TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER FACULTY OF EXACT SCIENCES
THE BLAVATNIK SCHOOL OF COMPUTER SCIENCE

High-Performance GPU and CPU Signal Processing for a Reverse-GPS Wildlife Tracking System

Thesis submitted in partial fulfillment of the requirements for the
M.Sc. degree of Tel-Aviv University by

Yaniv Rubinpur

The research work for this thesis has been carried out at Tel-Aviv University
under the direction of Prof. Sivan Toledo

January 2021

Abstract

We present robust high-performance implementations of signal-processing tasks performed by a high-throughput wildlife tracking system called ATLAS. The system tracks radio transmitters attached to wild animals by estimating the time of arrival of radio packets to multiple receivers (base stations). Time-of-arrival estimation of wideband radio signals is computationally expensive, especially in acquisition mode (when the time of transmission of not known, not even approximately). These computation are a bottleneck that limits the throughput of the system. The thesis reports on two implementations of ATLAS's main signal-processing algorithms, one for CPUs and the other for GPUs, and carefully evaluates their performance. The evaluations indicates that the GPU implementation dramatically improves performance and power-performance relative to our baseline, a high-end desktop CPU typical of the computers in current base stations. Performance improves by more than 50X on a high-end GPU and more than 4X with a GPU platform that consumes almost 5 times *less* power than the CPU platform. Performance-per-Watt ratios also improve (by more than 16X), and so do the price-performance ratios.

Chapter 1

Introduction

ATLAS is a reverse-GPS wildlife tracking system, targeting mostly regional movement patterns (within an area spanning kilometers to tens of kilometers) and small animals, including small birds and bats [18, 21]. ATLAS is a mature collaborative research effort: 6 systems have been set up and are operating in 5 countries on 3 continents. The first system has been operating for about 6 years almost continuously and has produced ground-breaking research in Ecology [4, 20].

ATLAS tracks wild animals using miniature radio-frequency (RF) transmitting tags attached to the animals [17, 19]. The transmissions are received by ATLAS base stations that include a sampling radio receiver and a computer running Linux or Windows. The computer processes RF samples to detect transmissions from tags and to estimate the time of arrival (ToA) of the transmissions. It reports the reception times to a server via an internet connection, usually cellular. The server estimates the location of a tag from ToA reports of the same transmission by different base stations [21].

The signal processing that ATLAS base stations performs is computationally demanding and is one of the main limiting factors of the throughput of the system (the number of tags that it can track and the number of localizations per second that it can produce). The signal-processing algorithms were initially optimized for single-threaded on CPUs, but no significant effort has been made to exploit multiple cores effectively.

This thesis presents a new implementation of the ATLAS signal-processing code[1] designed to effectively exploit graphical processing units (GPUs). Our aim in developing this implementation was to significantly improve the throughput of the system and to reduce the power consumption of base stations. Reduced

[1]The current CPU and GPU versions of the code are available, along with the data files requires to run the code, at `http://www.tau.ac.il/~stoledo/Tools/atlas-dsp-heteropar2020.zip`.

power consumption reduces the cost and complexity of base stations that rely on solar and wind energy harvesting, such as those deployed in the shallow Wadden sea; it is not particularly important in base stations connected to the power grid. High throughput is useful in most base stations. As part of this project, we also exposed a little more parallelism in the original CPU implementation, but it was not our intention to make it as parallel as possible, because that would have little value to users (who should use the GPU implementation) and would necessitate replacing our simple single-threaded task scheduler with a complex concurrent one.

We also evaluate the performance of both the (slightly improved) CPU code and the new GPU code on real recorded data. The evaluations, performed on two CPU platforms and on three GPU platforms, show dramatic improvements relative to our baseline, a high-end desktop CPU that is typical of the computers in current base stations. The improvements are both in terms of absolute performance (more than 50X with a high-end GPU and more than 4X with a GPU platform that consumes almost 5 times *less* power than the CPU platform), in terms of performance-per-Watt ratios (more than 16X), and in terms of price-performance ratios. However, because we did not attempt to achieve top multi-core performance on CPUs, these results should not be taken as fair comparisons of the hardware platforms; they are meant mainly to demonstrate the level of performance that is achievable on such tasks on GPUs using a single-threaded scheduler coupled with GPU data parallel tasks.

The rest of this thesis is organized as follows. Chapter 2 provides background material required to understand our contributions. It describes how ATLAS base stations operate, how their scheduler operates, and the basics of GPU programming with CUDA. Chapter 3 describes the main signal processing algorithms that ATLAS base stations use to detect tag transmissions and to estimate their arrival times. The CPU and GPU implementations of these algorithms are described in Chapter 4. The results of our evaluation of the performance and power consumption of the two implementation on a variety of platforms are described in Chapter 5. The new GPU implementation is already deployed in the field; experiences from this deployment are described in Chapter 6. Chapter 7 describes related work, and Chapter 8 discusses our results and presents our conclusions from this research.

A preliminary version of this thesis has been published in the proceedings of HeteroPar 2020 [15]; it was also submitted for journal publication.

Chapter 2

Background

ATLAS tags transmit a fixed unique pseudorandom packet every second, 2 s, 4 s, or 8 s. The packets are 8192-bit long and the bitrate is around 1 Mb/s. The data is frequency modulated (FSK); ATLAS can also use phase modulation (PSK; see [12] for details), but in this thesis we focus on signal processing for frequency modulation, which is what almost all the deployed tags use. The sampling receiver in each base station sends a continuous stream of complex RF samples, usually at 8 or 8.33 Ms/s, to a computer. The samples are placed in a circular buffer. A high-level scheduler repeatedly extract a block of samples from the buffer and processes it. The size of the circular buffer allows for processing delays of more than 10 s; this simplifies the scheduler and the signal-processing code considerably relative to in-order stream processing with hard deadlines.

The signal processing aims to detect whether packets from specific tags appear in the block, to estimate the precise (sub-sample) time of arrival (ToA) of each packet, to estimate the (relative) power of the packet, and to estimate a signal-to-noise ratio that is correlated with the variance of the ToA estimate. This data is sent to a server that estimates the locations of the tags [21].

2.1 The High-Level Scheduler

ATLAS base stations use a high-level scheduler, implemented in Java. The scheduler creates two kinds of tasks for the signal-processing code. *Searching-mode* (acquisition-mode) tasks process blocks of 100 ms and try to detect packets from all the tags that have not been detected in the past few minutes. This set of tags is called the *searching queue*. It can consist of over 100 tags. Since all tags transmit on one or two frequencies, the FSK demodulation step is performed only once or twice per block of samples, but the number of pseudorandom codes that must be correlated with the demodulated signals can be large. *Tracking-mode* tasks

aim to detect an 8 ms packet from one particular tag in a block of about 12 ms of samples. These tasks perform demodulation and correlate the demodulated signal with one pseudorandom code.

