On the Use of Small 2D Convolutions on GPUs

Shams A.H. Al Umairy[1], Alexander S. van Amesfoort[1],
Irwan D. Setija[2], Martijn C. van Beurden[3], and Henk J. Sips[1]

[1] Delft University of Technology, Delft, The Netherlands
[2] ASML, Eindhoven, The Netherlands
[3] Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. Computing many small 2D convolutions using FFTs is a basis for a large number of applications in many domains in science and engineering, among them electromagnetic diffraction modeling in physics. The GPU architecture seems to be a suitable architecture to accelerate these convolutions, but reaching high application performance requires substantial development time and non-portable optimizations. In this work, we present the techniques, performance results and considerations to accelerate small 2D convolutions using CUDA, and compare performance to a multi-threaded CPU implementation. To improve programmability and performance of applications that make heavy use of small convolutions, we argue that two improvements to software and hardware are needed: FFT libraries must be extended with a single convolution function and communication bandwidth between CPU and GPU needs to be drastically improved.

1 Introduction

Convolution on multi-dimensional periodic data is frequently employed in many applications. A few examples of convolution operations and their application areas are computing a weighted, moving average in statistics, (linear) time-invariant systems in signal processing, and in physics where many systems are modeled as linear systems. A typical physics system where convolution plays a role is the modeling of wave properties. If the wave equation is linear, which is very often (assumed to be) the case, the superposition principle and the convolution operator can be applied. More specifically, convolution is used as a processing step in many electromagnetics (EM) problems, where photon transport methods are used to derive the optical broad-beam responses from scattering/diffracting objects. A standard practical problem is modeling scattering by a perfectly conducting plate that has been meshed uniformly (discretized).

Several real-world EM problems involving scattering and radiation can not be solved analytically, because of the irregular geometry of the structures of interest. There is ample interest in accurate, numerical EM models in research, and also in the industry to integrate advanced, real-time analysis of electronic circuits or of other nano-scale objects into manufacturing equipment. But the computational demands for solving sets of linear equations numerically often imposes limits on precision, computation and memory usage, and recently, power/heat.

A.L. Varbanescu et al. (Eds.): ISCA 2010 Workshops, LNCS 6161, pp. 52–64, 2011.

Modern multi-core architectures promise to push away those boundaries. Hence, (massive) parallelization is an interesting and active research area.

One platform that could satisfy the hunger for a huge amount of floating point computations is the graphics processing unit (GPU). Computer graphics hardware has been rapidly increasing in performance, and is suitable for general-purpose computations using CUDA and OpenCL since a few years. However, the GPU performs most efficiently when there are lots of independent jobs with optimal size. For FFT-based convolutions, this means many FFTs that just fit in core-local memory, which means 1024–4096 points. Especially the spectral methods lead to small FFT sizes as they exhibit fast (exponential) convergence for analytical data. In addition, for signal/image processing, small FFT sizes are preferred and typically 9×9 pixels are used as a basis. Otherwise, it often depends on physical properties like structure size versus wavelength. If this ratio is small, we can expect small FFTs.

In this paper, we consider the 2D convolution operation in diffraction grating of multi-dimensional EM wave and interference pattern algorithms. We show the effectiveness of our parallelization techniques of 2D convolution using small FFTs on GPUs and the CPU. We demonstrate that even for small 2D convolutions using FFTs, good performance can be attained on GPUs, but at the cost of substantial programming effort. Afterwards, we will discuss other optimizations and more generic solutions.

In the remainder of this paper, we describe our specific application example in Section 2, followed by a brief overview of the CUDA GPU and programming architecture in Section 3. Next, Section 4 discusses the parallel implementation and explores the most important optimizations techniques for our application type on the GPU architecture. In Section 5, our performance measurements are shown as well as a comparison to a multi-threaded CPU implementation, followed by a discussion on the implications of the experiments in Section 6. We wrap up with the conclusion in Section 7.

