ESA SOCIS Project Report

An OpenCL Fast Fourier Transformation

Implementation Strategy

Sven DE SMET [a1]
[a] *Student at Ghent University*

Abstract. This paper describes an implementation strategy in preparation for an implementation of an OpenCL FFT. The two most essential factors (memory bandwidth and locality) that are crucial to obtain high performance on a GPU for an FFT implementation are highlighted. Theoretical upper bounds for performance in terms of the locality factor are derived. An implementation strategy is proposed that takes these factors into consideration so that the resulting implementation has the potential to achieve high performance.

Keywords. FFT, OpenCL, GPGPU, GPU, Parallelism

High-performance implementations of the FFT for GPUs already exist [1,2,3]. In an attempt to improve performance even further, optimization at every level of the architecture by combining the positive aspects of the different existing implementations is required. This includes

- Memory bandwidth maximization through the use of contiguous memory accesses for global and local memory
- Minimization of required memory bandwidth through the use of locality at the SIMT processor and core level
- Maximization of parallelism using instruction level parallelism and/or multiple warps per block
- Minimization of partition camping

To this end, I plan to construct an OpenCL FFT that combines the positive aspects of the hierarchical FFT and an FFT with contiguous accesses similar to the Stockham formulations. The hierarchical FFT will allow to optimize locality so that memory bandwidth requirements are minimized, while the contiguous formulation will allow to maximize memory bandwidth.

1. Notation

Let $\langle\psi\rangle$ denote $e^{-2\pi i\psi}$. The set of integers from $a \in \mathbb{Z}$ to $b \in \mathbb{Z}$ is denoted by $a..b \triangleq \{i|(i \in \mathbb{Z}) \wedge (a \leq i) \wedge (i \leq b)\}$. The set of non-negative integers smaller than

[1] sven.desmet@cubiccarrot.com

$c \in \mathbb{Z}$ is denoted by $\widehat{c} \triangleq 0..(c-1)$. Scalar multiplication and summation by a constant are overloaded to sets of integers (these are applied element-wise).

The i-th element of an array d is denoted with $d[i]$. For a set of integers I the sub-array with elements from I from lowest to highest and re-indexed from 0 to $I-1$ is denoted with $d[I]$ (so, for example, d[9] = d[$\widehat{23}+5$][2]).

2. The Fast Fourier Transformation

For a sequence of data d with length N the discrete Fourier transformation (DFT) $\mathcal{F}(d[\widehat{N}])$ is defined as:

$$\underset{f}{\overset{\widehat{N}}{\forall}} \mathcal{F}(d[\widehat{N}])[f] \triangleq \sum_{t}^{\widehat{N}} d[t] \left\langle \frac{ft}{N} \right\rangle \tag{1}$$

We decompose the time index t using two factors A and B of N such that $N = AB$ with $t \triangleq Br + s$:

$$\begin{aligned} \underset{f}{\overset{\widehat{N}}{\forall}} \mathcal{F}(d[\widehat{N}])[f] &= \sum_{r,s}^{\widehat{A}\times\widehat{B}} d[Br+s] \left\langle \frac{f(Br+s)}{N} \right\rangle \\ &= \sum_{r,s}^{\widehat{A}\times\widehat{B}} d[Br+s] \left\langle \frac{fr}{A} \right\rangle \left\langle \frac{fs}{N} \right\rangle \end{aligned} \tag{2}$$

Since f can be larger than A, different values of f can result in the same value for $\left\langle \frac{fr}{A} \right\rangle$. The decomposition of t thus induces a decomposition of $f \triangleq g + Ah$ with the same factors:

$$\begin{aligned} \underset{g,h}{\overset{\widehat{A}\times\widehat{B}}{\forall}} \mathcal{F}(d[\widehat{N}])[g+Ah] &= \sum_{r,s}^{\widehat{A}\times\widehat{B}} d[Br+s] \left\langle \frac{(g+Ah)r}{A} \right\rangle \left\langle \frac{(g+Ah)s}{N} \right\rangle \\ &= \sum_{r,s}^{\widehat{A}\times\widehat{B}} d[Br+s] \left\langle \frac{gr}{A} \right\rangle \left\langle \frac{(g+Ah)s}{N} \right\rangle \\ &= \sum_{r,s}^{\widehat{A}\times\widehat{B}} d[Br+s] \left\langle \frac{gr}{A} \right\rangle \left\langle \frac{gs}{N} \right\rangle \left\langle \frac{hs}{B} \right\rangle \end{aligned} \tag{3}$$

Many common FFT variants (the hierarchical or four-step FFT, DIT and DIF, both in-place and out-of-place) can be derived from this formulation.