Normally, the scheduler allocates 50% of the processor's time to searching and 50% to tracking, in an amortized sense, simply to avoid starvation of one of the tasks. If one of the queues is empty, all the processing resources are devoted to the other queue.

The scheduler is sequential; it generates one task at a time and performs it to completion, devoting to it all the cores except for one that handles incoming samples. this simplifies its algorithms but places all the responsibility to efficiently utilize multiple cores to the signal-processing code.

The behavior of the scheduler is illustrated in Figure 2.1. Each row in the figure depicts the state of the scheduler at a particular time and in most rows, also a decision made by the scheduler. Within each row, events are ordered from left to right. Each colored square represents a received transmission from a tag; the color represents the tag. The horizontal span of the square represents a span of time (samples) in which the transmission was received. In the top line, the base station has already successfully detected the six transmissions of the orange tag and the three of the blue tag shown to the left of the cyclic buffer. The check marks indicates that these transmissions were successfully detected. The tracking queue consists of the orange and blue tags, and the scheduler correctly predicts their first unprocessed transmissions, which are currently stored in the buffer. The buffer also contains one transmission of the orange tag that is still stored in the buffer, as well as additional transmissions of the orange, blue, and purple tags. The RF samples are stored at the front (right end) of the buffer and when that happens, old samples from the read (left end of the buffer) are discarded.

The second row the next action of the scheduler: it decides to serve the searching queue. It processes the range of samples shown by the bracket below the buffer. This range contains transmissions of the orange and blue tags, which the searching tag does not search for, and a transmission of the purple tag, which the task does search for. The third row shows the outcome of searching. The transmission of the green tag was detected, so it is checked. This caused the green tag to move from the searching to the tracking queue. The scheduler correctly predicts its next transmission time. Also, searching moves forward, with a 10 ms overlap with the previous span, to ensure that transmissions are fully contained in the span of some searching task (indeed, a transmission of the orange tag was partially contained in the first searching task). While the searching task was processed, new samples have arrived in the buffer and old ones discarded.

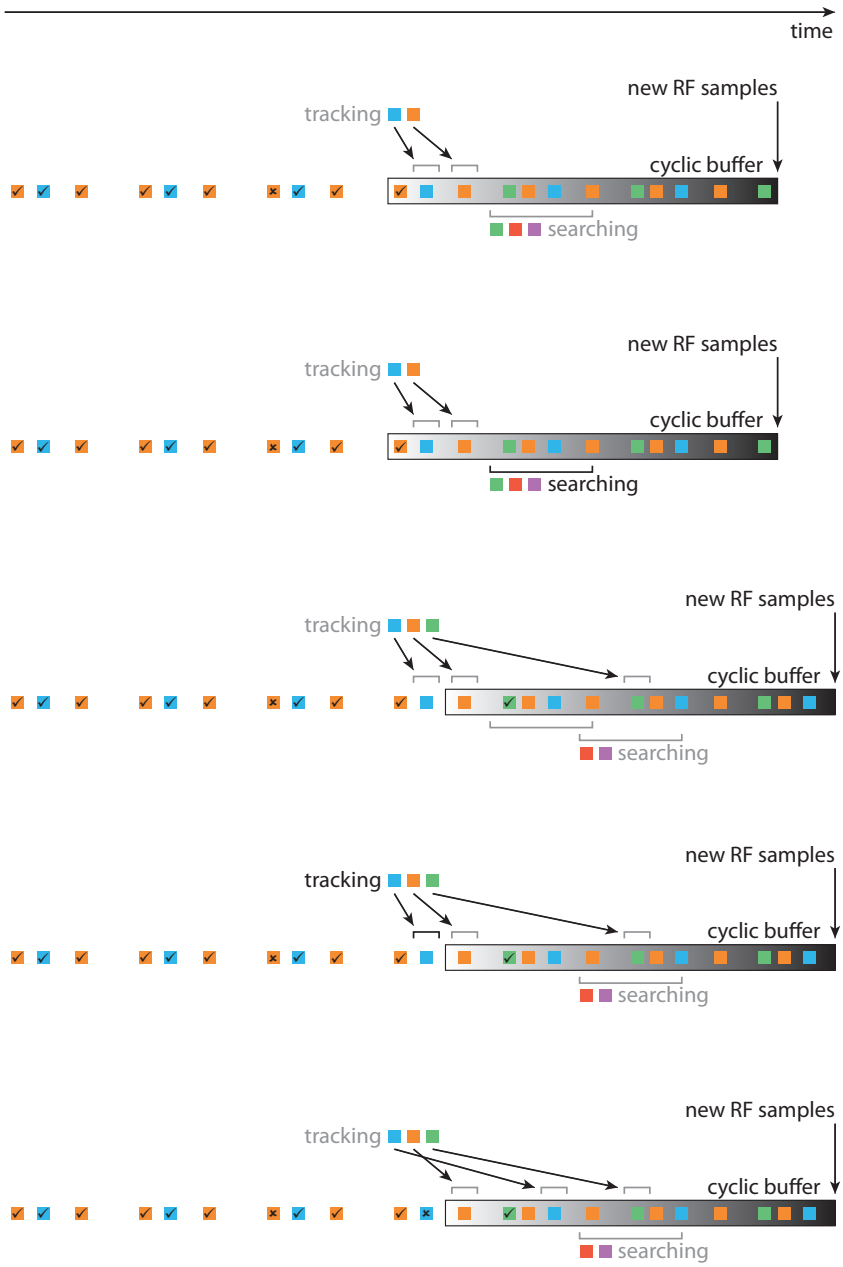

Figure 2.1: An illustration of the actions of the scheduler. Rows represent actions (steps) that the scheduler takes over time. Within each row, events are ordered from left to right. Each colored square represents a received transmission from a tag; the color represents the tag. The horizontal span of the square represents a span of time (samples) in which the transmission was received. See text for further explanation of the figure.

The fourth row indicates the next action of the scheduler. It decides to serve the tracking queue, and within that queue it processes the oldest transmission. In the figure, it is a transmission of the blue tag, which is no longer in the buffer. The bottom row shows the outcome of the scheduler's action: the missed transmission is marked as such (with an 'x') and the prediction of the next transmission of the blue tag is advanced by one inter-transmission period, to a transmission that is still in the buffer.

