Heterogeneous Computing Pipelining

1 CONTENTS

2 Abstract.................................................................................................................................................................................. 1
3 Intel® Core™ and Atom™ family Processors and Heterogeneous Processing Architecture .......... 2
4 Introduction ............................................................................................................................................................................. 3
5 Model ...................................................................................................................................................................................... 4
6 Memory requirements ............................................................................................................................................................. 6
7 Implementation – TBB+OpenMP+OpenCL ............................................................................................................................. 7
8 Performance results ................................................................................................................................................................. 8
9 Conclusion .................................................................................................................................................................................. 9
10 About the author ................................................................................................................................................................. 9
11 References ............................................................................................................................................................................... 9
12 Notices .................................................................................................................................................................................. 10

2 ABSTRACT

Most image and video effects consist of two or more stages. Software developers often execute them sequentially, one by one. The easy way to performance tune for such algorithms is to optimize each stage separately using vectorization, multi-threading and/or graphics offload. But, processing of the stages still happens sequentially. The Intel® processor provides multiple opportunities to offload and process the algorithms in parallel for the best performance with optimum power usage. Developers can always optimize an algorithm for one device (e.g. CPU or Intel® Processor Graphics), however they ignore significant performance increases by not using other processing components on the chip. Things get more complex if the application distributes the workload among different devices and while processing in parallel. In this article we will review an efficient method to use all available resources on Intel® CPUs via software pipelining CPU and OpenCL code.
3 Intel® Core™ and Atom™ Family Processors and Heterogeneous Processing Architecture

Intel® Core™ and Atom™ processor families provide system-on-a-chip (SoC) solutions that use multiple resources for heterogeneous compute and media processing. Figure 1 shows Intel® Core™ M Processor silicon die layout.

Figure 1. Silicon die layout for an Intel® Core™ M Processor. This SoC contains 2 CPU cores, outlined in orange dashed boxes. Outlined in the blue dashed box, is Intel® HD Graphics 5300. It is a one slice instantiation of Intel® Processor Graphics [7].

The Intel® microprocessors are complex SoCs integrating multiple CPU Cores, Intel® Processor Graphics, and other fixed functions all on a single shared silicon die. The architecture implements numerous unique clock domains, including a per-CPU core clock domain, a processor graphics clock domain, and a ring interconnect clock domain. The SoC architecture is designed to be extensible, and yet still enable efficient wire routing between components.
The 2\textsuperscript{nd} Generation Intel\textsuperscript{®} Core\textsuperscript{™} processor expanded heterogeneous media processing with the introduction of Intel\textsuperscript{®} Quick Sync Video (Intel\textsuperscript{®} QSV) through Intel\textsuperscript{®} Media SDK [9]. These kinds of innovations continued with the 3\textsuperscript{rd} Generation Intel\textsuperscript{®} Core\textsuperscript{™} processor and the support of OpenCL\textsuperscript{™} 1.2. It allowed applications to do heterogeneous computing on Intel\textsuperscript{®} Processor Graphics [7]. The new Intel\textsuperscript{®} Core M processors added more flexibility and programming ease with OpenCL\textsuperscript{™} 2.0 support. Intel\textsuperscript{®} Core\textsuperscript{™} processors already supported shared physical memory so applications shared data between the CPU and Intel\textsuperscript{®} processor graphics, but OpenCL\textsuperscript{™} 2.0 shared virtual memory support allows applications to share the data structures seamlessly between two devices.

4 \textbf{INTRODUCTION}

Intel\textsuperscript{®} processor architecture provides a cohesive environment for applications by sharing resources (e.g. caches) between the CPU and processor graphics. We will to take advantage of these features through software pipelining.

Let’s review a general algorithm execution model consisting of $N$ stages.

Let $T_i$ be the time to execute $i$-th stage, then the total execution time might be estimated as $\sum T_i$.

The straightforward way to optimize is to performance tune each stage individually via scalar code tuning, multi-threading, vectorization [1] or offloading to processor graphics (OpenCL etc.).
approach could be a first step, but might limit performance opportunities. For example, optimizing one out of five stages to be twice as fast might yield only ~10% overall performance increase if all stages take about the same time.

One can achieve parallelization by running multiple effects at the same time, as explained in “Expressing Pipeline Parallelism using TBB constructs” by Eric Reed, Nicholas Chen and Ralph Johnson [2]. Unfortunately, it is applicable only when there are multiple input data (frames) available at the same time. It may not be used, for example, in photo editing. The method might also underutilize available resources, and add additional memory bandwidth requirements. This method is still worth consideration and evaluation as it might be useful for many kinds of video processing algorithms.

We will show you a way to perform parallel stage execution on both devices, which can improve performance, data locality, and uses all available resources effectively. Both methods discussed can be used at the same time.

## 5 Model

To simplify the discussion, we will review the short version of an effect with only two stages.

As we want to distribute the execution between CPU and GPU we need to implement good and optimized versions of Stage 1 and Stage 2 algorithms. Suppose Stage 1 is optimized for CPU and Stage 2 for GPU. In order to successfully implement pipelined model one must be able to implement efficient tiled version of the algorithm as shown on the Figure 5.
Key to this method is to execute Stage 1 on the CPU for one tile at a time while finishing the previous tile on the GPU, as shown on Figure 6. Store Stage 1’s output data in an intermediate buffer, which will be used on the next iteration to produce the output on GPU using Stage 2 algorithm. Wait for both CPU and GPU to finish their tasks and repeat the process. There will be two pipelines executing simultaneously, therefore we need two intermediate buffers to pass the data between the stages, which can be allocated once and rotated appropriately.
Let $t_1$ and $t_2$ be the execution time for Stage 1 and Stage 2, so the total execution time will be $T = t_1 + t_2$. If we ignore threading and synchronization overhead, we can estimate that the wall time is $T = \max(t_1, t_2) + (t_1 + t_2)/N$, where $N$ is the number of tiles. Understand that $\lim_{N \to \infty} T = \max(t_1, t_2)$, so theoretically we can hide one of the effects execution, and reduce the wall time by a factor of $\frac{t_1 + t_2}{\max(t_1, t_2)}$.

