrated Performance Primitives library for computer vision applications," *Proc. SPIE Electronic Imaging 2014, Intelligent Robots and Computer Vision XXXI: Algorithms and Techniques*, 2014. [Online]. Available: https://github.com/CRVI/OpenCLIPP

[10] TI, "66AK2H06: Multicore DSP+ARM KeyStone II System-on-Chip (SoC)," 2014. [Online]. Available: http://www.ti.com/product/66ak2h06

[11] Khronos, "async_work_group_copy," 2009. [Online]. Available: https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/async_work_group_copy.html

[12] ——, "wait_group_events," 2009. [Online]. Available: https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/wait_group_events.html

[13] C. Zinner and W. Kubinger, "ROS-DMA: A DMA Double Buffering Method for Embedded Image Processing with Resource Optimized Slicing." in *IEEE Real Time Technology and Applications Symposium*. IEEE Computer Society, 2006.

[14] TI, "TMS320C66x DSP CPU and Instruction Set User Guide," 2010. [Online]. Available: http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf

[15] ——, "TMS320C66x DSP CorePac User Guide," 2013. [Online]. Available: http://www.ti.com/lit/ug/sprugw0c/sprugw0c.pdf

[16] ——, "KeyStone Architecture Enhanced Direct Memory Access (EDMA3) Controller," 2015. [Online]. Available: http://www.ti.com/lit/ug/sprugs5b/sprugs5b.pdf

[17] S. Saidi, P. Tendulkar, T. Lepley, and O. Maler, "Optimizing two-dimensional DMA transfers for scratchpad based MPSoCs platforms," *Microprocessors and Microsystems - Embedded Hardware Design*, no. 8, 2013.

[18] ——, "Optimizing Explicit Data Transfers for Data Parallel Applications on the Cell Architecture," *ACM Trans. Archit. Code Optim.*, 2012.

[19] S. Schneider, J.-S. Yeom, and D. S. Nikolopoulos, "Programming multiprocessors with explicitly managed memory hierarchies." *IEEE Computer*, 2009.

[20] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, 2009.

[21] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures." in *HPCA*, 2011.

[22] M. Lin, R. Chamberlain, and K. Agrawal, "Performance modeling for highly-threaded many-core GPUs," in *IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2014.

[23] T. Liu, H. Lin, T. Chen, J. K. O'Brien, and L. Shao, "DBDB: Optimizing DMA Transfer for the Cell Be Architecture," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09, 2009.

[24] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic Optimization for Image Processing Pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, 2015.