2.1. A Contiguous Decimation-In-Time Formulation

A decimation-in-time (DIT) formulation is readily obtained from (3) by interpreting the summation over r as the computation of a frequency of a fourier transform of length A:

$$\begin{aligned}\underset{g,h}{\overset{\widehat{A}\times\widehat{B}}{\forall}}\mathcal{F}(d[\widehat{N}])[g+Ah] &= \sum_{s}^{\widehat{B}}(\sum_{r}^{\widehat{A}} d[Br+s]\left\langle\frac{gr}{A}\right\rangle)\left\langle\frac{gs}{N}\right\rangle\left\langle\frac{hs}{B}\right\rangle \\ &= \sum_{s}^{\widehat{B}}\mathcal{F}(d[B\widehat{A}+s])[g]\left\langle\frac{gs}{N}\right\rangle\left\langle\frac{hs}{B}\right\rangle\end{aligned} \tag{4}$$

This formulation re-expresses a length N DFT in terms of B smaller length A DFTs on interleaved sequences of data. By using dynamic programming to compute these sub-problems once and reusing the results to compute the larger DFT, the computational complexity can be reduced. By applying this single decomposition step recursively an FFT is obtained.

Let us consider a k-step FFT that is obtained using this decomposition on a sequence of length L where in every step $\frac{L}{A_q}$ interleaved DFTs of length A_q are combined in groups of B_q DFTs into $\frac{L}{N_q}$ interleaved DFTs of length N_q with $q \in 1..k$.

$$\begin{aligned}&\underset{q}{\overset{1..k}{\forall}}\underset{z}{\overset{\widehat{\frac{L}{N_q}}}{\forall}}\underset{g,h}{\overset{\widehat{A_q}\times\widehat{B_q}}{\forall}}\mathcal{F}(d[\frac{L}{N_q}\widehat{N_q}+z])[g+A_qh] \\ &\qquad = \sum_{s}^{\widehat{B_q}}\mathcal{F}(d[\frac{L}{N_q}(B_q\widehat{A_q}+s)+z])[g]\left\langle\frac{gs}{N_q}\right\rangle\left\langle\frac{hs}{B_q}\right\rangle\end{aligned} \tag{5}$$

Here, z iterates over the $\frac{L}{N_q}$ DFTs that are computed in every step q. Rewriting this as an algorithm that reveals the exact indexes in the intermediate data array d_q of step q, we obtain

$$\underset{q}{\overset{1..k}{\Theta}}\underset{z}{\overset{\widehat{\frac{L}{N_q}}}{\Theta}}\underset{g,h}{\overset{\widehat{A_q}\times\widehat{B_q}}{\Theta}} d_q[\frac{L}{N_q}(g+A_qh)+z] = \sum_{s}^{\widehat{B_q}} d_{q-1}[\frac{L}{N_q}(B_qg+s)+z]\left\langle\frac{gs}{N_q}\right\rangle\left\langle\frac{hs}{B_q}\right\rangle \tag{6}$$

where Θ denotes a for-loop. Since the twiddle factors are independent of z and since z moves with stride 1 through the data, an algorithm that accesses the data contiguously and with good reuse of the twiddle factors can be obtained by letting z be the inner loop:

$$\underset{q}{\overset{1..k}{\Theta}}\underset{g,h}{\overset{\widehat{A_q}\times\widehat{B_q}}{\Theta}}\underset{z}{\overset{\widehat{\frac{L}{N_q}}}{\Theta}} d_q[\frac{L}{N_q}(g+A_qh)+z] = \sum_{s}^{\widehat{B_q}} d_{q-1}[\frac{L}{N_q}(B_qg+s)+z]\left\langle\frac{gs}{N_q}\right\rangle\left\langle\frac{hs}{B_q}\right\rangle \tag{7}$$

