Fitting FFT onto an Energy Efficient Massively Parallel Architecture

István Lőrentz
Electronics and Computers Department
Transylvania University of Braşov, Romania
and
Splash Software, Braşov, Romania
istvan@splash.ro

Mihaela Maliţa
Computer Science Department
Saint Anselm College
Manchester, NH, USA
mmalita@anselm.edu

Răzvan Andonie
Computer Science Department
Central Washington University
Ellensburg, WA, USA
and
Electronics and Computers Department
Transylvania University of Braşov, Romania
andonie@cwu.edu

ABSTRACT

We present novel implementations of the Fast Fourier Transform on the massively parallel Connex Array™(CA) circuit. The estimated performance is 19 GFlops (BenchFFT metric) of parallel computing 64 FFTs of size 1024, using 5 Watts. We compare the CA and NVIDIA's GTX 285 GPU performance. The CA is not a direct NVIDIA competitor, targeting a different application area. Considering its low power dissipation, the CA is an excellent solution for low cost mobile computing equipment, including sensors.

1. INTRODUCTION

Because of its importance, the Fast Fourier Transform (FFT) is used as a standard benchmark for parallel computers such as the HPC challenge and the NAS parallel benchmarks. Parallel FFT algorithms and implementations vary widely in theoretical and practical performance as a function of the computing architecture. However, these elegant algorithms lack robustness, as they usually do not map with equal efficiency onto interconnection structures different from those for which they were designed [4]. Optimal parallel algorithms (with respect to sequential bounds) are known for some processor topologies, but not for others.

With the advent of new many-core and graphical processors, the parallel FFT algorithms have to be adapted to fully exploit their architecture. A FFT implementation has to address all following aspects: *i)* data access pattern and strided access; *ii)* calculation/distribution of sin/cos twiddle tables; and *iii)* numerical precision.

We present several FFT implementations on a recent parallel processor, the Connex Array™(CA), and compare the results to CUFFT (NVIDIA's FFT library for GPU).

The CA, developed by BrightScale (formerly Connex Technology, Inc.), is a low power consumption, data-parallel architecture of 1024 processing elements. The main motivation for this work is our believe that the CA is a good candidate for low cost wireless computing equipment, due to its low power dissipation and cost.

The CA is a Single Instruction Multiple Data (SIMD) parallel in-memory device with thousands of cells that permits fast general purpose computations. It contains standard RAM circuitry at the higher level of the hierarchy, and a specialized memory circuit at the lower level, the Connex Memory, that permits parallel search at the memory-cell level and shift operations. A controller oversees the exchange of data between the two levels. Just as regular memory circuits, the operations supported by the CA can be performed in well-defined cycles whose duration is controlled by the current memory technology, which in today's technology is in the 2.5 ns range. Several computational intensive applications have been developed on this machine: data compression [22], alignment of DNA sequences [23], DNA search [18], massive computation of polynomials [19], frame rate conversion for HDTV [14], real-time packet filtering for detection of illegal activities [20], and neural computation [1].

The CA can perform vector operations in a small number of cycles. Addition, subtraction, multiplication, search can be performed in one cycle. The resulted execution times are very competitive. For instance, the present CA implementation can perform 25 million scalar products of 1024-tupled vectors per second.

The novelty of our work is the FFT implementation on the CA. The code is executed on a simulation of the CA.

After a brief overview of FFT implementations on NVIDIA's GPU (Section 2), in Section 3 we introduce the main characteristics of the newest CA circuit, the BA1024. Sections 4 - 7 present our CA FFT implementations, starting with the standard sequential algorithm. In Section 8, we perform several tests and compare the CA FFT implementations with the one obtained on NVIDIA's recent GTX 285 GPU. We evaluate the performance for 1D, 2D and multiple 1D transformations. Section 9 contains the final remarks.

2. RELATED WORK: FFT ON GPU

Most FFT implementations on GPU use graphics APIs such as current versions of OpenGL or DirectX [8, 13]. These APIs do not directly support scatters, access to shared memory, or fine-grained synchronization available on modern GPUs [8]. Access to these features is currently provided only by vendor specific APIs.

NVIDIA's development of CUDA for its GPUs opened new platforms for FFT computation. Probably the most general FFT implementation for GPUs available today is NVIDIA's CUFFT library, written in CUDA. It operates by taking a user-defined plan as input which specifies the parameters of the transform. It then optimally splits the transform into more manageable sizes if necessary. CUFFT employs a Radix-n algorithm and can handle FFTs of varying sizes on both real and complex data [24].

In 2008, Volkov and Kazin optimized the FFT implementation adapting it to the GeForce 8800GTX hardware capabilities, such as the massive vector register files and small on-chip local storage [24]. Since then, NVIDIA released the improved CUFFTv3 and new GPUs.

Govindaraju *et al.* [8] developed FFT algorithms for a broader range of input sizes and dimensions, using a combination of Radix-2, mixed radix, and prime factor methods to build FFT implementations. They also present algorithms that contrast different memory access models. For power-of-two-length inputs, they give three versions: a global memory version for large problems, a shared memory version for problems that fit in the GPU's shared memory, and a hierarchical version that wraps the FFT into multiple dimensions so that it can compute the result using shared memory computations. In each case, they tweak the implementations to leverage the memory structure of the GPU.

For the time being, the [8, 24] approaches appear to be the best GPU implementation developed. However, they are very architecture oriented and these results will most likely be obsolete in the near future.

