The handle http://hdl.handle.net/1887/21017 holds various files of this Leiden University dissertation

Author: Balevic, Ana
Title: Exploiting multi-level parallelism in streaming applications for heterogeneous platforms with GPUs
Issue Date: 2013-06-26
M-JPEG Case Study

6.1 Introduction

To evaluate the benefits of the techniques presented in previous chapters, in this chapter we conduct a series of experiments on the M-JPEG encoder. The M-JPEG encoder is a streaming multimedia application from the realm of video compression that performs lossy still-image compression on the stream of input frames, and as a result generates an output data stream of reduced data size. Although the M-JPEG standard defines a relatively simple encoding workflow (in terms of video compression standards), it is still a very interesting application for a parallelization case study since it contains inherent task and data parallelism. On the one hand, the M-JPEG encoder is a typical streaming application, and as such it is easily modelled as a pipeline of tasks. On the other hand, it contains computationally intensive tasks, such as discrete cosine transform (DCT), that feature inherent data parallelism. As such, M-JPEG encoder provides rich experimentation opportunities with different types of parallelism. As the basis for our experiments, we adapted the M-JPEG encoder that was originally developed at LERC to demonstrate the Compaan/Laura approach [121].

6.2 M-JPEG Encoder and its PPN

An overview of the M-JPEG encoder workflow is given in Appendix C.1. The SANLP of the M-JPEG encoder used for experiments is given in Listing C.2. Running the code in Listing C.2 through the Compaan compiler results in the PPN shown in Figure C.2. The M-JPEG PPN automatically generated by the Compaan compiler is a four-node pipeline. The processes exchange data via tokens, which by default correspond to 8 x 8 pixel blocks. The four processes are connected by channels im-
6.3. EXPERIMENTAL SETUP

implemented as FIFO buffers. For each of the four processes, a task is generated in C. Following the typical PPN implementation and mapping approach, each PPN process is assigned for execution to a single asynchronous processing entity, implemented as a POSIX thread. Each POSIX thread is typically assigned for execution to a different core of the platform’s multicore CPU. The obtained parallel program exhibits task and pipeline parallelism. The code within the tasks is sequential. The whole PPN executes in asynchronous data-driven fashion.

6.3 Experimental Setup

6.3.1 Application Configuration

We performed experiments by running the M-JPEG PPN on a stream of 100 frames. Each frame is a color image of 128 × 128 pixels. In the given M-JPEG encoder implementation, the source image is given in YUV color space. Frames are split in 8 × 8-pixel blocks (1 KB each), which are processed independently. The baseline performance results are obtained by running the default PPN automatically generated by running the Compaan compiler on the M-JPEG SANLP. The experiments in the subsequent sections are designed to evaluate performance improvement over default PPN that are achieved by applying techniques from the previous three chapters.

6.3.2 Platform

The test platform used for all experiments in this chapter features an Intel Core i7-920 Nehalem architecture 2.66GHz processor, Intel Motherboard, and an NVIDIA Tesla C2050 GPU. The Tesla C2050 GPU has 448 streaming processors (SPs) organized in 14 streaming multiprocessors (SMPs) with 32 SP cores each. The Intel Core i7 (Nehalem) is a multi core, Hyper-threading technology (HT) enabled design [44]. Each socket supports one to eight cores, which share a last level cache (L3), a local integrated memory controller (IMC) and an Intel QuickPath Interconnect (QPI). The interface to the GPU is via PCIe 2.0 x16 bus. The microbenchmarks for the performance of the host memory subsystem and the host-GPU link are given in Section 5.5.2. The NVIDIA Tesla C2050 GPU is a second-generation GPU for general purpose computing featuring the Fermi architecture [105]. The Fermi architecture is the first CUDA-capable architecture with support for the task parallelism. In addition, the Tesla-line of GPUs contains two DMA engines, which are typically found only in high-end GPUs for high performance computing. As a result, Tesla C2050 GPU supports concurrent DMA transfers from and to the GPU over PCIe bus.

132
6.3.3 Experiments

To obtain the baseline, we measured the initial performance of the default PPN obtained by the COMPAN compiler and analyzed its results. Our observations on the performance of the task-parallel M-JPEG are given in Section 6.4.