2.2 General-Purpose GPUs and CUDA

Graphics cards (GPUs) that can run general-purpose code, sometimes called GPGPUs, have emerged as effective accelerators of computationally-intensive tasks [9]. This thesis focuses on GPUs produced by the market leader, NVIDIA. NVIDIA GPUs contains a large number of simple cores (execution units) under the control of a smaller number of instruction schedulers. In the Jetson TX2 GPU, for example, 256 cores are organized into *warps* of 32 cores that are controlled by a single instruction scheduler. The warps are organized into *streaming multiprocessors* (SMs; two in the TX2). All the cores in a warp perform the same operation at the same time, so the code must exhibit a high degree of data parallelism. Larger NVIDIA GPUs use the same basic structure, but with different numbers of cores and SMs. Many NVIDIA GPUs can only operate directly on data stored in the GPUs memory, not in the computer's main memory. NVIDIA GPUs have a memory hierarchy that includes a small block of so-called *shared* memory that is private to an SM; on the TX2, its size is 64 KB. Data-movement engines called *copy engines* in the GPU move data between main memory and GPU memories and within GPU memories.

NVIDIA GPUs run programs written in CUDA, an extension of the C language. CUDA programs consist of host code that runs on the CPU and invokes *GPU kernels* to perform data-parallel operations. The following code allocates three arrays on the GPU, copies data from CPU memory into two of them, invokes a kernel to add them, and copies the result back to the CPU. As explained above, our code always separates the memory allocation from invocation of kernels, to reduce overhead. We also reduce data movement between CPU and GPU memory as much as possible, storing vectors that are reused, like transformed codes, in GPU memory.

```
int *d_a, *d_b, *d_c ;       // pointers to GPU memory
cudaMalloc( &d_a , nbytes ); // allocate GPU memory
cudaMalloc( &d_b , nbytes );
cudaMalloc( &d_c , nbytes );
```

```
cudaMemcpy( d_a , a , nbytes , cudaMemcpyHostToDevice ); //
   ↪ copy to device
cudaMemcpy( d_b , b , nbytes , cudaMemcpyHostToDevice
add<<<BLOCKS,THREADS>>>(d_a , d_b , d_c ); // for all i, d_c[i]
   ↪  = d_a[i] + d_[i]
cudaMemcpy( c , d_c , nbytes , cudaMemcpyDeviceToHost ); //
   ↪ copy back to host
```

The CUDA implementation of a kernel that adds two vectors is extremely simple. The kernel is an abstraction that tells the GPU how to operate on a single tuple of data. The kernel is expressed as a function (like `add` below) that is invoked on multiple tuples, potentially in parallel. The space of tuples that are operated upon is split into blocks, here using a one-dimensional partition. Each invocation can access its block index (here `blockIdx.x`) and its index within the block (`threadIdx.x`). These numbers allow the invocation to find the data that it needs to operate on.

```
__global__ void add(int *a, int *b, int *c) {
  int index = blockDim.x * blockIdx.x + threadIdx.x;
  c[index] = a[index] + b[index];
}
```

Reductions (summations, maximum value in an array, etc) are quite difficult to implement in CUDA. There are libraries that implement reductions, but the use of these libraries does not allow a data parallel operation to be fused with a reduction, which increases memory traffic. A C++ source library called CUB [13] simplifies the implementation of reductions and allow them to be fused with data parallel operations. CUB uses shared memory to achieve high performance; it requires the caller to allocate this memory, which our code does. Here is a simple reduction implemented using CUB; it is typical of reductions in our code.

```
__global__ void maxInt(int* input, int* out)
{
    typedef BlockReduce<int, THREADS> BlockReduceT;     // a
   ↪ CUB reducer

    __shared__ typename BlockReduceT::TempStorage temp; //
   ↪ shared memory for CUB

    int index = blockDim.x * blockIdx.x + threadIdx.x

    // CUB computes the reduction over threads in each block
    float block_max = BlockReduceT(temp_storage).Reduce(input[
   ↪ index], cub::Max());
```

```
    // reduction over the blocks using a CUDA atomic primitive
    if(threadIdx.x == 0) {
        atomicMax(out, block_max);
    }
}
```

Chapter 3

Signal Processing in ATLAS

The signal processing that RF samples associated with searching or tracking tasks is virtually identical. The samples undergo the mathematical transformations described below. However, the transformations are not applied naively, but in an optimized way described in Section 4. We focus for clarity only on FSK; signal processing for phase-shift keying (PSK) is described by Leshchenko and Toledo [12]. The mathematical transformations are:

1. Conversion of the complex RF samples, residing in the cyclic buffer and represented by pairs of 16-bit integers, to a single-precision (`float`) complex vector x.

2. The complex samples are usually multiplied element-wise by a complex input vector l representing a local-oscillator signal, to shift the center frequency so that the spectrum of transmissions is centered at zero. That is, we replace $x \leftarrow x \odot l$ (for all i, $x_i \leftarrow x_i \cdot l_i$).

3. Next, a bandpass FIR (finite impulse response) filter, represented here by a circulant matrix that H_{BP}, is applied, to produce $y \leftarrow H_{BP}x$. We use filters with 200 coefficients.

4. Two short (8 samples) matched filters are applied to y, one that represent a single-bit (chip) period at the frequency representing a 1 symbol and one that represent a single-bit period at the frequency that represents a 0 symbol. We denote their outputs by $f_1 = H_1 y$ and $f_0 = H_0 y$.

5. The vectors f_1 and f_0 are used to demodulate the transmission in two different ways, with and without normalization,

$$\begin{aligned} d &= (|f_1| - |f_0|) \oslash (|f_0| + |f_0|) \\ u &= |f_1| - |f_0| \end{aligned}$$

(elementwise absolute value, elementwise subtraction and addition, and elementwise division). These signals are real.

6. The algorithm applies exactly the same steps to a *replica* of the transmission we are trying to detect, a synthetic noise-free zero-padded signal $r^{(c)}$ that represents an FSK packet with the same modulation parameters and a pseudo-random bit sequence c. The resulting demodulated vector is denoted $d^{(c)}$; it is computed once and stored. The discrete signal $r^{(c)}$ is padded so that the length of $d^{(c)}$ is identical to the lengths of d and u.

7. We cross-correlate d with $d^{(c)}$. The cross correlation vector is also real.

8. We compute the value and location j of the maximum of the absolute value of the cross correlation vector,

$$j = \arg\max_i \left|\mathrm{xcorr}(d, d^{(c)})\right| .$$

The elements of $\mathrm{xcorr}(d, d^{(c)})$ around j are subsequently interpolated to estimate the arrival time of the incoming signal. We also compute quantities that are used to estimate the signal-to-noise ratio (SNR) and the power of the signal. More specifically, assuming that the nonzero part of $d^{(c)}$ spans its first n elements, we compute:

$$\begin{aligned} w_c &= \sum_{i=0}^{n} d_i^{(c)} d_{i+j} , \\ q &= \sum_{i=0}^{n} d_{i+j}^2 , \text{ and} \\ p_c &= \sum_{i=0}^{n} d_i^{(c)} u_{i+j} . \end{aligned}$$

For details on how power and SNR are estimated and how they are used, see [12, 15].