3. THE CA BA1024 CIRCUIT

BA1024 is the latest CA chip, implemented in 2008 by BrightScale Inc., a start-up in Silicon Valley, and is described in [10, 12, 14, 15, 21]. We aim to review here its main performances and the Vector-C library (used Section 8).

The CA is a parallel programmable VLSI chip which consists of an array of processors. Functionally, it is an array/vector processor. It is not a dedicated, custom-designed (ASIC) chip, but a general purpose architecture. It is a fast, very cheap device, with low power consumption.

The CA operates on 1024-component vectors. Several CA chips can be integrated on the same board. Thus, the length of the processed vectors can be extended in increments of 1024, while receiving instructions and data from only one controller.

The 1024 cells are individually addressable as in a regular RAM, but can also receive broadcast/instructions or data on which they operate in parallel at a peak rate of 1 operation per cycle. This general concept fits the Processor-In-Memory paradigm. The cells are connected by a linear chain network, allowing fast shifting of data between the cells, as well as the insertion or deletion of data from cells while maintaining the relative order of all the data. All these operations are performed in a single memory cycle. The performances of BA1024 are:

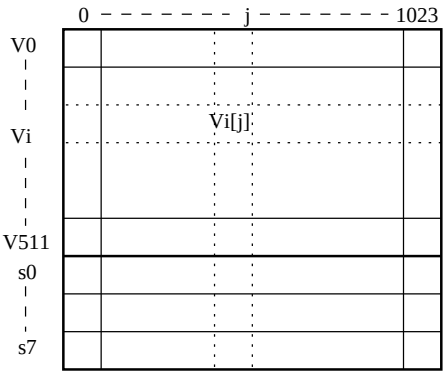

Figure 1: The internal state of Connex data parallel machine. There are 512 vectors, each having 1024 components, and 8 selection vectors, each having 1024 Booleans. (Reproduced from [12])

.

- Memory cycle: 2.5 ns.
- Computation: 400 GOPS at 400 MHz (peak performance)
- External bandwidth: 6.4 GB/sec (peak performance)
- Internal bandwidth: 800 GB/sec (peak performance)
- Power: $\approx$ 5 Watt
- Area: ≈ 50 mm^2 (1024-EU array, including 1Mbyte of memory and the two controllers).
- 65nm implementation

Using a 16-bit arithmetic, the BA1024 computes the scalar product of a 1024-toupled vector in 37.5 ns (26 millions scalar products/sec), and performs 1024×1024 matrix multiplications in 40 ms (25 operations/sec). Adding up to 1024 numbers is done in 5 cycles. Multiplication is done in 10 cycles.

The $P = 1024$ processing elements, each containing 512 registers, are interconnected in a ring. From an algorithmic point of view, the chip can be considered as an array of $P = 1024$ columns and $M = 512$ rows (Fig. 1). By convention, we represent it as an array of horizontal vectors. In C-style row-major notation, $A[i][j]$ denotes the i'th register inside the j-th processing element.

We use the Vector-C library [11]. Vectors are represented horizontally, with fixed size $P = 1024$ (see Fig. 1), and are declared as

```
vector A,B,C;
```

The operation $A = A + B * C$ means: $A_k = A_k + B_k * C_k, k = 0, \ldots, P-1$. Using Vector-C and operator overloading, this can be written as:

```
A += B * C;
```

Each processing element supports predicated execution. From a vectorial point of view, an operation (assignment, arithmetic) $A * B$

is performed only for positions $A_k * B_k$, $k = 0, \ldots, P-1$ where $k \in selectionset$.

Vector-C has a construct for "parallel-if":

```
WHERE(subset)
    A = B + C
ENDW
```

This is equivalent to the serial pseudo-code:

```
push selection
selection = subset
for (k = 0; k<1024; k++)
  if (selection[k])
    A[k] = B[k] + C[k];
pop selection
```

The WHERE ... ENDW blocks are nestable. Each WHERE saves the previous selection to a stack and the current selection becomes the disjunction of the previous set and the set given as argument. At the end of ENDW the previous selection set is restored. The outermost operations (not inside any WHERE... blocks) perform on the full set {0,..., 1023}. We use the term "complex vector" for brevity, which is stored by two vectors, holding the real and imaginary parts.

4. THE SERIAL FFT

The starting point for our CA FFT implementations is the serial FFT code. We use the basic Radix-2 Complex-to-Complex algorithm (see [5]). For the forward transform we use the *Decimation in Frequency* method. For the inverse transform we use the *Decimation in Time* method. The bit-reversal permutation needed for the in-place transform is done as a final stage of the forward transform, and as an initial stage for the inverse transform.

For certain applications (for example, convolution), the signals are transformed, multiplied and inverse-transformed. In this case, there is no need to re-order the FFT, and the bit-reversal step can be skipped (Fig. 2).

Below is the forward transform code for a complex data vector of length N with decimation in frequency:

```
complex data[N]
for (n=N; n>=2; n/=2)
  for (k=0; k<n/2; k++)
   complex w (cos(2*pi*k/n),
              -sin(2*pi*k/n));
   for (j=0; j<N; j+=N)
      butterfly(data[j+k],
                data[j+k+n/2]);
      data[j+k+n/2] *= w;
permute(data);
```

We use a simplified C-code representation, extracted from the real code, by omitting braces. Vectors are represented by uppercase letters. The Vector-C library functions will appear in the parallel code snippets, later.