2 Electromagnetic Diffraction at Nano-structures

In physics, waves have the property of diffraction, where an incoming wave that hits a single-slit or grating object is split into several diffracted beams. The reflected light is a convolution of the patterns from diffraction and interference. The angles of diffraction are determined by the (ratio between) slit width and wave length. Electromagnetic (EM) diffraction techniques are commonly used to model light propagation on a scattering medium with a varying refractive index. The EM response of the scattering object to the incident field is described by a shape function that models physical quantities such as absorption and reflectance. With a known incident field, the measured reflected field can be used to deduce the shape of very small grating structures, such as those placed on a silicon substrate. The actually produced shapes can be compared to the intended shapes to calibrate the manufacturing process to produce incredibly small scale and high density electronic circuits at high yields.

The diffracted light beam $C(x, y, z)$ from a plane positioned perpendicular to the z-axis can be computed as shown in the 2D convolution Equation 1. A light source with beam profile $E(x, y)$ (represented as an $N \times N$ matrix) is convolved with the shape function $S(x, y, z)$ on that plain (represented as an $M \times M$ matrix), that models the behavior of a beam on a grating. Normally, M is greater than N.

$$C(x, y, z) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} S(x - x', y - y', z) E(x', y') dx' dy' \tag{1}$$

$$C(m_1, m_2, z) = \sum_{m_1=-\infty}^{m_1=\infty} \sum_{m_2=-\infty}^{m_2=\infty} S(m_1 - m_1', m_2 - m_2', z) E(m_1', m_2') \tag{2}$$

To retrieve its shape numerically, the space in terms of grids in (x, y, z) has to be discretized through a fourier expansion of the fields in the periodic x- and y-direction and through slicing the object in layers along the z-direction. We can solve the convolution at each point in the grid (Equation 2), by rewriting the integral over (x, y) to a summation over the lateral fourier modes $(m1, m2)$. The most common fast convolution algorithms apply the fast fourier transform (FFT) via the circular convolution theorem. Specifically, the circular convolution of two finite-length sequences is found by performing an FFT, multiplying point-wise, and performing an inverse FFT. To apply this convolution, the input sequences must often be zero-extended (padded) and/or portions of the output sequences discarded. Thereby, a 2D fourier-based convolution is computed in five steps:

1. Pad the EM vector fields.
2. Apply 2D FFT to 2D arrays on a layered stack of slices ((x, y)-planes).
3. Perform element-wise multiplication with the shape function.
4. Apply inverse 2D FFT.
5. Extract the scattered vector fields.

To increase the accuracy of computing the angular-resolved spectrum, the diffraction grating can be lit from multiple, independent angles of incidence. These angles θ_i are independent and can be processed in parallel.

All gratings have intensity maxima (m) at angles θ_m, where m is an integer specifying the order of the diffracted beam, resulting in diffracted orders on both sides of the zero-order beam. These parameters can be varied and influence the size of the problem, such as the number of harmonics in the x- and y-directions, the size of the FFTs, the number of layers in the z-direction, and the number of angles of incidence. We concentrate on small FFT sizes, which we believe are undervalued, while highly relevant for the described application types in general, and in particular, EM diffraction modeling. Convolutions referred to typically contain an FFT size between 16×16 and 128×128 and take 60–80% of the total runtime.

The computational complexity of the fourier-based convolution method depends largely on the size of the EM vector field (N). In convolutions using this method, the solution matrix is of the size $(M + N - 1) \times (M + N - 1)$. The major

computational steps for each of these elements (except those near boundaries) include FFT and IFFT of $(M+N-1)\times(M+N-1)$ matrices, so the complexity is $O[(M+N)^2 log(M+N)]$. Large scale computational EM model problems for small (x, y)-planes have computational limitations derived from soft real-time requirements. Thus, to speedup this numerical analysis, parallelization of these computations is of interest. The next section will outline the most important features of our target platform, the NVIDIA CUDA GPU platform.