Chapter 4

High-Performance Design and Implementations

Our implementations of the computation described above are optimized. We use fast Fourier transforms (FFTs) to reduce the operation counts. We also use high-performance implementation principles in both the CPU implementation in C and in the GPU implementation in CUDA. In most cases the principles are applicable to both implementations; we highlight the differences when this is not the case.

4.1 Using Fast Fourier Transforms

We use techniques that minimize the operation counts that the signal-processing building-blocks perform.

In particular, we use the fast-Fourier transform (FFT) to compute cross-correlation and to apply FIR filters with many coefficients. To compute cross correlation, we use the identity $\mathrm{xcorr}(d, d^{(c)}) = \mathrm{ifft}(\mathrm{fft}(d) \odot \mathrm{fft}(d^{(c)}))$. We compose FIR filters that are applied in a sequence ($H_{BP}H_1$ and $H_{BP}H_0$) and we use FFTs to apply long FIR filters (filters with many coefficients). The formula is similar, except that the filters are natually expressed using a convolution, not a cross correlation.

We use the overlap-add method to apply medium-length filters and cross correlations. This reduces the operation count from $\Theta(m \log m)$ to $\Theta(m \log n)$ when applying a filter of length n to a block of m RF samples.

We pad input lengths to lengths that are a product of small integers, usually 2, 3, and 5; this ensures that applying FFT is as inexpensive as possible.

The actual data flow in the code is shown in Figure 4.1. Type conversion and demodulation is done once on the RF samples associated with each task, both searching and tracking. The demodulation takes advantage of the overlap-

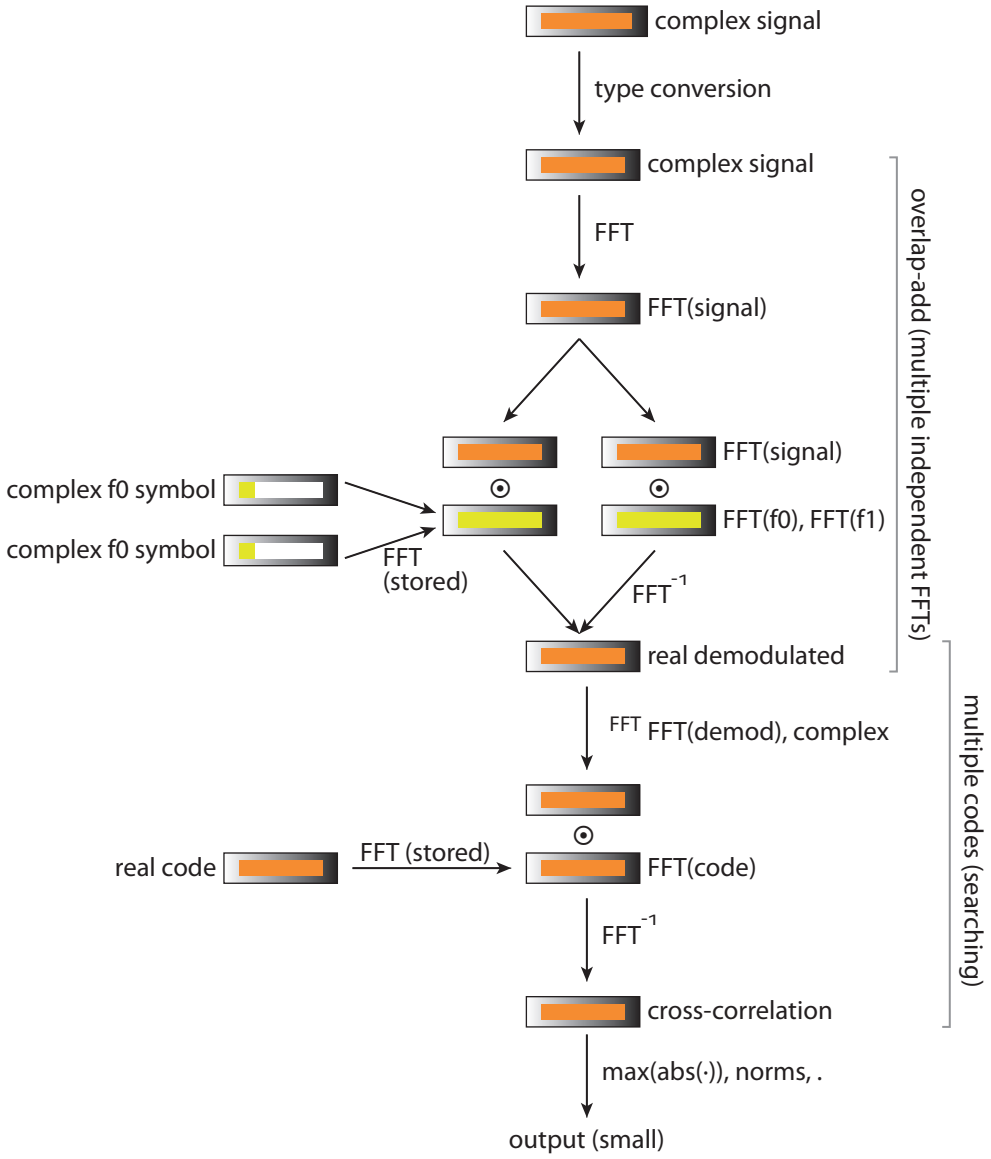

Figure 4.1: The signal processing data flow in ATLAS.

add method. The cross-correlation with the demodulated signal is done once in tracking tasks (with the code that we are tracking) and multiple times on the same demodulated signal in searching tasks, once per code.

4.2 Memorization and Planning

The implementations allocate arrays when they are needed and reuse them aggressively. In general, they are never released. For example, demodulation of a block of RF samples of a certain size is always done using the same temporary arrays; for other sizes, we use other arrays. This reduces memory-allocation overheads. Allocated arrays are aligned on cache-line boundaries in the CPU implementation and are allocated in GPU memory in the GPU implementation.

Auxiliary vector, like $\mathrm{fft}(d^{(c)})$, are computed when needed and stored indefinitely, to avoid recomputation.

Allocating auxiliary arrays of a given size and reusing them enables *preplanning* of all the FFT calls. We use comprehensive high-performance FFT libraries to compute FFTs. On CPUs, we use FFTW [7]; on GPUs, we use NVIDIA's cuFFT. Both FFTW and cuFFT requires calls to be *planned* in order to achieve high performance. An FFT plan refers to specific arrays at fixed (virtual) addresses, to allow optimization based on cache lines and similar hardware boundaries. Therefore, for every pair of arrays that are used as input and output of FFT calls, we plan the corresponding FFT operation and retain in memory both the plan and the input and output arrays.

4.3 Reducing Communication and Batching

Loops are aggressively fused in the CPU implementation and kernels are aggressively fused in the GPU implementation. This reduces data movement (e.g., cache misses) and allows elimination of some temporary arrays.