Unfortunately, it is not always possible to execute a “perfect” parallelization because synchronization and data transfer penalties grow with the additional number of tiles.

6 Memory Requirements

As stated in the beginning, we will show how proposed algorithm may reduce memory requirements as well as improve data locality. Let $S$ represent the input data (an image buffer, for example) and assume that we need $S$ space for intermediate data and the same amount of memory for output data, so the total working set size estimate is $3S$. In the proposed model we only need two intermediate buffers of $S/N$ size, therefore the total memory requirements will be $M = (2 + \frac{2}{N})S$, which is $< 3S$ for all $N > 2$. Thus we might expect total memory requirements to drop by a factor of $\frac{3S}{M} = \frac{3N}{2N+2}$. We would also like to note that $\lim_{N \to \infty} \frac{3N}{2N+2} = 3/2$.

Now count the number of tile buffers in the flight at a time - $2 \cdot \text{Input} + 2 \cdot \text{Intermediate} + 1 \cdot \text{Output}$ in the case when Stage2 uses the input data as well. Then the working set size would be $m = 5S/N$ vs. $2S$ for a baseline implementation, which is 1.6 times less for $N = 4$ and more than 6 times less for $N = 16$.

On the other side, as mentioned earlier, improved data locality and reduced working set size might also improve performance. One of the interesting cases would be when we choose the number of tiles in such a way that $m$ is equal or less than last level cache (LLC) size to avoid additional latency on the data transfer between the devices.
We implemented one of the photo editing effects with two stages using this methodology. We used all of the optimizations, from scalar optimization and vectorization, up to parallel computing using OpenMP and OpenCL.

First, we implemented “tiled” versions of both Stage1 (CPU/AVX2) & Stage2 (OpenCL) algorithms, as shown of Fig. 5, and used OpenMP to improve Stage1 performance and utilize all available CPU cores.

To apply software pipelining and parallel execution, we used Intel TBB [5], which is available open source and/or together with Intel the C/C++ compiler. To use tbb::pipeline for software pipelining, define stage classes (tbb::filter) and token structure. Token is what will be transferred between the stages and includes the tile information and other intermediate data. TBB manages token transfer to the next stage as (void*) operator input parameter.

In order to get the best performance, use zero-copy OpenCL capabilities [6] available in Intel® Graphics architecture or Shared Virtual Memory (SVM) available in OpenCL 2.0 stack and supported by Intel® Graphics starting from Broadwell.

```c
struct token_t {
    void *buffer;       // intermediate buffer
    int tile_id;        // Tile number
};

class Stage1: public tbb::filter {
public:
    void* operator() (void *) {
        if (tile_id == num_tiles)
            return null;       // tells TBB to stop pipeline execution

        // Create token and select preallocated buffer
        token_t *t=new token_t();
        t->tile_id = tile_id++;
        t->buffer = buffers[tile_id % 2]; // select one of the buffers

        DoStage1_CPU(t, input);   // Process tile on CPU

        return t;  // Transfer intermediate data to the next stage
    }
};

class Stage2: public tbb::filter {
public:
    void* operator(void* token) {
        token_t *t = (token_t*)token;     // Receive data from the previous stage

        DoStage2_OpenCL(output, t);        // Do second stage on the Processor Gfx
                                             // and write output data

        delete t;                        // Destroy token
        return 0;                        // Finish pipeline
    }
};
```
After that, pipeline (tbb::pipeline [4]) creation and execution looks pretty simple.

```cpp
// Create and initialize stages
Stage1 s1(input, W, H);
Stage2 s2(input, output, W, H);

// Create pipeline
tbb::pipeline ppln;
// Add filters(stages)
ppln.add_filter(s1);
ppln.add_filter(s2);
// Run 2 pipelines
// One will be executed on the CPU,
// while the other one on Processor Gfx
// and vice versa on the next “step”
ppln.run(2);
```

8 PERFORMANCE RESULTS

Ideally we want to have an unlimited number of tiles, but multi-threading and synchronization overhead might reduce the algorithm efficiency.

We’ve done experiments on Haswell i7-4702HQ @ 2.2GHz with numerous tiles and got up to 1.7x overall performance improvements over serial implementation. We also found out that four to eight tiles are optimal for our algorithm on this particular platform.

---

*Figure 7. Effect execution time.*
9 CONCLUSION

In this article we illustrated an effective algorithm to that distributed workloads between available computing resources, and gained up to 1.7x better performance by running the algorithm on both CPU and processor graphics and the same time. This approach can be extended to use other devices or features available on Intel® processors. One of most and very popular hardware feature in Intel Processor Graphics is Intel® Quick Sync Video (QSV) [9]. Applications can use same approach to divide the preprocessing stages and then feed this super-fast fixed function blocks to encode the frames.

10 ABOUT THE AUTHOR

Ilya Albrekht been working for Intel for 5 years as an Application Engineer and has strong skills in algorithms optimization, modern CPU microarchitecture and OpenCL. He has Master’s degree in Mathematics and Information Security. He loves to explore natural wonders in Arizona and California with his wife.

11 REFERENCES


INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel, the Intel logo, Core, Iris, and Iris Pro are trademarks of Intel Corporation in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

Copyright © 2014, Intel Corporation. All rights reserved.