3 NVIDIA CUDA GPU Platform

As graphics processing units (GPUs) have become more powerful, each generation has focused more on general-purpose processing (GPGPU). Since 2007, NVIDIA distributes CUDA (Compute Unified Device Architecture), a hardware/software platform to make developing highly parallel GPGPU applications more straightforward. A GPGPU consists of a set of multi-processors, caches, an interconnection network, and memory controllers, connected to off-chip, high bandwidth ("global") memory. Each multi-processor has a flexible vector unit, a large scalar register set, local scratch ("shared") memory, and may read through or around various caches. A multi-processor executes vector instructions ("warps") in a SIMT fashion, a variant of SIMD where vector components are considered as individual threads that can branch independently and have their own address generator. A warp is currently 32 threads, which is wider than the vector unit. Many warps must be active concurrently per multi-processor to hide execution stalls.

The CUDA programming model encourages programmers to partition the work into independent sub-problems (blocks) that can be solved in parallel, and then into finer pieces (threads) that can be solved cooperatively in parallel. This hierarchy of threads fits well on modern multi-core systems and encourages mainly spatial locality of reference in global memory, few diverging branches, and usage of shared memory within blocks of threads. At kernel launch time, an execution configuration must be provided, which specifies the number of blocks and its size in up to three dimensions. A sane block size ranges from 32 to 256 threads, while many hundreds or thousands of blocks should be launched to scale transparently to any number of multi-processors. Together with the register and shared memory requirements, the number of blocks that can run concurrently per multi-processor can be determined by the hardware scheduler. Data transfers between main memory and GPU global memory must be managed explicitly. Since the PCIe bus is relatively slow, this I/O is often costly. The easiest way to get started with programming CUDA GPUs is using the CUDA programming guide [3] and optimization manual [2] from NVIDIA.

The basic software toolchain of CUDA consists of a compiler, driver, and runtime libraries. NVIDIA also provides a basic profiler, GDB debugging support, and several libraries of commonly used functionality, among them, the CUDA FFT (CUFFT) library. CUFFT provides a FFTW library-like interface for computing FFTs in parallel on CUDA GPUs. A problem is that each CUFFT

function is a black box that will launch a GPU kernel, so we cannot easily modify or compose with it. The CUDA SDK contains an image convolution example [5] and describes FFT based convolution [4], but does not nearly go as far as this study. We selected the GPU, because we expected that an architecture with small, core-local memories and many cores would be the best fit for many, small convolutions. Nevertheless, the following aspects must to be taken into account to benefit from the GPU.

1. Maximize the amount of independent work. (Section 4.2)
2. Tune the execution configuration. (Section 4.3)
3. Minimize the transfers between CPU and GPU memory. (Section 4.5)

4 Parallel Implementation on the GPU

After an initial CUDA implementation, we first explain how we exposed more independent work and with which execution configuration, then we optimize within the convolution, and lastly we try to reduce CPU/GPU data transfers.

4.1 Initial CUDA Implementation

The first parallel version of a fourier-based 2D convolution algorithm can be written in CUDA in a fairly straightforward way. The data-independent sections of the program are identified, implemented as kernels and mapped to the GPU. The input and output of the kernels are data arrays transferred and stored as arrays in GPU memory. Our initial GPU algorithm executes the following steps, traversing sequentially through the stack of slices and through the angles of incidence.

1. Pad each signal stream with zeros.
2. Copy each data stream and the shape function to GPU memory.
3. Set up the CUDA execution configuration.
4. Invoke CUDA kernel to apply forward 2D FFT.
5. Invoke CUDA kernel to multiply element-wise with the shape function.
6. Invoke CUDA kernel to apply backward 2D FFT.
7. Copy the resulting arrays back from GPU memory.
8. Extract each signal stream from resulting arrays.

There are two layers of parallelism to work on: the nested loops and the 2D convolution itself. The nested loops over all layers (and three components x, y, z) can be batched into a single kernel, such that each loop iteration is executed in parallel on its own CUDA thread(s). We also turn to the convolution itself, since it is not very efficient by default, especially without batched 2D FFT, and because not all convolution applications using small FFTs have enough independent data streams to satisfy the GPU. It can be split into the forward and backwards FFT tasks and into the multiplication. Since the 2D CUFFT consumes the majority of cycles, we dived deeper and split the 2D CUFFT into 1D CUFFT and transpose sub-tasks.