In the CPU implementation, we also batch cross correlation operations: a single call to FFTW computes many cross-correlation vectors. This exposes "embarrassing" parallelism (completely independent operations) that FFTW should be able to easily exploit, at least in principle. Batching is possible during extend-add operations on single vectors, and also when cross correlating a single demodulated signal with many codes.

4.4 CUDA Implementation

Here is how our CUDA host code implements the data type conversion and demodulation (the computation of d and u from the samples x). The code is slightly simplified, but not by much. The vectors `fd_f0` and `fd_f1` contain the transformed $H_{BP}H_0$ and $H_{BP}H_1$; they are preallocated, precomputed, and stored on GPU memory.

```
convertAndPad<<<BLOCKS,THREADS>>>(d_rf_samples, params,
   ↪ d_power_out, d_padded);

cufftExecC2C(plan_forward, d_padded, d_padded, CUFFT_FORWARD);

elementwiseMult<<<BLOCKS,THREADS>>>(d_padded, fd_f0, fd_f1,
   ↪ d_f0, d_f1);

cufftExecC2C(plan_backward, d_f0, d_f0, CUFFT_INVERSE);
cufftExecC2C(plan_backward, d_f1, d_f1, CUFFT_INVERSE);

elementwiseDemod<<<BLOCKS,THREADS>>>(d_f0, d_f1,
   ↪ demod_normalized, demod_unnormalized);
```

We use CUB to implement reductions, because it allows us to perform multiple reductions in one pass over the data and to fuse reductions with data parallel operations. We preallocate shared (fast) GPU memory for CUB.

In kernels that do not use CUB we do not use shared memory because they implement low data-reuse data-parallel operations over large vectors. The cuFFT library might also use shared memory, but if it does, it allocates it internally.

Chapter 5

Experimental Evaluation

This section presents our experimental evaluation of the effectiveness of GPUs for our task, in terms of both performance and energy efficiency.

5.1 Methodology (Test Data)

To test the codes, we modified the CPU-based DSP C code so that it stores all its inputs and outputs in files. We then ran the ATLAS base station code in an ad-hoc mode (that is, not as part of a localization system) on a computer connected to a USRP B210 sampling radio and configured the base station to detect a tag that was present in the room. This produced files that contained the RF samples that were processed in both searching and tracking mode, inputs that represent filter coefficients and the signal to correlate with, and the outputs of the signal-processing algorithms.

Next, we wrote a C program that reads these files, calls the signal processing routines on the recorded data, measures their running time and optionally the power consumption of the computer and its components, and stores the results in files. The program can use the recordings in both single-code single-RF-window mode and in batch mode that processes many codes in one call. The former is typical of tracking mode and the latter of searching mode. The program checks that the returned results are identical, up to numerical rounding errors, to those returned by the full base station run that detected the tag correctly. This ensures that all the results that we report represent correct executions of the algorithms. The code then stores the running times and the power measurements, if made, to log files.

We also tested that the new CUDA-based code works correctly when called from Java through the JNI interface and detects transmissions from tags and their arrival times. This test was performed on the Jetson TX2 computer de-

scribed below and the same URSP B210 radio.

5.2 Platforms

We evaluated the code on several platforms using both the CPU code and the GPU code.

Our baseline is a small form-factor desktop computer, representative of those currently used in ATLAS base stations, with an Intel i7-8700T CPU. This CPU has 6 physical cores running at clock frequencies between 2.4 and 4 GHz and thermal design power (TDP) of 35W. This CPU was launched in Q2 2018 and is fabricated in a 14 nm process. The computer ran Linux kernel version 5.3. We compiled the code using GCC version 7.5. Both our code and FFTW version 3.3.8 were compiled using the optimization options that are built into FFTW. The code that was produced ran slightly faster than code compiled with only `-O3 -mtune=native`.

Our main target is a low-power Jetson TX2 computer [2, 6], which has a 256-core NVIDIA Pascal GPU, four ARM Cortex-A57 cores and two ARM Denver2 cores, launched in Q2 2017 using a 16 nm process. The Cortex-A57 cores were designed by ARM and the Denver2 cores were designed by NVIDIA for higher single-threaded performance; both use the same 64-bit ARMv8 instruction set. It also has 8 GB of memory that both the CPU and GPU can access, with 59.7 GB/s memory bandwidth. The TX2 ran Linux kernel 4.9.140-tegra. We used nvcc version 10.0.326, CUDA library 10.0.130, gcc 7.4.0, and FFTW 3.5.7. CUB version 1.8.0 was used on all platforms.

We measured power consumption on the TX2 using two ina3221 current sensors built into the TX2 module and a third built into the motherboard [5]. Each sensor senses current on three different rails, and all the measurements are available by reading special files exposed by the driver under `/sys/bus/i2c/drivers/ina3221x`. The values that we report are the maximum value observed during the computation.

The power-vs-performance profile of the TX2 can be adjusted by turning cores on or off and by changing their clock frequency. NVIDIA defined several standard modes, which we use below in our tests. Table 5.1 describes these modes. The nominal TDP of the TX2 ranges from 7.5 W for the highest power efficiency mode, to 15 W for the highest performance modes. Both the TX2 module and the motherboards include power sensors that we use to measure the power consumption directly in our tests.

We also ran the GPU code on two additional platforms. One is an NVIDIA GeForce 1050 GTX GPU. This GPU uses the Pascal architecture, 640 cores running at 1.455 GHz, and 2 GB of RAM. The TDP is 75 W. It was plugged into a

Table 5.1: Standard power modes on the Jetson TX2.

Mode Name	Denver2 Cores	A57 Cores	GPU Frequency
Max-Q	—	4 × 1.2 GHz	0.85 GHz
Max-P All	2 × 1.4 GHz	4 × 1.4 GHz	1.12 GHz
Max-P ARM	—	4 × 2.0 GHz	1.12 GHz
Max-P Denver	2 × 2.0 GHz	—	1.12 GHz
Max-N	2 × 2.0 GHz	4 × 2.0 GHz	1.30 GHz