First, we applied techniques for multi-level parallelization presented in Chapter 4 to obtain a multi-level M-JPEG, illustrated in Figure 4.2. At the top-level (Level1) of the program we again constructed a task-parallel program from a PPN, but with one significant difference. The tokens processed and exchanged between the processes in the new M-JPEG PPN have different granularity than the tokens in the default PPN. The tokens in the new M-JPEG PPN are not limited to 8 × 8 pixel blocks, but they can also represent entire frames. We show the impact of the token granularity adjustment in the PPN in Section 6.5.

Second, at the bottom-level (Level2) of the multi-level program model depicted in Figure 4.2, there is a node T which executes the DCT computation. We further parallelized this node to reveal its data parallelism. Using the techniques described in Chapter 3, we obtained a data parallel CUDA kernel for DCT. We first measured in isolation the computational performance of the default kernel resulting from the KPN2GPU, and then showed the improvements after including the optimizations proposed in Chapter 3. We then show the total speedup of the DCT processing on the GPU which is affected not only by the computation but also by the time to transfers the input data for the DCT to GPU and to transfer the results back. The overall results of DCT processing on Tesla C2050 GPU are given in Section 6.6.

Third, we measured the performance of the overall solution. The result of Chapter 4 is a multi-level parallel program that features task, data, and pipeline parallelism. The coarse-grain tasks are mapped onto platforms devices (CPUs, GPU). At the platform level (Level1), we exploit task and pipeline parallelism. At the component level (Level2), we explore data parallelism in the DCT node by executing its process iterations on Tesla C2050 GPU. To obtain the overall results, we measured the performance of the multi-level parallel M-JPEG using the stream buffer design with kernel offloading methods presented in Chapter 5. The results for the overall solution are given in Section 6.7.

6.4 The Performance of Default M-JPEG PPN

To evaluate the performance of the initial task-parallel program obtained using PPN model, we compared the throughput of the sequential M-JPEG encoder with the throughput of the default parallel M-JPEG encoder derived from the PPN model. The performance of the default PPN model can already be improved using a waste range of techniques, such as node splitting and merging [93], which were developed
in the Daedalus context at LERC [2]. In this chapter, we are interested to measure only the incremental improvement stemming from the techniques proposed in Chapters 3, 4, and 5, and for this purpose we use the default PPN derived by the Compaan compiler as the baseline.

We report the results as the average throughput in KB/s for the entire stream. The throughput measured for the sequential M-JPEG encoder amounts to 385 KB/s. The throughput measured for the automatically generated parallel program obtained from the M-JPEG PPN is 470 KB/s, which corresponds to an improvement of approximately 22% over the sequential version. To estimate the computational demands of M-JPEG encoder tasks, we profiled the sequential application and compared the execution time spent in each stage. The results are shown in Table 6.4.

<table>
<thead>
<tr>
<th>M-JPEG Task</th>
<th>vin</th>
<th>dct</th>
<th>q</th>
<th>vle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time [%]</td>
<td>2.50</td>
<td>48.50</td>
<td>25</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 6.1: Percentage of M-JPEG Execution Time spent in each task.

The analysis of M-JPEG performance lead to the following observations.

First, Table 6.4 shows that the DCT node is the bottleneck node of the M-JPEG pipeline. As the execution time of a bottleneck node determines the overall execution time of a pipeline application, the overall performance could be potentially improved by acceleration of the DCT node.

Second, the overall application performance in the parallel case is also affected by large amount of (physical) data movement operations involved in realizing the PPN channel-based communication.

Third, the default M-JPEG PPN uses small fixed-size tokens corresponding to 1 KB image blocks. The use of small tokens leads to inefficient data transfers as indicated by experiments conducted in Section 5.5.2 (Figure 5.8). In addition, small tokens require large amount of synchronization between nodes.

In the subsequent experiments, we show how the techniques proposed in the previous chapters alleviate these issues.

6.5 Adjusting Token Granularity by Encapsulation

A well known problem in parallel applications is the cost of communication and synchronization, as it can easily outweigh the benefits of parallelization [118]. The overall percentage of time spent in synchronization can be reduced if we could reduce the number of synchronization points. In the PPN model, the synchronization points are induced by blocking read and blocking write operations. The number of blocking operations can be reduced by reducing the number of process iterations that are ex-
6.5. ADJUSTING TOKEN GRANULARITY BY ENCAPSULATION

cuted in order to process the same amount of input data. We achieve this by packing blocks into larger composite tokens. This is realized by using the token composition concepts and techniques presented in Chapter 4.