When the value of B_q is small, the summation over s and the iteration over h are often expanded in the inner loop,

$$\underset{q}{\overset{1..k}{\Theta}}\underset{g}{\overset{\widehat{A_q}}{\Theta}}\underset{z}{\overset{\widehat{\frac{L}{N_q}}}{\Theta}}\underset{h}{\overset{\widehat{B_q}}{\Theta}} d_q[\frac{L}{N_q}(g+A_qh)+z] = \sum_{s}^{\widehat{B_q}} d_{q-1}[\frac{L}{N_q}(B_qg+s)+z]\left\langle\frac{gs}{N_q}\right\rangle\left\langle\frac{hs}{B_q}\right\rangle \tag{8}$$

For example, in a classical power-of-two DIT FFT where $B_q = 2$ for every $q \in 1..k$, the above algorithm specializes to

$$\mathop{\Theta}_{q}^{1..k}\ \mathop{\Theta}_{g}^{\widehat{A_q}}\ \mathop{\Theta}_{z}^{\widehat{\frac{L}{N_q}}} \left[\begin{array}{l} d_q[\frac{L}{N_q}g+z] = d_{q-1}[\frac{L}{N_q}(2g)+z] + d_{q-1}[\frac{L}{N_q}(2g+1)+z] \left\langle \frac{g}{N_q} \right\rangle \\ d_q[\frac{L}{N_q}(g+A_q)+z] = d_{q-1}[\frac{L}{N_q}(2g)+z] - d_{q-1}[\frac{L}{N_q}(2g+1)+z] \left\langle \frac{g}{N_q} \right\rangle \end{array}\right. \tag{9}$$

The write reference in the iteration over h in (8) has a stride of $\frac{L}{B_q}$ and the read reference in the summation over s has a stride of $\frac{L}{N_q}$. Within a z-iteration, there is one write access for every h and there is one distinct read access for every s. The next inner iteration over z accesses the data with stride 1 and is therefore contiguous for every write and read reference.

The next inner iteration over g accesses the data with stride $\frac{L}{N_q}$ for every write reference (h). Since this is equal to number of iterations of the z-loop this extends the contiguity of the write accesses over z to the next loop over g. Since the stride over s is also equal to the number of iterations of the z-loop the combined data read in an iteration of g for the entire summation is also contiguous. Since the stride of the data read for the iteration over g, is equal to the length of the data read over the summation in an iteration over g, $\frac{L}{A_q}$, the g-iteration also extends the contiguity of data read for the combination of read references.

Observe that, if the used twiddle factors are stored as a precomputed array with an array of stride 1 for every step q, different length FFTs can reuse the same precomputed table as long as the same factors B_q are used for the common step indexes q.

Since every d_q depends only on d_{q-1}, memory for d must only be available for two subsequent q-values at a time.

These out-of-place formulations can be transformed to in-place formulations to halve the memory required by performing a single step bit-reversal at every step such that the results for every inner iteration are stored in the same location as the source operands and performing a complete bit-reversal of the data before or after (depending on the choice of single-step bit-reversal) the algorithm completes.

2.2. A Hierarchical Formulation with Improved Locality

A hierarchical formulation can be obtained by interpreting (3) as a two-dimensional DFT with intermediate twiddle factors:

$$\begin{aligned} \mathop{\forall}_{g,h}^{\widehat{A}\times\widehat{B}} \mathcal{F}(d[\widehat{N}])[g+Ah] &= \sum_{r,s}^{\widehat{A}\times\widehat{B}} d[Br+s] \left\langle \frac{gr}{A} \right\rangle \left\langle \frac{gs}{N} \right\rangle \left\langle \frac{hs}{B} \right\rangle \\ &= \sum_{s}^{\widehat{B}} \mathcal{F}(d[B\widehat{A}+s])[g] \left\langle \frac{gs}{N} \right\rangle \left\langle \frac{hs}{B} \right\rangle \\ &= \mathcal{F}(\lambda s : \widehat{B} \,.\, \mathcal{F}(d[B\widehat{A}+s])[g] \left\langle \frac{gs}{N} \right\rangle)[h] \end{aligned} \tag{10}$$

The number of floating point operations for a standard power-of-two FFT on a sequence of length N is $5N\log_2 N$. This gives a *locality factor* of $\tau(N) \triangleq 5\log_2 N$ floating point operations per data element. The performance is therefore constrained by the bandwidth μ to $\tau(N)\frac{\mu}{2}$.