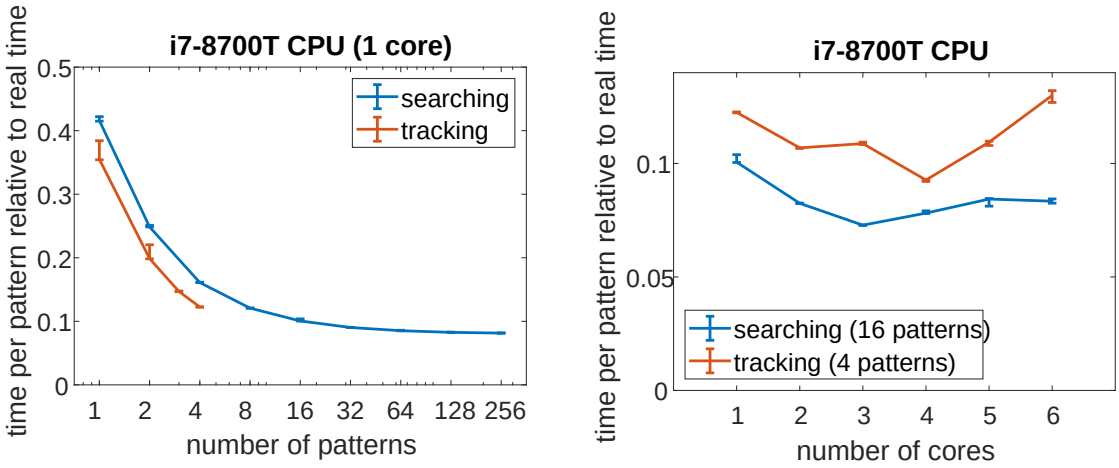

Figure 5.1: The performance of the DSP code on one CPU core (left) and its speedup on multiple cores. The vertical bars show the miniumum and maximum values over 10 experiments, and the actual data points are median results of the 10 experiments. The number of RF windows is 10 in the searching experiments and 100 in the searching experiments.

desktop running Windows 10 with a quad-core Intel i5-6500 CPU; we used CUDA 10.1, nvcc version 10.1.168, Microsoft's C++ compiler (cl) version 19.00.24210 for x64. The last GPU platform that we used is an NVIDIA Titan Xp GPU. This GPU also uses the Pascal architecture and has 3840 cores running at 1.582 GHz. It has 12 GB of memory and a high-bandwidth memory interface. The thermal design power is 250 W. It was plugged into a server with a 10-core Intel Xeon Silver 4114 CPU running Linux. We used CUDA and nvcc 10.0.130 and gcc 4.9.2.

5.3 Results

Figure 5.1 shows the performance of our C implementation on the baseline platform, which has an Intel i7-8700T CPU. We present the performance in terms of the ratio of processing time per pattern relative to the length of the RF window.

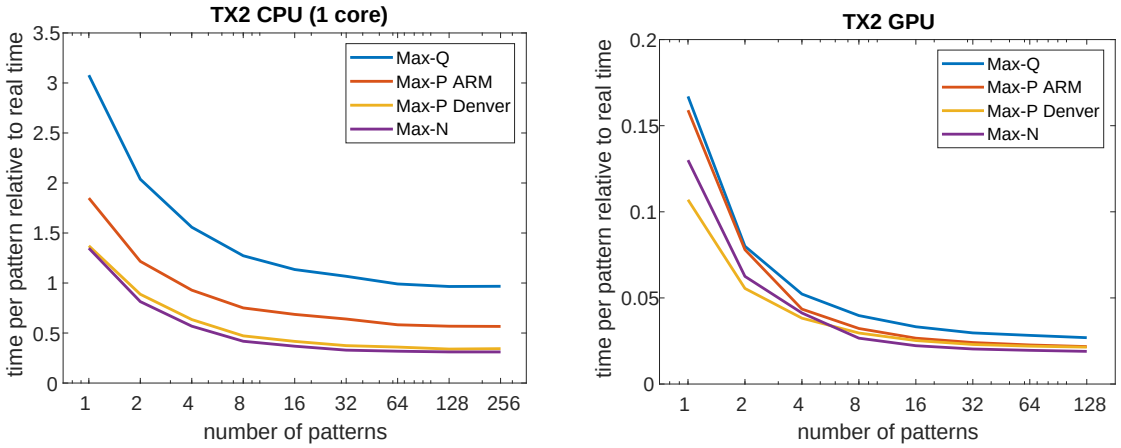

Figure 5.2: Searching performance on the Jetson TX2 on both the ARM cores (left) and the GPU (right) under four standard power configurations.

That is, if the code takes 1 s to process one 100-ms window of RF samples and to correlate the demodulated signal with 16 different code patterns, then we report the performance as $(1/16)/0.1 = 0.625$. A ratio of 1 implies that the base station can search for one tag continuously, that searching for 2 tags would drop 50% of the RF samples, and so on. A ratio of 0.1 implies that the station can search continuously for 10 tags without dropping any RF sample, and so on. Lower is better.

The results on one core (Figure 5.1 left) show that the performance per pattern improves significantly when we process multiple patterns in one window of RF samples (which is how the experiment was structured, since this is typical given how ATLAS systems are usually configured). This is mostly due to the amortization of the cost of demodulation over many patterns. The graph on the right in Figure 5.1 that using 2 or 3 cores improves performance relative to using only one core, but the improvement is far from dramatic or linear. Using 4 or more cores actually slows the code down relative to 2 or 3 cores. The parallelization in the CPU code is only within FFTW and it does not appear to be particularly effective in this code, perhaps due to the length of the FFTs.

Performance on the TX2 is excellent on the GPU but poor on the CPU, as shown in Figure 5.2. Our CUDA code running on the TX is about 4.3 times faster than the single-core i7 code and about 3 times faster than the i7 multicore runs. However, even at the highest performance mode, the TX's CPU cores perform about 4 times worse than the i7. We also measured the power consumption of the TX2 while it was running our code. The results, shown in Figure 5.3, indicate that when running the GPU code, the GPU is the largest power consumer, but the memory and other parts of the system-on-chip (most probably the memory

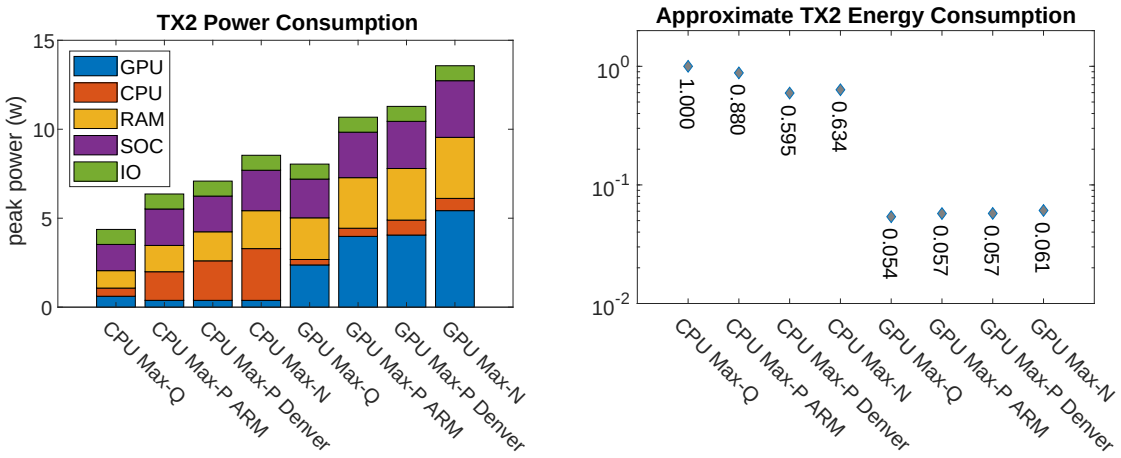

Figure 5.3: Power consumption during searching tasks (left), broken down by system component, and approximate total energy consumption during searching with 16 patterns, normalized to the largest energy expenditure (right). Both graphs show the data for searching on either the CPU or the GPU of the Jetson TX2 and under four standard power configurations. The rated accuracy of the power sensors is 2% for values above 200 mW and 15% for smaller values.

