Discrete Fast Fourier Transform

Severin Davis

Teja Dharma

High Performance Computing with Graphic Cards

University of Stuttgart

Contents

Motivation

Fast Fourier Transforms are useful in a variety of applications, most notably signal processing. If signal processing must occur as fast as possible or with a very large input size, parallelization of a Fast Fourier Transform algorithm can reap large benefits in required processing time.

In this project, a Fast Fourier Transform algorithm is implemented. Initially it is implemented as single threaded, serially-executing program. This implementation is analyzed for candidates for parallelization which are then moved to an OpenCL kernel.

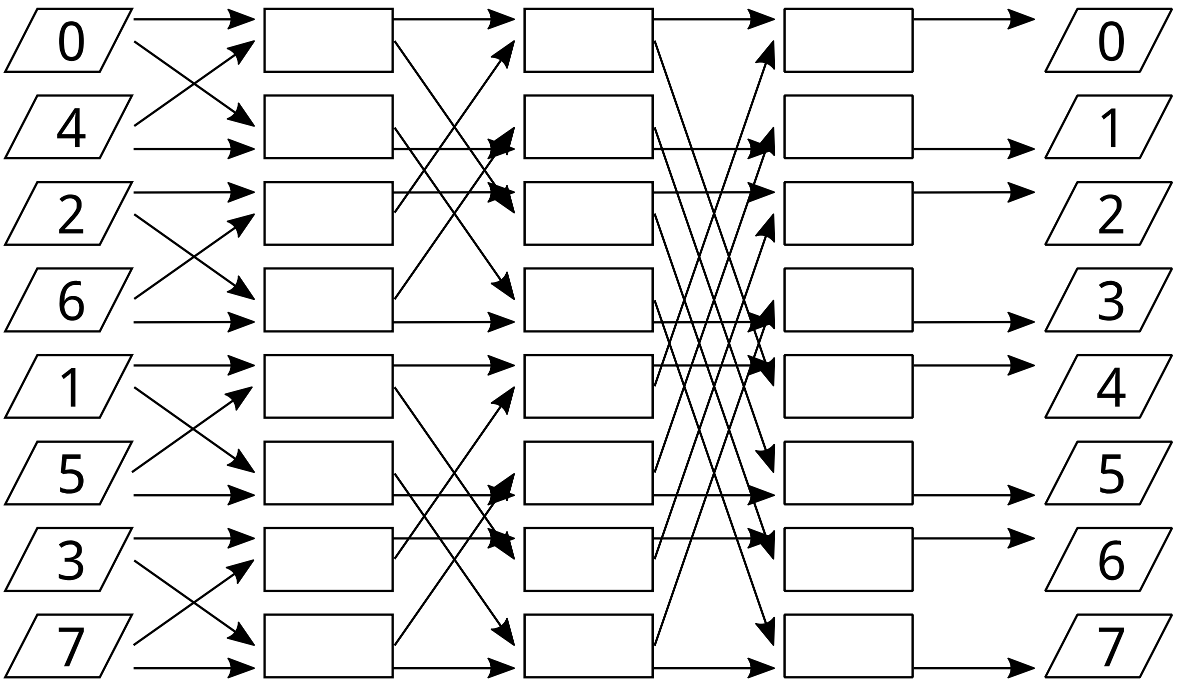
These two implementations are compared both in theoretical complexity as a function of input size, as well as real world execution speed on multiple platforms.

Program structure

For this Fast Fourier Transform implementation, the Tukey-Cooley Algorithm was chosen. This algorithm works in stages in which two independent indices of the input array interact. The results of these interactions are placed back into the array and the algorithm is run in a chained manner, in which the results of previous stages are used in the next stage. Each interaction two indices and a computed root of unity. Additionally, the input must be placed in a specific order.
As such, our implementation could be broken down into three sub problems, each with their unique challenges.

1. Reordering of the input
2. Mapping and implementation of interactions
3. Computation and mapping of roots of unity

Each of these is visible in the example diagram of an input size of 8 below.
The reordering of the input is visible on the input nodes at the left of the diagram.
The interacting index pairs, or stages, are shown as arrows. The boxes shown indicate the computations performed involving the roots of unity and the two inputs.



Reordering of the input

Inputs need to be reordered before the algorithm can be run. Unfortunately, because this reordering does not operate on independent indices, this portion of the FFT implementation was not a candidate for parallelization. Instead, this was implemented as a recursive function that executes prior to any FFT execution.

```
std::vector<doublec> reorder_input(std::vector<doublec> input)
{
	//check base case
	if (input.size() <= 2)
		return input;

	//create even and odd index reorder vectors
	std::vector<doublec> even(input.size()/2);
	std::vector<doublec> odd(input.size()/2);

	//split even and odd indices
	for (unsigned int i = 0; i < even.size(); i++)
	{
		even.at(i) = input.at(i * 2);
		odd.at(i) = input.at((i * 2) + 1);
	}

	//recursive call to reorder even and odd index vectors
	even = reorder_input(even);
	odd = reorder_input(odd);

	//regroup returned vectors and return
	std::vector<doublec> result;
	result.reserve(input.size());
	result.insert(result.end(), even.begin(), even.end());
	result.insert(result.end(), odd.begin(), odd.end());

	return result;
}
```

Mapping and implementation of interactions

The number of interactions occurring at each stage is half that of the input size. For example, in the given diagram, the sample size is 8. The left column of arrows, however, always shows two pairs of nodes interacting, creating a total of four interactions in that stage. Similarly, stages 2 and 4 also have 4 interacting pairs. As such, the number of threads spawned is half of the input size.