Choosing a decomposition of N in factors such that one factor allows the corresponding DFT to fit in local memory improves locality and thus reduces memory bandwidth requirements by increasing $\tau(N)$. If the other factor is too large to allow the DFTs along the other dimension to fit in memory, the hierarchical decomposition can be recursively applied to obtain a number of dimensions such that all DFTs along every dimension fit in the smaller and faster memory.

This leads to the four-step algorithm that performs multi-FFTs along both dimensions and also includes a step to transpose the data to ensure contiguity and a step to apply the twiddle factors.

3. GPU Implementation

3.1. Optimizing Memory Bandwidth with Contiguous Accesses

The contiguous access pattern of (8) can be used to optimize memory bandwidth by mapping iterations that access separate elements of a contiguous sequence of the data array to subsequent threads within a warp. This results in a bijection between the iteration indexes $(g, z) \in \widehat{A_q} \times \widehat{\frac{L}{N_q}}$ and the indexes $(w, v) \in \widehat{\frac{L}{\phi B_q}} \times \widehat{\phi}$ where

- v is the index of a thread within a warp,
- w is a warp-index (which may be mapped sequentially or in parallel, or a combination of both) and
- ϕ is the number of threads per warp.

The elements of every corresponding pair (g, z) and (w, v) are mapped to the same unique index $j \triangleq \frac{L}{N_q} g + z = \phi w + v$. As long as $\phi \leq \frac{L}{N_q}$ the separate threads within a warp execute a contiguous subset of iterations of z for the same iteration of g. The number of contiguous iterations of g that are executed by a warp is $\left\lceil \frac{\phi N_q}{L} \right\rceil$. The required bandwidth for fetching or computing the twiddle factors (one per distinct value of g) is therefore low too, except for the last steps.

As long as the number of iterations z is no less than ϕ, every distinct read and write access will be performed to a contiguous set of indexes of the data array. So, if the last factor B_k is larger than ϕ, optimal bandwidth utilization is ensured for every step except the last step.

For the last steps where multiple iterations of g are processed in a single warp, every distinct write access is still contiguous, but the read accesses used must be grouped in the same coalesced access to ensure optimal bandwidth utilization. The values read must then be communicated through local memory to the threads that work on the same iteration of g. Govindaraju et al. [2] discuss how such a transposition can be done efficiently through the use of padding to avoid bank conflicts.

3.2. Optimizing Locality for the Contiguous Formulation

The improved locality obtained with a hierarchical formulation can also be applied to the contiguous DIT formulation (8) by interpreting the summation over s as a local DFT:

$$\overset{1..k}{\underset{q}{\Theta}} \overset{\widehat{A_q}}{\underset{g}{\Theta}} \overset{\widehat{\frac{L}{N_q}}}{\underset{z}{\Theta}} \overset{\widehat{B_q}}{\underset{h}{\Theta}} d_q[\frac{L}{N_q}(g + A_q h) + z] = \sum_{s}^{\widehat{B_q}} d_{q-1}[\frac{L}{N_q}(B_q g + s) + z] \left\langle \frac{gs}{N_q} \right\rangle \left\langle \frac{hs}{B_q} \right\rangle$$
$$= \mathcal{F}(\lambda s : \widehat{B_q} \,.\, d_{q-1}[\frac{L}{N_q}(B_q g + s) + z] \left\langle \frac{gs}{N_q} \right\rangle)[h] \tag{11}$$

or, equivalently,

$$\overset{1..k}{\underset{q}{\Theta}} \overset{\widehat{A_q}}{\underset{g}{\Theta}} \overset{\widehat{\frac{L}{N_q}}}{\underset{z}{\Theta}} d_q[\frac{L}{N_q}(g + A_q \widehat{B_q}) + z] = \mathcal{F}(\lambda s : \widehat{B_q} \,.\, d_{q-1}[\frac{L}{N_q}(B_q g + s) + z] \left\langle \frac{gs}{N_q} \right\rangle) \tag{12}$$

We thus obtain a locality factor of $\tau(B_q)$ for the local DFT while we can maintain the highly contiguous access patterns. If the DFT over s is computed in a single core of a streaming multiprocessor (SM), B_q is constrained by the number of elements that can be fit into the registers after other local variables have been allocated. The in-place formulation for (8) can be used to minimize the required registers.