interface) consume a lot of power, about 50% of the total. The CPU and IO interfaces also consume power, but not much. In the C-code runs, the GPU is essentially off; the CPU, memory, and system-on-chip are the largest power consumers. The graph on the right in Figure 5.3 shows that the CUDA code is about 10 times more energy efficient than the C code running on the CPU, for the same task.

Figure 5.4 shows that our CUDA code is also very effective on desktop and server GPUs. A low-end GPU 12.8 times faster than a single x86_64 desktop core that is 2 years newer. A server GPU is 51.4 times faster than the desktop CPU.

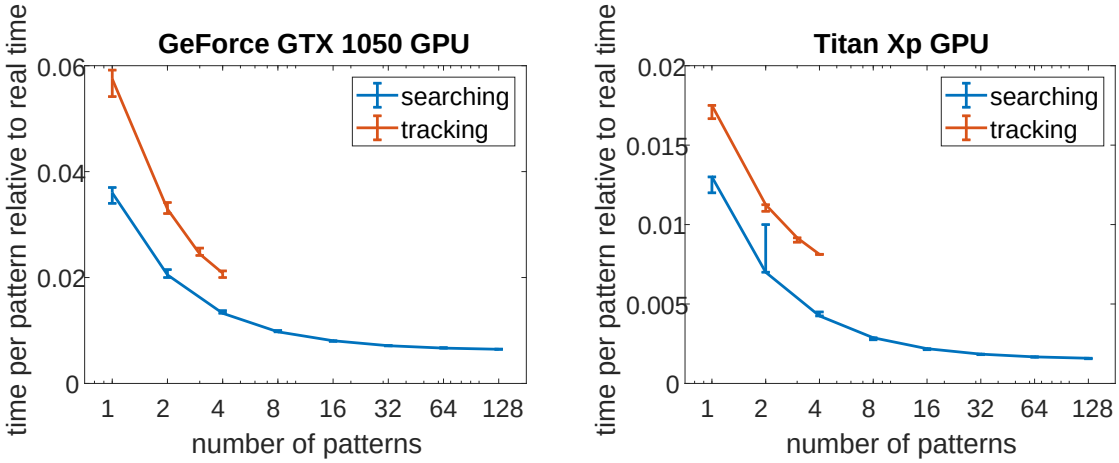

Figure 5.4: The performance of two desktop GPUs, a low-end (and somewhat old) GeForce GTX 1050 and a high-end Titan Xp.

Chapter 6

Deployment

Two base station (receivers) in the ATLAS system in the Hula valley in northern Israel have recently been equipped with computers with an NVIDIA GeForce GTX 1650 Super GPU, in order to improve the performance of the system, especially the searching-mode performance. Their GPUs have 1280 CUDA cores and they have 6-core Intel i5-10500 CPUs.

Figure 6.1 shows the searching performance of the base stations in the system over a 24-hour period when the system was attempting to track over 100 tags. The graph shows the fraction of RF samples processed while searching for tags. During the night (leftmost and rightmost parts of the graph) the percentage at base stations with GPUs is 113%. The value is over 100% because of overlaps in the searching periods, which add about 13% redundancy. The value of 113% implies that all the RF samples were searched for all the tags in the searching

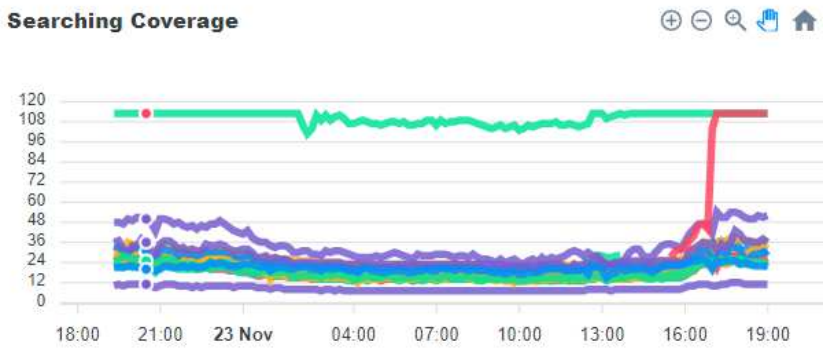

Figure 6.1: The searching performance of base stations in the ATLAS system in the Hula Valley during a 24-hour period. The graph is a screenshot from a web application that is used to monitor the system. Times are UTC and local time is 2 hours later.

queue.

We see in the graph that during the day, between about 02:00 UTC and 14:00 UTC, the searching performance of all the base station dropped. This happened because most of the tags were attached to bats, so during the night many of them were in the tracking queue and the searching queue was emptier; during the day, almost all the tags were in the searching queue. At about 17:00 UTC the second GPU-equipped base station was turned on; its performance is shown by the red curve.

In a separate experiment, we configured the system to track 115 tags, none of which were tracked. The searching coverage in a GPU-equipped base station was 87%. When we configured the base station to use the old CPU code, performance dropped to 20%.

Chapter 7

Related Work

Alawieh et al. [1] and Hendricks et al. [10] compare the performance of several types of compute nodes, including GPUs, CPUs, and FPGAs, in the context of RF ToA estimation, with application to a location estimation system called RedFIR. Their requirements are more demanding, in the sense that RedFIR requires real-time processing of a stream of samples, whereas we buffer samples for a few seconds and use a priority scheduler to simplify the signal-processing code. It is well known that real-time scheduling on GPUs is challenging [22]; our scheduler allows ATLAS to avoid the difficulty. Also, RedFIR does not rely on periodic transmit schedules, whereas ATLAS reduces the computational load by tracking tags rather than just searching for them. Finally, signal processing in RedFIR is a bit simpler than in ATLAS because they use PSK transmitters, not FSK transmitters. Belloch et al. [3] and Kim et al. [11] present acoustic localization systems that exploit GPUs for ToA estimation.

Our use of cuFFT follows the advice of Střelák and Filipovič [16, Section 2.5]. Other CUDA FFT libraries [8, 14] appear to be no longer maintained.