Throughput (KB/s)

Token Size (KBs)

Figure 6.1: Effect of Token Granularity on Parallel Execution Time

The default M-JPEG PPN generated by the Compaan compiler communicates tokens of the size of a single image block between the processes. For example, in order to process a sequence of 10 frames, with each frame containing $8 \times 16$ blocks, each PPN process executes 1280 iterations. There are two synchronizations points in each process iteration (one for blocking on read and one for blocking on write). This results in 2560 synchronization during process execution. By coarsening the granularity of the composite token to token cardinality $TC = 128$ blocks, the number of synchronizations is cut by the factor of $MR$, i.e. only 200 synchronization points are required for the complete frame sequence.

The result of token granularity coarsening is shown in Figure 6.1. The horizontal axis represents token size as a multiple of 1 KB units, i.e. blocks. The mjpeg-ppn bars show the throughput of the task-parallel M-JPEG obtained by using the PPN model of computation in KB/s. The first bar corresponds to the reference performance of the default PPN generated by the Compaan compiler. The subsequent bars show how the PPN throughput changes with the change in token granularity. We
observed an improvement of the mjpeg-ppn execution time of 27% after we tuned
the token granularity size. The optimal token size for M-JPEG execution following
PPN model of computation on the given test platform is found to be 16 KB. As a
result of token granularity coarsening, the speedup over sequential version increased
from 22% for the initial PPN to 55% for the PPN with the adjusted token granularity.

6.6 Leverage Data Parallelism for GPU Acceleration

6.6.1 DCT Kernel Execution

Combining task and data parallelism has been considered in many areas of parallel
and distributed computing - from programming large scale distributed systems to
programming second-generation GPUs with support for concurrent kernel execution
[18] and Cell B.E. [133]. We pursue this idea by exploiting inherent data parallelism
in DCT node of the M-JPEG encoder. We first performed the hierarchical splitting of
M-JPEG. At Level1, we generate a PPN featuring task parallelism at selected token
granularity. The blocks in each composite token that comes into the DCT node are
offloaded for processing on GPU SMPs. To generate a CUDA kernel for the
DCT computation, we created a Level2 PPN from the code in mainDCT function (See
Appendix C, Listing C.2) Following the approach outlined in Chapter 3, we obtain
a pipeline of data parallel CUDA kernels. Each 8 × 8 block is processed by a thread
block consisting of 64 CUDA threads.

Let’s now evaluate the performance of the CUDA code obtained using the tech-
niques presented in Chapter 3. The support for composite tokens introduced in Sec-
tion 3.8.3 allows us to obtain a pipeline of 14 data parallel kernels (one kernel for
each loop nest in mainDCT function). Figure 6.2 shows the summary of the computa-
tional speedups achieved by executing DCT computations on the GPU. The horizontal
axis shows the number of composite tokens processed on the GPU as a multiple of
1 KB blocks. The vertical axis shows computational speedup as a function of GPU
execution time divided by CPU execution time for the same M-JPEG code.

The baseline performance of the data parallel DCT CUDA code produced in line with
Chapter 3 is shown as trace kpn2gpu-dp-def in Figure 6.2. It achieves a speedup of
only 3.5× compared to the sequential DCT code running on the CPU. This is primarily
due to the excessive global-memory communication and kernel launch overheads
caused by mapping each PPN node of the DCT computation pipeline onto a separate
kernel. To improve the performance we manually modified the code to include the
optimizations presented in Section 3.8:

- First, to avoid kernel launch overheads, we merged 14 kernels of the DCT com-
  putation pipeline into a single CUDA DCT kernel.
6.6. LEVERAGING DATA PARALLELISM FOR GPU ACCELERATION

Figure 6.2: Computational Speedup is given as a ratio of the time required for sequential processing of reference C code on a single CPU core vs the GPU processing time

- Second, we remapped the PPN channels connecting 14 nodes from GPU’s global to GPU shared memory.
- Third, we increased amount of data reuse by exploiting the multiplicity property discussed in Section 3.8.3 to load the input blocks only once into a shared memory buffer instead of reading it again from the global memory for each thread separately.