Since local memory can be used to communicate quickly between the threads executed on an SM, letting multiple threads work on the same DFT can further increase $\tau(B_q)$ for the SM level. If the data must be communicated between threads on the same SM core only, this communication may be done efficiently by a re-indexing approach for devices with CUDA Compute Capability 2.x as a result of reduced bank conflict constraints.

Table 3.2 lists the locality factors $\tau(N)$ for different values of N and the resulting upper bound on performance that can be achieved for several GPUs. The computed values show that it is necessary to use a hierarchic approach for locality since otherwise the performance is bounded by the memory bandwidth. At the same time, contiguity remains essential, because halving memory bandwidth due to non-contiguous accesses requires that the length of the FFT is squared to maintain the same performance bound.

The upper bound of the percentage of peak performance for a given locality factor is comparable for float and double computations on the Fermi architecture, because both bandwidth (in elements per second) and processing power are halved. Since twice as much local memory is required for double precision computation with the same locality factor, the maximum locality factor for doubles is one step lower than for floats.

The locality factor is constrained by the amount of local memory. Since the amount of local memory is of the same order as the amount of memory in all the registers of an SM combined, the amount of local memory is effectively constrained by the total register size.

Table 3.2 lists some of the relevant GPU parameters for different CUDA Compute Capabilities. The values show that a four-step hierarchic approach with intermediate transposition can have a very high locality factor. High performance could be achieved if the transposition could be carried out efficiently (without sacrificing too much memory bandwidth and while doing other computations in parallel). However, doing this transposition efficiently on the current architectures is not easy and even sacrificing half of the bandwidth erases the benefit of the increase in local register size.

Using the contiguous approach with maximal bandwidth requires that at least ϕ different sub-FFTs are computed simultaneously. For most of the steps s of larger FFTs no transposition in local memory is required if we let each SM core compute its own

GPU	GeForce 9400M G (DDR3)	GTX280	GTX220	8800GTX
Bandwidth (GB/s)	17.056	141.7	111.9	86.4
Bandwidth (complex floats/s)	2.132	17.7	14.0	10.8
Processing power (GFlops, float)	54	715.392	933.120	518
Maximal $log_2 N$ per CUDA core	6	7	7	6
CUDA Compute Capability	1.0	1.3	1.3	1.0
Performance bound (GFlops, float):				
$log_2 N = 1$	5.33	44.25	35	27
$log_2 N = 2$				
$log_2 N = 3$				
$log_2 N = 4$		177		
$log_2 N = 5$				135
$log_2 N = 6$	**31.98**	265.5	210	**162**
$log_2 N = 7$		**309.75**	**245**	189
$log_2 N = 8$		354	280	216

Table 1. Performance bound resulting from the locality factor when memory bandwidth is used optimally. The theoretical maxima that take the number of registers per SM core into account have been highlighted in bold. Architecture parameters have been obtained from [4,5].

CUDA Compute Capability	1.0	1.1	1.2	1.3	2.x
Registers (32-bit) per SM	8K	8K	16K	16K	32K
Registers (32-bit) per SM core	256	256	512	512	1024
Complex floats in registers per SM	4K	4K	8K	8K	16K
Complex floats in registers per SM core	128	128	256	256	512
Maximum number of resident threads per SM	768	768	1024	1024	1536
Maximum number of resident warps per SM	24	24	32	32	48

Table 2. Relevant GPU parameters for several CUDA Compute Capability values from [5].

sub-FFT. By using multiple threads per SM core, we can make sure most of the registers of the SM core are used (stronger restrictions on the number of registers per thread than on the number of register per SM core seem to apply) while the higher occupancy also allows to hide latencies (which is particularly important for accesses to global memory).

As an example for compute capability 2.x, if the number of 32-bit registers per thread is limited to 64, a 16 point float FFT could be performed in a single thread (using a *partial* in-place transformation) and 16 separate threads on the same SM core could be used to increase this to a 256 point FFT (while 32 of these FFTs are computed in parallel across the cores of the same SM). The size of the FFT per thread can be traded for the number of threads that work on the same FFT while keeping the size of the FFT per SM core (and the locality factor) constant. This trades occupancy for synchronization cost. Higher occupancy provides a more flexible form of parallelism than ILP (which is also available). For compute capability 1.2 and 1.3, a similar reasoning allows us to use a 128 point FFT per SM core.