4.2 Increasing Independent Work

Parallelizing the computations of each slice separately performs poorly, because there is too little work per kernel invocation. The easiest way to improve is to move loops into the GPU kernels (steps 4, 5, and 6) and parallelize them. We can execute all layer (and field component) iterations in a single kernel (as long as data fits in GPU memory), such that the computations in each kernel are applied to all slices in parallel. As a result, more threads run per kernel invocation and concurrently. This can also be applied to the angle loop to convolve the data for all angles of incidence in parallel. Such code transformations look trivial, but the generation and outline of the data structures in our sequential application was unsuitable to parallelize for multiple angles in the same, initial effort as for multiple layers. Data for multiple angles must be available at the same time and preferably stored in a single, contiguous buffer. Unfortunately, such parallelization obstacles in sequential code are common, because the extra requirements from parallel programming are unnatural for sequential code, memory requirements are higher, and even if the operations are independent, the output locations may not be if output is of variable length (not in our case).

4.3 Tuning the Execution Configuration

The execution configuration can be specified using up to three dimensions, however, CUDA imposes some limits on the size of each dimension: $(65535, 65535, 64)$. One can try to unroll a loop with independent iterations onto the GPU, but it is too restrictive to map the loop index of many small 2D FFTs or matrix computations onto the (only free) third dimension. The block and grid dimensions must be chosen to run an optimal number of threads, preferably as a multiple of the warp size. More threads per block can lead to better shared memory usage (locality), but worse concurrency as local barriers operate per block. For the transpose and multiply kernels, we set up a grid of $(3\times(NLayers+1))$ matrices of size $Height\times Width$, for N angles; where each grid is structured as a square 2D array of square threads blocks (i.e. 16×16 threads). This is done by distributing the threads horizontally and vertically, where a square array of blocks is repeated horizontally as to represent the number of matrices and vertically to represent the number of angles. As a general configuration, one can use a block size of $(16, 16, 1)$, in blocks of $((FFT\_Width/16) * (3 * (NLayers + 1), (FFT\_Height/16) * NAngles)$ with this pattern. We can apply this idea as long as the transform size is a multiple of 256, otherwise a smaller pattern (e.g. of size 64) should be set up. This strategy enables a great flexibility in the implementation of such algorithm with variety of geometry sizes and number of angles of incidence. As the number of layers along the z-direction and the number of angles in the spectrum increases, more elements can be explored in parallel, which may yield an increase in performance especially with small convolution sizes.

4.4 Optimizing the 2D Convolution

At this moment, the FFTs take a lot of time, so we turn to the convolution itself. To perform 2D FFTs on a batch of matrices on the GPU concurrently, we constructed our own batched 2D FFT on top of the batched 1D CUFFT functionality. Up till CUDA 3.0 final released in March, CUFFT supported batched execution for 1D FFTs only, but with the release of CUDA 3.0 final in March, 2D FFTs can also be batched, gaining the same speedup with this optimization as our code. The required steps are shown in Figure 1. Computing the 2D FFT