The resulting data parallel DCT implementation is shown in Figure 6.2 as the kpn2gpu-dp-opt trace. This optimized version is approximately 80× faster than its sequential source code executing on the CPU.

As the last optimization, we added task parallelism stemming from independent processing of different color channels (YUV). The results are presented as kpn2gpu-best-(dp+tp) trace, which reaches a speedup of 87× for 16234 KB input data size. The hand-optimized DCT [101] in the CUDA SDK achieves a 102× speedup over its sequential counterpart executed on the same CPU.

The CUDA DCT performance increase from only 3.5× to 87× speedup has been achieved as a combination of reducing synchronization costs by merging PPN nodes
6.6. LEVERAGING DATA PARALLELISM FOR GPU ACCELERATION

in the pipeline, reducing communication costs by smarter mapping of PPN channels to the rich memory hierarchy of the GPU, and taking advantage of data reuse in the kernel by exploiting the multiplicity property for improving data locality. The tremendous performance jump achieved by leveraging the concept of multiplicity in the PPN specification points to the relevance of further work on directions outlined in Section 3.8.3.

6.6.2 Offloading Computation to the GPU

![GPU Performance Breakdown](image)

Figure 6.3: Breakdown of GPU time: Time to compute the DCT kernel $\text{comm}$ vs. the time spent in transferring the input and output data to the host ($\text{comp}$).

The computational speedup is only a single side of the performance coin. To describe the effect of GPU acceleration on the overall M-JPEG application performance, it is also necessary to consider communication and runtime overheads. Figure 6.3 depicts normalized GPU time versus data size. GPU time in this section denotes the total time required to send the data to the GPU, compute it on the GPU, and send it back to host. The GPU time is measured on the GPU using the NVIDIA’s CUDA profiler tool. Each of the bar shows the ratio of time spent in communication...
6.6. LEVERAGING DATA PARALLELISM FOR GPU ACCELERATION

(data transfers from host to GPU, and from GPU to host) and time spent in computation on the GPU, i.e. kernel execution time. For small number of elementary tokens (1KB blocks), the time to transfer the data is still comparable to the computation time. However, as the utilization of the GPU increases, the computation throughput grows. The time to transfer the data becomes significantly larger than the actual GPU computation time. For larger data sizes (higher token cardinality (TC) factor), communication takes up more than 80% of the GPU time. This points to the significance of (a) being able to adjust the token granularity, (b) overlap the transfers. The significance of adjusting token granularity was already shown in Section 6.5. Before we discuss the transfer overlapping for which we introduced the concepts in Chapter 5, let us first see what is the overall performance gain achieved by DCT acceleration on the GPU when data transfers are taken into account. Figure 6.4 shows a comparison of the speedups achieved by measuring the following times: (a) kernel computation time on the GPU (denoted as $\text{GPUKernel(s)}$), (b) kernel computation time and (synchronous) data transfers to the GPU (host-to-device and device-to host) (denoted as $\text{GPUKernel(s) + Transfers}$), (c) kernel computation time, data transfers (denoted as $\text{Complete Offload}$).

Figure 6.4: Comparison of speedup achieved with offloading DCT transform onto GPU: (a) computational speedup, (b) gpu (comp+comm) speedup, (c) complete offloading speedup including runtime overheads.