3.3. Hiding Latency through Prefetching

Global memory accesses have a high latency, and if several SMs access the same memory bank this latency can be even higher. The optimal execution of an FFT on an SM takes

$5Nlog_2(N)$ cycles. If c SMs access the same memory bank simultaneously, the latency is $c\zeta$ for the access that is served last, where ζ is the access latency for a single access. On a GPU with many SMs and few memory banks, this latency may be high compared to the total clock cycles required for computation and the other threads on the SM may not be able to hide this latency.

To reduce the impact of long stalls for accesses to global memory, we can use some of the registers that remain available to implement prefetching from global memory and reduce the dependency constraints on registers used for writes to global memory. Other remaining registers can be used to improve pipelining of memory operations with the computation and to increase ILP.

3.4. Minimization of Partition Camping

To further minimize the effect of partition camping, we may try to map subsequent values of w to different SMs. If the strides of memory accesses are such that all accesses of an SM are mapped to the same bank, separate SMs will access separate banks. This ensures that memory bandwidth is optimally used as long as sufficient SMs are available. A global memory layout transformation for the data with corresponding re-indexing in the kernels may also provide an alternative solution to reduce or eliminate the effect of partition camping.

4. Related work

In [3] a GPU implementation that uses the four-step hierarchical approach is used for FFTs with a limited range in lengths.

In [2] several different GPU implementations of the FFT are discussed:

- A global memory FFT that uses a contiguous access pattern similar to the contiguous approach discussed above (but transposed: the strides of read and write accesses is mirrored). The iterator j (which has a role similar to our iterator j) is mapped to the threads of a workgroup. This implementation does not use shared memory and the locality factor is therefore limited by the number of registers per thread.
- A shared memory FFT where several threads work on the same FFT. This allows to increase the locality factor but the implementation is limited to FFTs that fit in the local memory of each SM.
- A four-step hierarchical FFT where several threads work on the same sub-FFT. This allows to maximize the locality factor but requires a transposition step in global memory, thereby sacrificing memory bandwidth.

In [1] an auto-tuning framework built on further extensions of the implementations in [2] is presented. The auto-tuning framework uses a global memory FFT with contiguous access patterns that uses shared memory to increase the locality factor. This includes the approach presented above as a special case. However, in [1], the number of sub-FFTs that are executed simultaneously on an SM is auto-tuned, while we will attempt to keep this number minimal to obtain the highest possible locality factor. By keeping this number constant and explicitly mapping the sub-FFTs to separate SM cores, we might also be able to reduce the penalty of the transpositions in shared memory.

5. Performance Potential

A comparison of performance results obtained by the most recent auto-tuning approach [1] with the theoretical upper bounds presented in the above analysis shows that some potential for further performance improvement still remains. Since I believe the auto-tuning framework includes the approach presented above as a specific case, it seems unlikely that performance can be improved further, unless the combination of the focus on the locality factor, the potentially improved efficiency of the local memory transposition as a result of our explicit per-core mapping and the use of prefetching gives our implementation some advantage.

6. Conclusions

This paper presents a detailed derivation of an FFT and highlights the two most essential factors (memory bandwidth and locality) that are crucial to obtain high performance on a GPU. Theoretical upper bounds for performance in terms of the locality factor are derived. An implementation strategy is proposed that takes these factors into consideration so that the resulting implementation has the potential to obtain high performance.

7. Acknowledgements

This work was supported by an ESA Summer of Code in Space project. I thank my ESA SOCIS project mentors Trevor Clarke and Kip Streithorst for helping make this project possible.

References

[1] Dotsenko, Y., Baghsorkhi, S.S., Lloyd, B., Govindaraju, N.K.: Auto-tuning of fast fourier transform on graphics processors. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. PPoPP '11, New York, NY, USA, ACM (2011) 257–266

[2] Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High performance discrete fourier transforms on graphics processors. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. SC '08, Piscataway, NJ, USA, IEEE Press (2008) 2:1–2:12

[3] Volkov, V., Kazian, B.: Fitting fft onto the g80 architecture, 2008. Technical report, University of California, Berkeley (2008)

[4] Wikipedia: Nvidia comparison — wikipedia, the free encyclopedia (2010) [Online; accessed 16-September-2011].

[5] Wikipedia: Cuda — wikipedia, the free encyclopedia (2011) [Online; accessed 16-September-2011].