The $permute(data)$ function re-orders in-place data, using bit-reversal permutation, i.e.: $data[k] <-> data[bitrev(k)]$.

This algorithm takes $2N \log_2 N$ multiplications and $3N \log_2 N$ additions. Several techniques allow to reduce the number of multiplications [5]: avoid trivial multiplications by $\pm 1, \pm i$; use higher radix 2^r algorithms; or trade multiplications for an increased number of additions. All retain the asymptotic $O(N \log N)$ complexity.

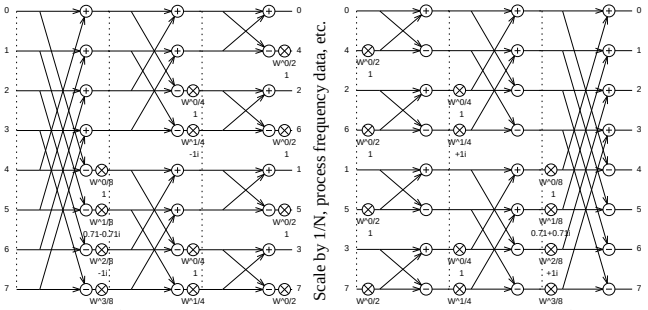

Figure 2: Data flow for a FFT of size 8, followed by an IFFT. Bit reversal not included: notice that the output order of the FFT matches the input order of the IFFT

5. VERTICAL FFT

The outer loop of the serial FFT code represents stages of smaller FFTs to be computed, whereas the inner j, k loops are the 'butterfly' loops. Since there is no data dependency in the inner loops, we can parallelize them. The outer loop has to be executed sequentially. In the following sections, we will describe several parallel FFT implementations on the CA. The parallel algorithms behind them are from the literature (see [5]). The implementations on the CA architecture are novel, adapted to the hardware characteristics of this chip.

The vertical FFT is the straightforward parallelization of the serial FFT code. Each processing element computes FFTs of size M over its internal registers, thus, having N processing elements, we can compute in parallel N FFTs of size M. No inter-processor data movement is needed.

5.1 The vertical butterfly

The building block of the vertical FFT computes the sum and difference of two complex vectors (Yr,Yi) and (Zr,Zi), using T for temporary vector:

```
T  = Zr; Zr = Yr - T;
Yr = Yr + T;
T  = Zi; Zi = Yi - T;
Yi = Yi + T;
```

If inputs are placed in vector arrays RE[] and IM[], each of size M, the forward transform is:

```
vector T1,T2;
for (n=M; n>=2; n/=2)
 for (k=0; k < (n/2); k++)
   wre=cos(2*pi*k/n);
   wim=sin(2*pi*k/n);
   for (int j=0; j<N; j+=n)
     ix  = j+k;
     iz  = ix + n/2;
```

```
      butterfly_vert(RE[ix],IM[ix],
                     RE[iz],IM[iz],T1);
      multiply(RE[iz],IM[iz],
               Wre, Wim, T1,T2, -1);
if (reorder)
  permuteVert(RE,IM, M);
```

The results will replace the input data RE[], IM[]. Notice that the complex multiplication operates on the result of the operation (Gentleman-Sande butterfly [7]).

The multiply() function executes the complex multiplication, which requires four real multiplications and two real additions, unless the following special cases:

- if $k = 0$, then skip multiplication
- if $n >= 4$ and $k = n/4$, then take the complex conjugate
- if $n >= 8$ and $k = n/8$, then the real and imaginary part of the twiddle factors are equal, the complex multiplication is done only in $2C_*$ and $2C_+$.
- if ($n >= 8$ and $k = 3*n/8$), then we have $Wre = -Wim$, which is similar to the previous case.

Each stage (loop n) consists of $M/2$ butterfly operations and $M/2$ complex multiplications. Each butterfly operation consists of 4 vector additions. If using four real multiplications and two additions per complex multiplication, we get a total of: $(4\ multiplications+ 6\ additions)M/2$. Having $\log_2 M$ stages, the parallel execution time is:

$$T_{vert}(M) = (3C_+ + 2C_{*scal})M \log_2 M \quad (1)$$

where

C_+ = cycles needed for vector addition.

C_{*scal} = cycles needed for vector - scalar multiplication.

During the vertical FFT execution, the chip, consisting of 1024 processing elements, computes in parallel a batch of $N = 1024$ FFTs, we can compute the cycles/FFT ratio as

$$cycles/FFT = \frac{T_{vert}(M)}{N}. \quad (2)$$

5.2 Computation vs IO throughput

In batch mode, where all $N = 1024$ FFTs are computed, in total $2MN$ scalar values are processed (real/imag parts of MN complex scalars). Thus, the number of cycles required to bring the data from external memory into the chip's vectors, and transfer the results back to external memory is:

$$T_{i/o} = 4MNC_{i/o} \quad (3)$$

where $N = P = 1024$ and $C_{i/o}$ = (average) number of cycles needed to bring one scalar value from external memory to chip.

The computation is not I/O bounded if $T_{i/o} < T_{vert}(M)$.

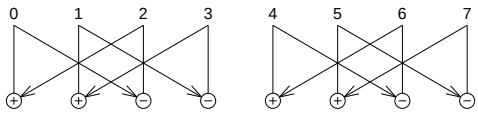

Figure 3: Stage n=4 for a Radix-2 FFT. The groups can be extended horizontally until all processing elements are filled

The vertical size, M, is bounded by the number of registers of a processing element. Thus, a complex element takes up to two registers (or four in floating point mode). Also, a slack space of 6 vectors must be kept for temporary storage of current twiddle factors, multiplications.