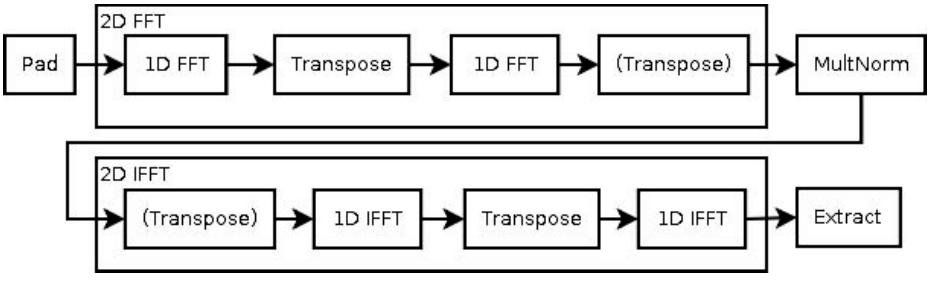

Fig. 1. 2D convolution using FFTs

is then a matter of applying the 1D FFT to every row and then to every column. Transferring matrix data to/from global memory column-wise is very inefficient. It is better to transpose twice: once to apply the column-wise FFT in row-wise order, and once to transpose back the output. All row-wise 1D FFTs can be computed in parallel, four times per 2D convolution, intermixed with transposes and the element-wise multiplication. Loading from and storing into shared memory is done in row-wise order. This transformation delivers a factor of 2–3 performance compared to an unbatched solution. We also merge the normalization for all FFTs into the multiplication kernel, but this computation takes only a fraction of the total time, and could just as well be multiplied into the shape vector when it is generated. Since the multiplication is element-wise, this algorithm could be optimized further by combining both kernels, which saves some memory transfers and kernel startup overhead. This was not implemented, since it requires creating our own batched 1D FFT. A high-performance evaluation of 1D and 2D discrete fourier transforms on CUDA GPUs was released over a year ago [1]. Comparing their performance results for small 1D FFTs with performance from CUFFT indicates that for FFTs with a power-of-two size up to 64 points, CUFFT can still be improved by up to a factor of three. We did not provide a custom 1D FFT, because we feel that this is exactly what CUFFT is for. With opening up 2D FFTs the situation is different, as there we can strip the inner transposes, so while both batched 1D and 2D CUFFT kernels can be improved for small FFTs, as soon as transposes take a significant time, it pays off to open up the 2D FFT.

4.5 Optimizing Transfers between CPU and GPU

A batched 2D solution also enables other optimizations that save work and data transfers. Since the forward 2D FFT is followed by an inverse 2D FFT afterwards, we can strip out the inner two transposes. The initial implementation pads and extracts on the CPU. Padding and extraction on the GPU is faster, not because of any speedup on these very short operations, but because it reduces the amount of data copied between CPU and GPU memory. (Avoiding an increase in GPU memory footprint takes a bit of care.) Apart from avoiding I/O and memory transfers altogether, we can optimize the remaining transfers. A related effect of the loop unrolling/batching is that we can copy fewer, larger buffers between CPU and GPU memory which is more efficient. To allow this, the input data structure must be rearranged into a continuous sequence of slices. In general, it is a good idea to move data inflating and/or reduction operations to the GPU if they are placed around the computations that need to be accelerated, even if those operations themselves take only a small fraction of the total run-time. Naturally, we use pinned memory to reach peak I/O bandwidth.

5 Experiments and Results

This section describes the experiments and test setup, and shows the performance of our multi-core CPU and GPU implementations.

5.1 Experimental Setup

Table 1 shows a range of different problem cases that have been approached in this study. The listed number of convolutions per angle is computed as $3\times(nLayers + 1)$. In all cases, we run through many grating iterations to ensure stable measurements, each on eight angles of incidence (each on the three field components). Beyond eight angles, the (GPU) platforms will not run more efficiently, as experimentally derived. Even more independent work cannot be processed in parallel. Table 2 shows the characteristics of the platforms used in this study. The Geforce 8800 GTX is the principal GPU from the "8 series", the first NVIDIA architecture supporting CUDA. The subsequent "200 series" architecture is represented by the Tesla C1060 and Geforce GTX 280. Tesla C1060 is equipped with more memory running at a lower (safer) clock frequency (0.8 GHz GDDR3) than the GTX 280 (1.1 GHz GDDR3) and is intended for

Table 1. Properties of considered problem cases

Problem Case	#Harmonics in X and Y	FFT Size	#Layers	#Convolutions/Angle
1	-3 to 3	16×16	32	99
2	-7 to 7	32×32	64	195
3	-15 to 15	64×64	128	387
4	-31 to 31	128×128	256	771