139
6.7. OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION

as CompleteOffload), which additionally accounts for CUDA runtime overheads. The results for variants (a) and (b) are obtained using CUDA profiler to record timestamps on the GPU, while the results for variant (c) are measured using a CPU-based timer which was started before the start of synchronous offload and stopped after the synchronous offload has returned control to the calling thread. The vertical axis shows speedup, as the reminder of the CPU execution time for sequentially executed DCT source code with the GPU time for three different measurement scenarios described above. The horizontal axis shows data size as the number of 1KB blocks sent to a GPU for processing. So far, we found out that the best KPN2GPU variant of mainDCT achieves more than 12 GB/s computation throughput on the GPU. Compared to the throughput of the original DCT code which is 180 MB/s, this results in approximately 87 x computational speedup. However, when data transfers and runtime overheads are taken into account, the overall performance numbers drop since extensive amount of time goes on transferring the data to and from the GPU. When communication to/from the GPU is included, the best performing code achieved slightly more than 2 GB/s throughput. This corresponds to approximately 14 x speedup with data transfers. Finally, when all run time overheads for synchronous offloading are taken into account, We found out that for < 128 blocks, overheads of offloading to the GPU cause a slow-down instead of speed-up. The maximal achieved speedup with synchronous kernel offloading is reduced to 13.76 x for large data sizes. There is also a further performance drop for complete offloading of small data sets. However, we believe that Figure 6.4 may be giving an overly pesimistic result for the complete offload performance, since the execution time for smaller data sizes lasts shorter and the measurement may be affected by timer precision. For the same reason, the recorded performance may also suffer from synchronization required on the CPU side to complete the measurement. Detailed time breakdown and investigation of overheads affecting offload performance within a complete application would definitely require more sophisticated performance analysis tools for hybrid application than the tools publicly available today. In any case, the results presented above point out the importance of efficient offloading and efficient communication with the GPU, and the benefit of leveraging asynchronous transfers for overlapping communication and computation whenever possible.

6.7 Overall Performance Results with GPU Acceleration

In this section, we analyze the overall performance obtained by generating and mapping a two-level parallel program using the methods described in previous sections. Partial results for token granularity and GPU offloading were given in the previous sections. In this section, we show these results in the context of complete M-JPEG
6.7. OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION

PPN execution with task, data, and pipeline parallelism on the streaming platform.

![Hybrid Parallelization with GPU Acceleration of DCT Node](image)

Figure 6.5: Hybrid PPN Parallelization with Data Parallel DCT (Converted with KPn2GPU) and Host-Accelerator Stream Buffers as a Function of Token Size.

Figure 6.5 depicts PPN application throughput as a function of token size in the top-layer PPN. The bar chart gives performance comparison of the following PPN variants:

- `mjpeg-ppn`
- `mjpeg-ppn-kpn2gpu-sok`
- `mjpeg-ppn-kpn2gpu-aok`

The first variant `mjpeg-ppn` represents the initial PPN obtained by the Compaan compiler, converted into a multi-threaded program, and executed on the multicore CPU. The second variant `mjpeg-ppn-kpn2gpu-sok` is an extension of the initial PPN with synchronous offloading of parallelized DCT to the GPU. The third variant `mjpeg-ppn-kpn2gpu-aok` is a variation of `mjpeg-ppn-kpn2gpu-sok`, which uses the principles presented in Chapter 5 for asynchronous communication and asynchronous DCT kernel offloading.

141
6.7. OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION

For all variants, the low point occurs for the default PPN token size which corresponds to a single block. For 1 KB tokens, the throughput of \texttt{mjpeg-ppn} is 470 KB/s, while the throughputs of \texttt{mjpeg-ppn-kpn2gpu-sok} and \texttt{mjpeg-ppn-kpn2gpu-aok} are even below 100 KB/s. This dramatic performance drop is due to the inefficient processing on the GPU when the whole workload contains only one single block. In this case, there is simply not enough work to utilize multiple SMPs on the GPU. All 14 SMPs on the Tesla C2050 are idle apart from a single SMP. In addition, the kernel offloading overheads for small token data sizes are large in comparison to the duration of the token processing as shown in Section 6.6.2.

The two observed problems can be alleviated by processing composite tokens, which have more than 1 block. In Section 6.5 we demonstrated the performance improvements achieved on the multi-core CPU by tuning the PPN token size. The use of TC tokens is however not only beneficial for reducing the amount of synchronization in PPN, but it also allows us to achieve better utilization of GPU SMPs. Figure 6.5 clearly shows a considerable performance leap that occurs with increase of the token granularity for all three variants. The performance increase is much larger for PPN variants with GPU acceleration of DCT, as the efficiency of GPU processing increases with the number of independent work items. With composite token tokens it is possible to occupy multiple SMPs, and thus achieve much higher GPU accelerator utilization. For example, with token cardinality factor $TC = 128$, 128 blocks are processed on the GPU in a single offload. As a result of better GPU utilization and smaller runtime overheads, the overall performance of the parallel M-JPEG with GPU acceleration increases from 480 KB/s to 1826 KB/s.