Observation: in the vertical FFT, each processor multiplies synchronously with the same twiddle factor. This can be viewed as a vector by scalar multiplication. For a given M, a source code can be generated where twiddle factors are scalar constants embedded in the code.

6. HORIZONTAL FFT

The horizontal FFT is a true vectorial FFT. The complex data to be transformed is brought from external memory into two vectors (real and imaginary parts). To compute FFT of size N, the data is distributed to N processing elements. However, if $N < 1024$, we can use the remaining processors to compute additional FFTs in the same time, as seen below.

Recall that in computing FFT of size N, each stage can be considered as computing N/n subproblems of size n. In serial code this is done by the two inner loops:

```
complex data[N]
for (k=0; k<n/2; k++)
  complex w = (cos(2*pi*k/n),
               sin(2*pi*k/n));
  for (j=0; j<N; j+=N)
    data[j+k+n/2] *= w;
    butterfly(data[j+k],
              data[j+k+n/2]);
```

On the CA, both loops are parallelized at once. Due to the butterfly pattern, vectors must be shifted (rotated) horizontally, that is a nearest-neighbor communications among processing elements.

6.1 The horizontal butterfly

To compute horizontal butterflies, data must be aligned for summation and subtraction. Note that the chip does vector operations element-wise on the same indices of vectors. We perform the alignment by left/right shifting of $n/2$ positions (code fragment and Figures 3 and 4). Note that all $N = 1024$ processing elements can be filled with data. Thus, for a stage n, we compute in parallel N/n butterflies.

```
butterfly2_horiz(
 vector Dre, vector Dim,
 const vector Are, const vector Aim,
 int n)
{
 vector Lre, Lim,  Rre,Rim; // temp
 Lre = rotateAllLeft(Are,  n/2);
```

```
 Lim = rotateAllLeft(Aim,  n/2);
 Rre = rotateAllRight(Are,  n/2);
 Rim = rotateAllRight(Aim,  n/2);
 WHERE ( selectFirstHalf(n)  )
   Dre = Are + Lre;
   Dim = Aim + Lim;
 ENDW
 WHERE ( selectSecondHalf(n)  )
   Dre = Are - Rre;
   Dim = Aim - Rim;
 ENDW
}
```

Function $selectFirstHalf(n)$ sets a predicate vector in the format: first group of $n/2$ elements of a vectors are selected for operation, second group of $n/2$ elements are not selected. The pattern is repeated over the entire length of a vector.

Similarly, $selectSecondHalf(n)$ selects the second half of each group. In set notation, the two subsets are defined as

$$selectFirstHalf(n) = \{k : (k \mod n) < n/2\}$$
$$selectSecondHalf(n) = \{k : (k \mod n) >= n/2\}$$
$$over\ k = 0...N-1$$

where n is the stage (subproblem) size and N is the total vector size.

index	0	1	2	3	4	5	6	7
first half	1	1	0	0	1	1	0	0
second half	0	0	1	1	0	0	1	1

Table 1: Example selection for N=8 and n=4

Now, in the longest case when $n = 1024$ (the entire machine vector length) we have the identity rotateLeft $(V, 512)$ = rotateRight $(V, 512)$. By exploiting this property, it is enough to execute rotation in a single direction instead of both (of 512 positions), saving 512 cycles.

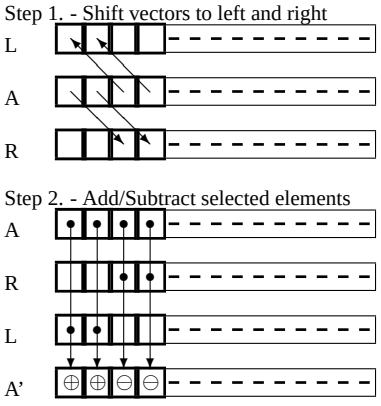

Figure 4: Horizontal butterfly calculation steps for n=4. A=input vector, R=right rotated, L=left rotated, A' = result vector. The entire vector is rotated, the arrows indicate only the data that will be used later. • indicates operand for addition or subtraction, selected using WHERE.. ENDW. The operation of groups of 4 elements are repeated (in parallel) over entire vector length of 1024.

k	0 . $\frac{n}{2}-1$	$\frac{n}{2}$ $n/2+1$. n-1
cos	. . .	1 $\cos\frac{2\pi}{n}$. $\cos\frac{2\pi\frac{n}{2}-1}{n}$
sin	. . .	0 $\sin\frac{2\pi}{n}$. $\sin\frac{2\pi\frac{n}{2}-1}{n}$

Table 2: Vectors of twiddle factors for a given n. The table is repeated horizontally N/n times. Values noted with (.) represent empty positions which do not take part in multiplications

6.1.1 Twiddle multiply

For the forward transform (isign=-1) we choose the decimation-in-frequency flow, with Sande-Gentleman style butterfly, where the multiplication is done after the add/subtraction. For the reverse transform (isign=+1) we used decimation-in-time, where the multiplication is done before the add/subtraction.