Table 2. Hardware platform characteristics

Processor	Cores	Freq(GHz)	Memory/Core	Memory	Gflop/s	Mem Bw(GB/s)
Intel Core i7 920	4	2.66	32+32 L1 256 L2 1x8192 L3	6 GB	85.2	32.0
NV GF 8800 GTX	16	1.35	16+8+8	768 MB	345.6	86.4
NV Tesla C1060	30	1.30	16+8+8	4 GB	936.0	102.0
NV GF GTX 280	30	1.30	16+8+8	1 GB	936.0	141.7

scientific/industrial computing instead of 3D gaming. While NVIDIA markets its GPUs as having hundreds of (CUDA) cores, in our opinion it makes more sense in architecture comparisons to count a multi-processor as one (albeit more flexible) SIMD core. The listed core clock frequency for the GPUs applies to the multi-processors. The chip core clock driving caching and interconnection subsystems runs at less than half of that frequency. A similar remark applies to our core i7, where the "uncore" clock drives the L3 cache and memory controllers at 2.133 GHz. The GPU platforms use the listed CPU platform as host system with 6 GB of DDR3-1333 memory. The Core i7 GPU features hyper-threading to run up to two threads per core and turbo boost to dynamically increase the clock frequency when the thermal situation allows it, often more on workloads that use few cores. For the Core i7 920, this means that the core clock can be increased once with 133 MHz and, if only one core is in use, it can be increased twice, leading to a 10% increase. This should be considered when interpreting the multi-threaded speedups.

As for the software setup, we ran our tests on Linux with a 2.6.31 x86-64 kernel and compiled our programs with GNU GCC 4.3.4. The multi-core CPU code uses the FFTW library, version 3.2.1, with SIMD and multi-threading support. The element-wise multiplication executes sequentially on the CPU. All tests have been run in complex single precision and produce the same output. To acquire stable performance measurements, we increase experiment duration by computing (including data transfers) multiple rounds sequentially.

5.2 Performance Measurements

The raw performance results of small 2D convolutions on the introduced platforms are shown in Figure 2. This does not include data transfers between CPU and GPU, as we compare pure 2D convolution processing here. It does imply that real implementations likely have to move more parts of the application onto the GPU, even if insignificant in terms of runtime, to avoid high transfer overhead. For problem case 1, the 8800 GTX GPU is already 50% faster than the CPU, and for larger cases, this becomes a factor of two. The GTX 280 GPU extends that to a factor of two for case 1, and to a factor of three for larger cases. It is a bit faster than the Tesla C1060 for case 1, and this difference increases to about 33% for case 4. Being mostly memory-bound, all experiments

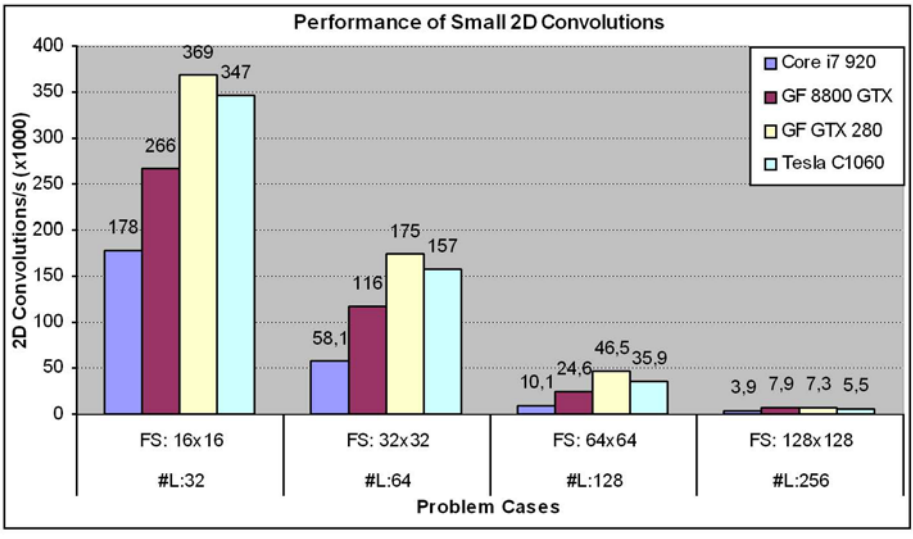

Fig. 2. Performance of small 2D convolutions on CPU and GPUs

under-use the computational capabilities of the platforms, but this is even more true for the smaller problem cases. There are some relative performance differences between different problem cases per platform, but they are not that large. This means that all platforms benefit/suffer by about the same factor from the efficiency problems inherent in the FFT computations and in the memory-bound portions, although we readily admit that the number of experiments is somewhat small for a broad statement here.