Use of the AOK mechanism brings an improvement of 5-28% to the overall performance depending on the data size. Contrary to our expectations, the performance improvement in this case is not primarily due to asynchronous data transfers and thus streaming to/from the GPU. In depth investigation of GPU figures within CUDA GPU profiler tool revealed that there are only a few overlapping data transfers between host and GPU accelerator, since the best M-JPEG PPN data rate is only around 460 KB/s. This is much smaller than the transfer rate of the PCIe link connecting the GPU to the host, which is around 4 GB/s. For any overlaps to be possible, the data rate of the application must be much higher, preferably closely matching or larger of the link data rate. Otherwise, there are large gaps between the GPU transfers, and there is nothing to be overlapped. The primary performance impact of AOK in this case comes from the reduction of communicated data volume achieved by using the more efficient PPN channel design which reduces the pressure on the memory subsystem.

The most significant part of performance improvement comes from the speed-up gained by data-parallel processing of mainDCT node on the GPU. Detailed performance/time breakdown is illustrated for three different token sizes for the first variant \texttt{mjpeg-ppn} in Figure 6.6 and for the second variant \texttt{mjpeg-ppn-kpn2gpu-sok} in
6.7. **OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION**

Figure 6.6: Execution Time Breakdown for Parallel PPN Execution of M-JPEG Tasks (mjpeg-ppn). Each task executes a five-phase sequence: block on read, read data transfer, computation, block on write, write data transfer.

Figure 6.7. The detailed performance breakdown for the third variant mjpeg-ppn-kpn2gpu-aok is left out due to technical hurdles involved in detailed performance benchmarking when using asynchronous data transfers and execution on the GPU.

For token size 128 KB (stemming from the token cardinality factor $TC = 128$), the DCT task amounts to 60.3% of total computation time, as illustrated by the second stacked histogram in Figure 6.6. By offloading the DCT task to the GPU and processing it in a data parallel manner, the compute phase of the DCT task with GPU offloading is reduced to 22% of the total execution time. The performance breakdown with DCT execution on the GPU is illustrated by the second stacked histogram in Figure 6.7.

After GPU acceleration of DCT, the new bottleneck task is VLE with 37.6% computation time. Since the new bottleneck is the VLE task, further work on the optimization of the DCT computation on the GPU would have a small impact on the overall execution time. The new bottleneck task VLE is difficult to parallelize for GPU, due to its computational structure. While in [13], we presented a novel approach for parallel variable-length encoding (PAVLE) on the GPU, this parallelization approach requires use of a completely different algorithm and is a non-trivial task for a compiler. Some
6.8. CONCLUSIONS

Figure 6.7: Execution Time Breakdown for Parallel PPN Execution of M-JPEG Tasks with DCT Task Offloading to a GPU Accelerator ($mjpeg$-$ppn$-$kpn2gpu$-$sok$). Compute bar for DCT task represents complete GPU offloading time. The offloading time is composed of data transfers to/from GPU, sub-PPN computation on a GPU, and CUDA runtime API overheads.

Further gains are achieved with the use of stream buffer design since it efficiently reduces pressure on the memory subsystem due to its improved data access protocol and thus improves the overall application performance.

6.8 Conclusions

In this chapter, we performed a case study on multi-level parallelization on the example of M-JPEG encoder. Using concepts and techniques presented in previous chapters, we generated a top-level PPN featuring task and pipeline parallelism with adjusted token granularity. Parallelization of the main DCT functionality using the KPN2GPU tool revealed inherent data parallelism within the DCT node. Finally, we mapped the two-level PPN onto a heterogeneous platform with a modern Tesla C2050 GPU accelerator. We exploited both data and task parallelism within the DCT node for efficient mapping onto second-generation Fermi-architecture architecture GPU, and
6.8. CONCLUSIONS

used the stream buffer design to improve efficiency of PPN communication channels. After transformations with the techniques presented in the previous chapters, the resulting multi-level program performs $4 \times$ faster compared to the default task-parallel PPN obtained by the Compaan compiler.
6.8. CONCLUSIONS