Only the 'right wing' of each butterfly needs to be multiplied, so we can use the same selection pattern as selectSecondHalf. Denote that in the serial code, the twiddle factor depends only on k and n, and same value used for $k, k+n, k+2n, \ldots$ For a given n, we need distinct values for $e^{\frac{i2\pi 0}{n}}, e^{\frac{i2\pi 1}{n}}, \ldots, e^{\frac{i2\pi(n/2-1)}{n}}$. Note that we fill the entire vector repeatedly, so we can compute multiple stages of same n in parallel (Fig. 4):

```
for (i=0; i<N; ++i)
  if ((i % n) >= n/2)
    k = (i % n) - n/2;
    horiz_cos[i] = cos(2*PI*k/n);
    horiz_sin[i] = sin(2*PI*k/n);
 else
    horiz_cos[i]=1.0;
    horiz_sin[i]=0;
```

Note that we store sin/cos values for isign=+1. For the forward transform, where we need isign=-1, we used the same sin/cos tables, but multiply with complex conjugate.

By isign we denoted the sign of i which appears in $e^{\frac{\pm i2\pi k}{n}}$. By convention, -1 is used for the forward FFT and +1 for the inverse FFT.

For a stage of length n, we need $n/2$ complex twiddle factors, all stored in two vectors. Since we have $\log_2 N$ stages, the total number of vectors required to compute FFT of size N is $2\log_2 N$. For example, for $N = 1024$ we get 20 vectors, which can be loaded once in the CA and re-used for several computations.

We see that butterfly operations in a stage of the horizontal FFT are done in parallel. The butterfly code requires 4 vector multiplications and 6 vector additions. In addition to that, each stage requires 4 rotations by $n/2$ positions (real and imaginary vectors rotated each to the left and right).

$$T_{hstage}(n) = 4C_* + 6C_+ + 2nC_{shift} \quad (4)$$

The execution time of a horizontal FFT is [1]:

[1] In this analysis, we did not count the reduction of trivial multiplications.

index	0 ... 511	512 .. 1023
2 vectors real, imag	FFT(512)	FFT(512)

$$T_{horiz}(N) = \sum_{r=1}^{log_2 N} (4C_* + 6C_+ + 2^{r+1}C_{shift}) \quad (5)$$

becomes

$$T_{horiz}(N) = (4C_* + 6C_+)\log_2 N + 2NC_{shift} \quad (6)$$

From this formula, it is clear that for large N, the shift operations becomes the dominant factor. However, having 1024 processing elements, we are constrained for $N <= 1024$.

If we compute horizontal FFTs for smaller sizes than the machine vector length, we can pack multiple FFTs in the same vector. For example, we can compute 4 FFTs of size 256, or 2 FFTs of size 512. The average cycles per computing horizontal FFTs of size N, using P processors ($N \leq P$) becomes:

$$cycles/FFT_h = T_{horiz}(N)/(1024/N) \quad (7)$$

The timings of horizontal FFTs are presented in Table 8.

6.2 Horizontal permutation

In order to obtain results in the correct order, the output of the Decimation-in-frequency FFT must be bit-reversal permuted. On vertical mode this is trivial, since it involves data exchange between the registers of the same processing element, however in horizontal mode we must use rotations. We rotate the data vector in both directions, and at each step we select those elements from the rotated vectors that are aligned with the bit-reversed index of the original vector. The code to permute a vector D of size N is:

```
vector L=D,R=D;   // temp
for (k=1; k<N; ++k)
  L = rotateAllLeft(L);
  R = rotateAllRight(R);
  WHERE(selectBitRev(k,N))
    D = L;
  ENDW
  WHERE(selectBitRev(-k,N))
    D = R;
  ENDW
```

the result is placed into D vector. The function selectBitRev(k,N) is defined as

$$sel[i] = (i \mod n) + k >= 0 \, and$$
$$(i \mod n) + k < n \, and$$
$$(i \mod n) = bitrevn(p, r)$$

where $i = 0, \ldots, 1023$, r = $\log_2 N$ and $bitrev(p, r)$ returns the integer p (r bits long) with bit order reversed.

6.3 Horizontal vs Vertical

Each method has it's own strengths and weaknesses. In Table 3 it is shown that for a given FFT $size <= 1024$ the fastest way to solve a single FFT is by the horizontal method. However, considering batch mode (cycles/FFT), the vertical arrangement is more efficient. The horizontal FFT can use pre-computed twiddle factors and load them in a number of $2\log_2 N$ vectors while the vertical FFT uses constant scalar multiplicators.

In this table we use N as FFT size, M as the number of FFTs being computed in parallel, R as the number of registers, and P as the number of processing elements.

7. 2D FFT

One of the nicest properties of the multidimensional Fourier transform is the separability that is, a 2D transform can be performed using 1D transforms on each row, then over the result, 1D transforms over each column. Having implemented the basic blocks for horizontal and vertical FFTs, we compute, on M rows and N columns:

```
void FFT2D(N,M) {
 fft_vert(re,im,M);
 for (v=0; v<M; ++v)
  fft_horiz(re[v],im[v]);
}
```

The for loop contains the horizontal transform. This is because we use the batch feature of the vertical transform: all N columns are computed in parallel. Adding the execution times of vertical FFT of size M and M horizontal FFTs of size N, we get

$$T_{2D}(N, M) = (2C_* + 3C_+)M\log_2 M + M((4C_* + 6C_+)\log_2 N + 2NC_{sh})$$

For $N = M$ this becomes

$$T_{2D}(N, N) = 3(2C_* + 3C_+)N\log_2 N + 2N^2 C_{sh}$$

To perform 2D transform, twiddle transforms for both horizontal and vertical are needed. The overall data requirements are:

- $2M$ data vectors (real + imaginary parts)
- $2\log_2 N$ vectors holding horizontal twiddle factors
- four vectors for temporary usage

On the chip having 512 vectors, we can perform 2D transforms of at most 128 rows and 1024 columns, for complex 2D arrays

Routine	Vertical	Horizontal
Constrains	$N < R/2, M = P$	$N \leq P, M = P/N$
Cycles	$(2C_* + 3C_+)N\log_2 N$	$(4C_* + 6C_+)\log_2 N + 2NC_{sh}$
Cycles / FFT	$\frac{(2C_* + 3C_+)N\log_2 N}{P}$	$\frac{(4C_* + 6C_+)\log_2 N + 2NC_{sh}}{P/N}$
Data vectors	$2N$	2
Twiddle factors	N, scalar constants	$2\log_2 N$ vectors

Table 3: Comparing the vertical vs horizontal algorithm

(2×128 data + 14 twiddle + 4 temporary vectors, while some vectors remains unused).

If $N < 1024$, the above routine actually computes $1024/N$ two dimensional transforms in parallel. For example, we can compute 16 complex FFTs of 64×64 in parallel, by partitioning a data matrix of 1024 columns and 64 rows into 16 blocks of size 64×64.

7.1 Computing large 1D FFT using the 2D FFT

We have not discussed yet the case of FFTs for data larger than the machine vector length, 1024. For this, we have an ingenious solution which will be described in the following. However, it is not a silver bullet and should be used with care, because two factors limit the practical maximum size of FFT. First, similar to the 2D case discussed before, the maximum 1D FFT size computable in-core is 1024×128 (for complex data). For even larger FFTs data must be brought in and partial results out of core sequentially, but this is not the scope of our work. The second issue is the numerical accuracy: it was shown in [9, 17] that the relative RMS error grows proportional to $\log N$. This puts an upper bound of FFT sizes computed using fixed-point 16-bit arithmetic and another bound using floating-point 32-bit.

The principle of the Cooley-Tukey algorithm [3] is to compute a FFT of composite size $N = N_1 \times N_2$ in terms of FFTs of size N_1 and N_2. We can reuse the same structure of the 2D algorithm, with a twiddle multiplication between the vertical and horizontal stages.

We have to compute 1D FFT of size $N \times M$ in three steps:

1. Compute N vertical FFTs of size M.
2. Multiply the array with twiddle factors $e^{\frac{i2\pi(1:NM)}{NM}}$.
3. Compute M horizontal FFTs of size N.

Note that the results are in transposed order.

The twiddle multiply requires $N \times M$ twiddle factors, which means an additional storage of M complex vectors. These factors are different from the factors used by horizontal or vertical computation, since they represent roots of unity of order $N \times M$ instead of N or M. If there is no space for store all M vectors, they must be considered "on the fly", using a recursive procedure. However, we have to be cautious, because this technique potentially amplifies numerical inaccuracy [17].

The composite FFT results are shown in Table 11. For example, a FFT of size 4096 can be factored as different combinations of horizontal (N) and vertical (M) sizes: $1024 \times 4, 256 \times 16, ...$, a FFT of size 1024 can be computed as 32×32 and so on (Table 8). If horizontal size is less than 1024, we can compute in parallel $1024/N$ transforms. When computing the performance in Table 8, we take into account the number of parallel FFTs computed.

8. EXPERIMENTAL RESULTS

For all CA FFT computations, we use the Vector-C emulator library to determine the following values:

C the number of cycles used in the computation, *not* taking into account the cycles needed to read/write data from/to external memory.

C/data samples the number of cycles divided by total number of data samples that were processed.

MOps measures scaled performance. This metric was defined by BenchFFT [6]:

$$MOps = M \frac{5N \log_2 N}{T} \tag{8}$$

where M is the number of FFTs computed, N is the FFT size, T = measured execution time, in μsec. The MOps is a FFT-specific measure, it is an estimate of the actual arithmetic operations/sec. This translates to MFlops when using 32-bit floating point.

Bandwidth is theoretical memory bandwidth, in MBytes/sec, required to fill the machine with data, and store the results back to external memory. The chip has an I/O plane capable of communicating with external memory in parallel while the array's processing elements are performing calculations. This plane must sustain the following data rate:

$$BW[MBytes/sec] = 2N_T S/time[\mu sec] \tag{9}$$

where N_T is the total number of data samples (vector size $\times$ number of used vectors). The 2 factor appears because each sample is transferred twice: once as an input and once as a result. S = sample size in bytes. For complex 16-bit numbers $S = 4$ and for 32-bit numbers $S = 8$. The calculation is I/O bounded if the required bandwidth given by the formula (9) is greater than the physical transfer rate of the device's external memory bus.

The emulator library uses the cycle counts of vector operations described in Table 4. For the vertical FFT, twiddle factors incorporated as constants for scalar $\times$ vector multiplication. Also the optimizations to avoid trivial multiplications were done. Table 6 depicts the results.

Vector operation	C_+	C_*
16-bit fixed point	1	10
32-bit floating point	12	19

Table 4: Cycle counts of vector operations

A core frequency of 400 MHz is used to estimate execution time.