Although not plotted, we can report that the speedup of our multi-threaded CPU implementation over sequential execution using the FFTW library is worthwhile, but not very high. For the smallest problem case, four threads execute only 34% faster, and another four extra threads add another 5%. For problem case 2, the speedup is a bit higher. For the largest problem cases, the speedup approaches a factor of two using eight threads. The speedup of four threads is about half that, which indicates that hyper-threading, and multi-threading in general, make a real difference at medium-small 2D convolutions, but with a huge caution for poor scalability.

Using the CUDA profiler, we continuously measured the relative processing time of each GPU kernel and transfer during optimization. Figure 3 shows the final performance breakdown on the Tesla C1060. It indicates that for the larger cases, the transposes still take a lot of time, even after optimizing out the inner transposes. Although not shown, significant time is also spent in copying data between CPU and GPU memory. Data padding takes more time than extraction, because padding also clears memory areas, while data extraction simply skips those memory blocks. This difference is getting larger when increasing the problem size. The runtime fraction of the 1D FFT kernel is only large for

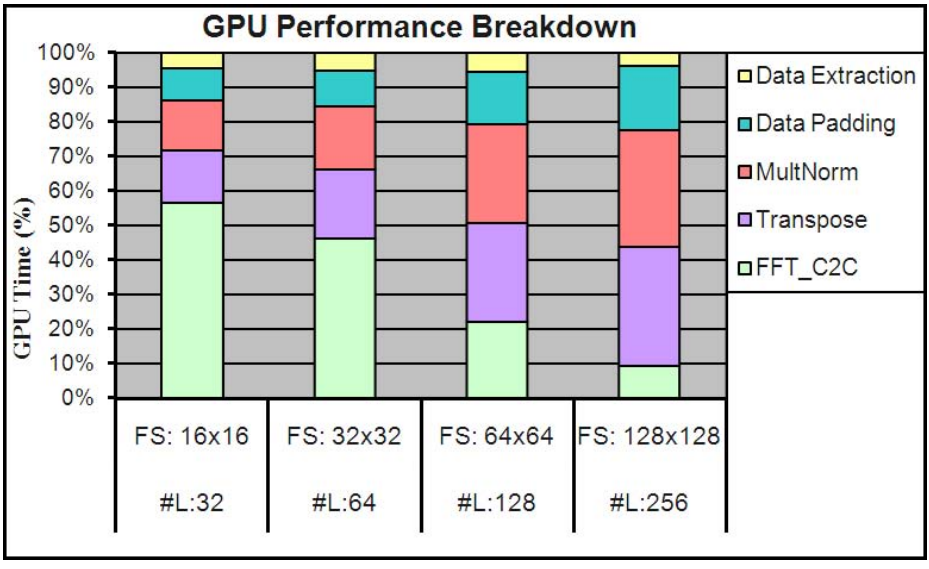

Fig. 3. Performance breakdown of small 2D convolutions on Tesla C1060

problem cases 1 and 2. Better CUFFT 1D performance would certainly help for small convolutions. Since many of the kernels, namely transposes, multiplication/normalization, and padding/extraction only load and store data with little or no computations in between, our last observation is that the best way to optimize further would be to merge kernels together, rather than to optimize the task that takes the most time.

6 Discussion

There are a number of properties inherent to computing large numbers of small (2D) convolutions. The large amount of independent work can be solved with fine-grain cooperative plus coarse-grain independent, massive multi-threading, which is exactly what the GPU needs. Also, there are no main memory alignment or transfer size issues (we consider adding transpositions to turn column- into row-wise FFTs as a given). However, the number of compute operations per data byte ("arithmetic intensity") is low. Batching even more work does not help, because you cannot fill one bandwidth hole with another. After transforming the application to submit a large volume of work at a time to the GPU, most optimization effort is aimed to get rid of unnecessary memory and I/O transfers, either by optimizing them out completely, or by moving more CPU or GPU kernels into fewer GPU kernels. This requires opening up FFT libraries like CUFFT and FFTW. The problem is that this is not productive for application writers and contrary to the solution that these libraries are offering (or supposed to offer). The introduction of batched 2D FFT support to CUFFT improves

the situation, but still leaves much to be desired for convolution processing. The suboptimal performance of small 1D FFTs in CUFFT is indeed causing performance problems for convolutions with a size up to 64 points.