Thread mapping

One major challenge to this approach was finding the corresponding mapping from the thread number to the nodes it operates on in a specific stage.

As an example, the diagram shows thread0 mapped to index 0 and index 1 of the input array. Similarly, thread 0 is mapped to index 0 and 2 and index 0 and 4 at stages 2 and 4, respectively.

By analyzing the pattern, the following mapping was found to output an index, given the thread number and the current stage.

```
unsigned int thread_index_map(unsigned int thread_id, unsigned int stage)
{
      return (stage * 2 * (thread_id / stage)) + thread_id % stage;
}
```

Finally, after retrieving the one thread to index mapping, the other interacting index can be found simply by adding the current stage to the returned mapping.

Interaction implementation

Once the correct mapping is found, the two data items at those indices interact with each other, as well as an additional root of unity.

This interaction is defined as:

result[home_index] = result[home_index] + (result[target_index] * root)

result[target_index] = result[home_index] – (result[target_index] * root)

Of interest is the fact that the root for a particular interaction remains constant. As such, the (target_index*root) can be computed only once, further optimizing for speed of execution.

This is implemented in the code shown:

```
doublec pq = doublec_mul(result.at(target_index), roots.at(home_root));
doublec top = doublec_add(pq, result.at(home_index));
doublec bottom = doublec_sub(result.at(home_index), pq);
```

Computation and mapping of roots of unity

The required degree of the roots of unity for this FFT implementation is directly related to the size of the input. For instance, and input of size 8 will require 8 roots of unity. As such, instead of computing the necessary root for that particular stage and thread, the required roots are computed at the beginning of the algorithm and are simply referenced later via an additional mapping function.

Additionally, because roots are symmetric around the unit circle, we can save time by simply computing half of them. For example, an input size of 8 will only require 4 roots to be computer. Coincidentally, since 4 threads are spawned for the same input size, each thread is tasked with computing one root.

This computation is shown below:

```
for (unsigned int i = 0; i < roots.capacity(); i++)
{
      double arg = (2 * pi * (double)i) / ((double)result.capacity());
      roots.at(i).real = cos(arg);
      roots.at(i).imag = -1 * sin(arg);
}
```

Like the thread to index mapping function, threads need a mapping function to find the correct root to use, as shown below:

```
unsigned int thread_root_map(unsigned int thread_id, unsigned int estage,
unsigned int istage)
{
      return istage * (thread_id % estage);
}
```

In this function, estage counts up (1, 2, 4, 8, ...) while istage counts in reverse (..., 8, 4, 2, 1).

Parallelization and expected results

Examination of the diagram of the FFT algorithm reveals that the number of stages of the algorithm is logarithmically related to the size of the input. Similarly, the number of operations performed within each stage is linearly related to the size of the input. Additionally, the non-recurring cost of computing the roots is also linearly related to the size of the input.
Thus, the runtime complexity of the serially executing algorithm O(n+n*log(n)) where n is the size of the input. This reduces to just O(n*log(n)).
Analyses of the algorithm reveals several two primary candidates for optimization:

1. Root computation
2. FFT algorithm

Parallelization

Root computation

Parallelizing the root computation simply maps one root computation to each thread. Thus, the runtime complexity for this portion reduces from O(n) to O(1).

FFT algorithm

The stages of the algorithm cannot be parallelized, as they rely on previous stages' results. However, the interactions within each stage are easily parallelized by mapping each interaction to its own thread. Thus, the runtime complexity for this portion reduces from O(n*log(n)) to O(log(n)).

Overall this results in a runtime complexity reduction from O(n*log(n) to O(log(n)).

Expected Results

Timing results from the implementation are expected to roughly follow these complexities.
However, there is an overhead cost associated with moving data the OpenCL device as well as an associated cost in starting up the computation on the device. This introduces an overhead and causes a breakeven point in the total run time. This means that for dataset inputs below a certain size, the OpenCL overhead would cause the theoretically faster OpenCL implementation to be slower.

Results

Run code on multiple systems, multiple times.

Compile results graphically.

Identify and discuss improvements.

Identify and discuss bottlenecks and potential solutions.

Discuss difference in performance between different systems (dedicated GPU vs integrated graphics)

Sources