M	N	cycles	cycles/ sample	MOps	Bandwidth [MB/sec]
4	1024	33	0.008	496484.8	397187.9
16	1024	553	0.03	237019.9	94808.0
64	1024	4997	0.07	157380.8	41968.2

Table 5: Vertical 1D FFT, with reordering. M=FFT size, N=batch size, 16-bit fixed point

The results show that, for executing FFTs up to size 1024 the horizontal mode offers the fastest solution, whereas the vertical mode is best to compute multiple FFTs of same size. Also, in the vertical mode offers the lowest cycles/sample ratio, since it calculates 1024 FFTs in parallel.

M	N	cycles	cycles/ sample	MFlops	Bandwidth [MB/sec]
4	1024	220	0.1	74472.7	59578.2
16	1024	2510	0.2	52219.9	20888.0
64	1024	18930	0.3	41544.2	11078.5

Table 6: Vertical 1D FFT, with reordering. M=FFT size, N=batch size, 32-bit floating point

N	M	cycles	cycles/ sample	MOps	Bandwidth [MB/sec]
4	256	43	0.04	95255.8	76204.7
16	64	259	0.3	31629.3	12651.7
64	16	835	0.8	14716.2	3924.3
256	4	2851	2.8	5746.8	1149.4
1024	1	6532	6.4	3135.3	501.7

Table 7: Horizontal 1D FFT, with reordering. N=FFT size, M=batch size, 16-bit fixed point

N	M	cycles	cycles/ sample	MOps	Bandwidth [MB/sec]
4	256	142	0.1	28845.1	23076.1
16	64	562	0.5	14576.5	5830.6
64	16	1342	1.3	9156.5	2441.7
256	4	3562	3.5	4599.7	919.9
1024	1	7447	7.3	2750.1	440.0

Table 8: Horizontal 1D FFT, with reordering. N=FFT size, M=batch size, 32-bit floating point

N (horiz)	M (vert)	nFFTs	cycles	cycles/ sample	MFlops	Bandwidth [MB/sec]
256	4	4	15076	3.68	5433.8	869.4
64	16	16	27014	1.65	12130.0	1940.8
32	32	32	41402	1.26	15829.2	2532.7
16	64	64	67626	1.03	19381.9	3101.1

Table 9: Computing 1D-FFT of size 1024, as different combinations of N (horizontal) $\times$ M (vertical) sizes (32-bit floating-point)

	N				
M	4	16	64	256	1024
2	180	612	1764	5796	13158
4	465	1329	3633	11697	26421
8	1085	2813	7421	23549	52997
16	2533	5989	15205	47461	106357
32	5809	12721	31153	95665	213457
64	13169	26993	63857	192881	428465

Table 10: Cycles of composite $N \times M$ 1D FFT (N=horizontal, M=vertical size), 16-bit fixed point

For an application of fixed FFT size, one can choose the most suitable algorithm, either horizontal, vertical or the combined. The combined horizontal-vertical method allows in-core computing of

	N				
M	4	16	64	256	1024
2	538	1378	2938	7378	15148
4	1396	3076	6196	15076	30616
8	3330	6690	12930	30690	61770
16	7814	14534	27014	62534	124694
32	17978	31418	56378	127418	251738
64	40746	67626	117546	259626	508266

Table 11: Cycles of composite $N \times M$ 1D FFT (N=horizontal, M=vertical size), 32-bit floating point

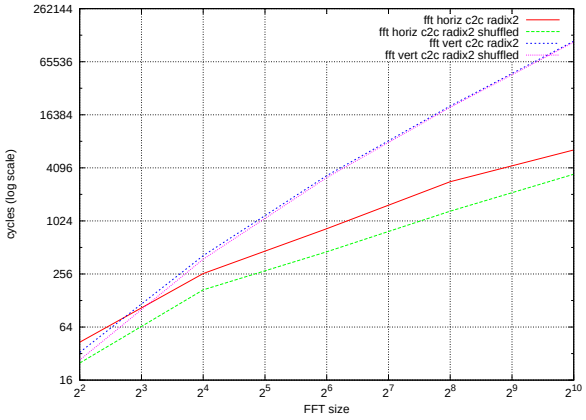

Figure 5: Cycle counts for 1D FFT (16-bit)

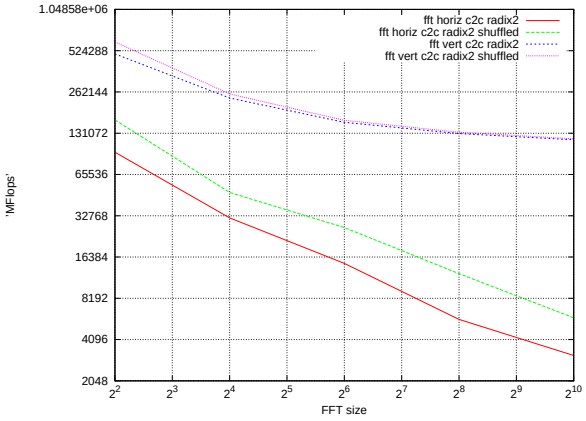

Figure 6: Batch MOps performance for 1D FFT (16-bit)

1D FFTs of sizes up to 1024×128 or 2D FFTs up to 1024 (horizontal) $\times$ 128 (vertical), for complex numbers.

One of the key features of the CA chip is the power dissipation: using 5 Watts, we estimated that can compute 64 1D FFT-s of size 1024 in 67 Kcycles that's 169 μsec, at 400 MHz. Every second, the chip can compute 378552 FFTs of size 1024. For this setup, using the BenchFFT metric, we measured 19GFlops, that is 3.8 GFlops/Watt.