Ideally, FFT operations could be programmed at a higher level, such that composing FFT kernels with other operations is easy and efficient. However, we do not see a high-performance solution of that kind coming any time soon. Instead, we propose to extend the commonly used FFT API to make the convolution a first class citizen. We believe that the two principal requirements for this have been met. The extension must:

1. be simple with many applications that can take advantage of it.
2. allow enough performance improvement to warrant this "specialized" functionality.

As for item one, only a single function is needed with a prototype that is very similar to the prototype of the n-dimensional FFT function. There are many applications that can take advantage, especially of 1D and 2D convolutions of power-of-two sizes. As for item two, this specialized function would unlock the following optimizations:

1. expose more independent work benefiting small batches.
2. optimize out the inner transposes.
3. merge point-wise multiplication with FFTs around it. This merges a memory-intensive with a more compute-intensive kernel.
4. merge padding and extraction operations with outer 1D FFTs, which can optimize for padded values and only compute extracted values.

The fall-back implementation for not yet optimized cases is simple to implement on top of existing library functionality.

Data transfers between CPU and GPU memory are always a problem in real applications that do not output through the graphics connector. With the delay of PCIe gen 3.0 and its modest bandwidth improvement of a factor of two, this upgrade will not even remotely resolve the I/O bottleneck. We have already described how to reduce I/O requirements, but typically, convolution data is generated elsewhere/measured, so its generation cannot be moved to the GPU. Apart from reducing it, CUDA allows the programmer to deal with it. This requires programming the host code in terms of asynchronous GPU operations. Programming asynchronous operations is notorious for good reasons: it is complicated to get working reliably, and it complicates error handling and subsequent performance analysis. With only the possibility to overlap one transfer delay (partially), the payoff is very limited. We think that if it is too expensive or difficult to provide faster off-board I/O, the GPU will have to come closer to the CPU. This major architectural change has already been set into motion, but only to integrate some GPU-like cores on CPUs or to place an FPGA in a CPU socket, not for high-performance GPU computing.

7 Conclusion

There are many applications that need to perform lots of small FFT-based convolutions very quickly. This paper explains that straightforward implementations do not exhibit significant performance improvements and that drastic code transformations must be adopted to exploit the potential of GPUs. To efficiently run large numbers of small 2D convolutions on GPUs, it is important to:

1. maximize independent parallelism in the algorithm.
2. minimize the time it takes to transfer data between CPU and GPU memory.
3. tune the execution configurations.
4. move many CPU and GPU kernels into fewer GPU kernels.

Even so, data transfers may take valuable execution time and the need to open-up and/or reimplement FFT black box functionality dominates implementation time. To overcome this, our primary conclusion is that to provide application-programmer friendly, high-performance (small) convolution functionality, one new library function must be added to existing FFT libraries. Such a function has many users and brings the high-performance/low effort ratio to convolution applications. Second and just as critical to GPGPU performance in general is to provide a faster communication channel between CPU and GPU memory, if necessary, by moving the GPU closer to the CPU. This will significantly benefit all non-graphics GPU applications.

Acknowledgments. This work has been supported by Delft University of Technology and ASML Research. We would like to thank NVIDIA for donating some of the GPU boards used in this work.

References

1. Govindaraju, N., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High performance discrete fourier transforms on graphics processors. In: Proc. of the ACM/IEEE Conf. on Supercomputing, pp. 1–12. IEEE Press, Los Alamitos (2008)
2. Podlozhnyuk, V.: Image convolution with CUDA. Tech. rep., NVIDIA (2007)
3. NVIDIA: CUDA Programming Guide (February 2010)
4. Podlozhnyuk, V.: FFT-based 2D convolution. Tech. rep., NVIDIA (2007)
5. Podlozhnyuk, V.: Image convolution with CUDA. Tech. rep., NVIDIA (2007)