For comparison, we performed a test on NVIDIA's GTX 285, using CUFFTv3, and the same BenchFFT benchmark. We obtained around 340GFlops at 204 Watts, using a plan to compute 16K FFTs

of size 1024, with CUFFT library functions:

```
cufftPlan1d(1024,CUFFT_C2C,16384);
cufftExecC2C(plan,data,data,CUFFT_FORWARD);
```

The comparison must be treated very carefully, since the general-purpose graphical processor is a chip of different category and complexity (Table 12).

Parameter	CA BA1024	NVIDIA GTX 285
Freq (MHz)	400	1476 (shader clock)
Power (Watt)	5	204
Area (mm^2)	50	470
Technology	65 nm	55 nm
Bandwidth (GBytes/sec)	6.4	160
Bus width	128 bit	512 (32x16) bit
Processing Units	1024	240:80:32
Year	2009	2009 January
Transistors	120 Million	1.4 Billion

Table 12: Characteristics of CA and NVIDIA GTX 285. Information retrieved from [16]

9. CONCLUSIONS

The CA is not a direct competitor to existent GPUs, since it is targeting different application areas. We are at the debut of energy-conscious computing, with a great deal of the industry's attention being given to the introduction and use of power-management mechanisms and controls in individual hardware components. There is some evidence [2] that the amount of energy consumed by mobile and desktop computing equipment is of roughly the same magnitude as that used by servers in data centers, although. Considering its low power dissipation, the CA is an excellent solution for low cost mobile computing equipment, including sensors.

We choosed the NVIDIA GPU comparison because this is a standard circuit. Certainly, a one-to-one comparison is almost impossible. In the future, we plan to compare the CA FFT with implementations on other parallel architectures.

10. REFERENCES

[1] R. Andonie and M. Maliţa. The Connex Array™ as a neural network accelerator. In *CI '07: Proceedings of the Third IASTED International Conference on Computational Intelligence*, pages 163–167, Anaheim, CA, USA, 2007. ACTA Press.

[2] D. J. Brown and C. Reams. Toward energy-efficient computing. *Queue*, 8(2):30–43, 2010.

[3] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[4] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. Logp: a practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.

[5] A. G. Eleanor Chu. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms (Computational Mathematics)*. CRC Press, 1999.

[6] M. Frigo and S. G. Johnson. BenchFFT. `http://www.fftw.org/benchfft/`.

[7] G. M. Gentleman and G. Sande. Fast Fourier transforms for Fun and Profit. In *1966 Fall Joint Computer Conference*, volume 29, pages 563–578. AFIPS Proc, 1966.

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

[9] P. Kabal and B. Sayar. Performance of fixed-point FFT's: Rounding and scaling considersations. *IEEE ICASSP*, 1:221–224, 1986.

[10] M. Maliţa, G. Ştefan, and D. Thiébaut. Not multi-, but many-core: designing integral parallel architectures for embedded computation. *SIGARCH Comput. Archit. News*, 35(5):32–38, 2007.

[11] M. Malita. Vector-C library. `http://www.anselm.edu/homepage/mmalita/ResearchS07/WebsiteS07/`.

[12] M. Malita and G. Stefan. Integral parallel architecture & berkeley's motifs. In *ASAP '09: Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 191–194. IEEE Computer Society, 2009.

[13] K. Moreland and E. Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, 2003.

[14] G. Ştefan. The CA1024: SoC with integral parallel architecture for HDTV processing. In *4th International System-on-Chip (SoC) Conference & Exhibit, November 1-2*, Radisson Hotel Newport Beach, California, 2006.

[15] G. Ştefan, A. Sheel, B. Mitu, T. Thomson, and D. Tomescu. The CA1024: a fully programmable system-on-chip for cost-effective HDTV media processing. In *Hot Chips: A Symposium on High Performance Chips, August 20-22*, Memorial Auditorium, Stanford University, 2006.

[16] NVIDIA Corporation. NVidia GeForce GTX 285. `http://www.nvidia.com/object/product_geforce_gtx_285_us.html`.

[17] J. C. Schatzman. Accuracy of The Discrete Fourier Transform And The Fast Fourier Transform. *SIAM J. Sci. Comput*, 17:1150–1166, 1996.

[18] D. Thiebaut, G. Ştefan, and M. Maliţa. DNA search and the Connex technology. In *Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'06)*, Bucharest, Romania, 2006.

[19] D. Thiebaut and M. Maliţa. Fast polynomial computation on Connex Array. Technical Report 303, Smith College, November 2006.

[20] D. Thiebaut and M. Maliţa. Real-time packet filtering with the Connex Array. In *Proceedings of the International Conference on Complex Systems*, pages 501–506, Boston, MA, 2006.

[21] M. Thiebaut and G. Ştefan. Memory engine for the inspection and manipulation of data. U.S. Patent No. 6,760,821, July 2004.

[22] M. Thiebaut and G. Ştefan. Ziv-Lempel compression with the Connex Engine. Tech. Rep. 077, Dept. Computer Science, Smith College, Northampton, MA, 01063, January 2002.

[23] M. Thiebaut and G. Ştefan. Local alignment of DNA sequences with the Connex Engine. In *The First Workshop on Algorithms in BioInformatics WABI 2001*, BRICS Univ. of Aarus, Denmark, August 2001.

[24] V. Volkov and B. Kazian. Fitting FFT onto the G80 architecture. UC Berkeley CS258 Project Report, May 2008.