Chapter 8

Discussion and Conclusions

We have shown that by implementing the DSP functionality of an RF time-of-arrival transmitter localization system in CUDA, we can improve the acquisition (searching) throughput of the system by a factor of 4 while reducing power consumption by a factor of 5 or so relative to a baseline single-core C code, even though the C code has been carefully optimized. Table 8.1, which summarizes the characteristics of our test platforms (as well as of a few newer platforms) show that higher-end GPUs can improve throughput dramatically higher, at the cost of higher power consumption, and sometimes also higher cost. The throughput of tracking modes also improves on GPU platforms.

Our baseline code does not effectively exploit multicore CPU platforms, even

Table 8.1: A comparison of GPU platforms. Column 3 shows the number CPU and GPU cores. The 5th column shows the TDP of the platform, either the overall power consumption or, if marked by a +, of only the device itself. The cost in USD is only indicative, and again shows either the total system cost or, when marked by a +, the cost of the device. The rightmost column shows the throughput, defined as the number of codes (tags) that can be searched for without dropping any RF samples, assuming batches of 128 and windows of 100 ms each; this also assumes that only 50% of the time is devoted to searching, the rest to tracking. The performance of the i7 processor assumes that only one of the six cores are used.

Device	Launch	Cores	Fab	W	USD	tput
i7-8700T	Q2 2018	6 × x86	14 nm	35^{+}	1000	6
Jetson TX2	Q2 2017	6 × ARM +256	16 nm	7.5–15	1000	26
GeForce GTX 1050	Q2 2016	+384	14 nm	75^{+}	110^{+}	77
Titan Xp	Q2 2017	+3840	16 nm	250^{+}	1200^{+}	315

though it relies heavily on a (high-quality) parallel multicore FFT library; this alone does not deliver good parallel speedups, perhaps due to the modest size of the tasks. It is likely that a careful parallel multicore implementation, perhaps in OpenMP, can improve the performance of the C code on multicore CPUs. However, this would entail programming that is at least as complex as our CUDA implementation, and it would still not attain the maximum performance of the GPU code or its power-performance ratio.

This code is now mature and in production. The entire signal-processing software library that ATLAS uses has been converted to CUDA (including the PSK code). We will continue to maintain both versions and users can switch between them easily at run time. Two base stations with GPUs have already been deployed and they speed up the acquisition time of the system. When all the base stations in a system have GPUs, tracking capacity will also increase significantly. We plan to test and deploy base station computers based on the NVIDIA Xavier AGX and/or Xavier NX development kits (512 or 384 CUDA Volta cores, respectively, and only up to 30W).

Acknowledgments

Thanks to NVIDIA Corporation for the donation of the Jetson TX2. This study was also supported by grants 965/15, 863/15, and 1919/19 from the Israel Science Foundation.

Bibliography

[1] Mohammad Alawieh, Maximilian Kasparek, Norbert Franke, and Jochen Hupfer. A high performance FPGA-GPU-CPU platform for a real-time locating system. In *Proceedings of the 23rd European Signal Processing Conference (EUSIPCO)*, pages 1576–1580, Aug 2015.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

[3] Jose A. Belloch, Alberto Gonzalez, Antonio M. Vidal, and Maximo Cobos. On the performance of multi-GPU-based expert systems for acoustic localization involving massive microphone arrays. *Expert Systems with Applications*, 42:5607–5620, 2015.

[4] Ammon Corl, Motti Charter, Gabe Rozman, Sivan Toledo, Sondra Turjeman, Pauline L. Kamath, Wayne M. Getz, Ran Nathan, and Rauri C. K. Bowie. Movement ecology and sex are linked to barn owl microbial community composition. *Molecular Ecology*, 20(7):1358–1371, 2020.

[5] NVIDIA Corporation. *NVIDIA Jetson Linux Developer Guide*, July 2020. 32.4.3 Release.

[6] Dustin Franklin. NVIDIA Jetson TX2 delivers twice the intelligence to the edge, March 2017. NVIDIA Developer Blog, `https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge`.

[7] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[8] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete Fourier transforms on graph-

ics processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 1–12, Nov 2008.

[9] Samuel Greengard. GPUs reshape computing. *Communication of the ACM*, 59(9):14–16, August 2016.

[10] Arne Hendricks, Thomas Heller, Andreas Schäfer, Max Kasparek, and Dietmar Fey. Evaluating performance and energy-efficiency of a parallel signal correlation algorithm on current multi and manycore architectures. *Procedia Computer Science*, 80:1566–1576, 2016.

[11] Seongseop Kim, Jeonghun Cho, and Daejin Park. Moving-target position estimation using GPU-based particle filter for IoT sensing applications. *Applied Sciences*, 7(11), 2017.

[12] Andrey Leshchenko and Sivan Toledo. Modulation and signal-processing tradeoffs for reverse-GPS wildlife localization systems. In *Proceedings of the European Navigation Conference (ENC)*, pages 154–165, 2018.

[13] Duane Merrill. Cub (cuda unbound) library version 1.8.0, 2018. A library of CUDA collective primitives; Available online at `https://nvlabs.github.io/cub/`.

[14] Sayantan Mitra and Ashok Srinivasan. Small discrete Fourier transforms on GPUs. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 33–42, May 2011.

[15] Yaniv Rubinpur and Sivan Toledo. High-performance gpu and cpu signal processing for a reverse-gps wildlife tracking system, 2020. `https://arxiv.org/abs/2005.10445`, to appear in the Proceedings of HeteroPar 2020.

[16] David Střelák and Jiří Filipovič. Performance analysis and autotuning setup of the cuFFT library. In *Proceedings of the 2nd Workshop on Autotuning and Adaptivity Approaches for Energy-Efficient HPC Systems (ANDARE)*. ACM, 2018. 6 pages.

[17] Sivan Toledo, Oren Kishon, Yotam Orchan, Yoav Bartan, Nir Sapir, Yoni Vortman, and Ran Nathan. Lightweight low-cost wildlife tracking tags using integrated tranceivers. In *Proceeings of the 6th Annual European Embedded Design in Education and Research Conference (EDERC)*, pages 287–291, Milano, Italy, September 2014.

[18] Sivan Toledo, Oren Kishon, Yotam Orchan, Adi Shohat, and Ran Nathan. Lessons and experiences from the design, implementation, and deployment of a wildlife tracking system. In *Proceeings of the IEEE International Conference on Software Science, Technology and Engineering (SWSTE)*, pages 51–60, Beer Sheva, Israel, June 2016.

[19] Sivan Toledo, Yotam Orchan, David Shohami, Motti Charter, and Ran Nathan. Physical-layer protocols for lightweight wildlife tags with Internet-of-things transceivers. In *Proceedings of the 19th IEEE International Symposium on a Wolrd of Wireless, Mobile, and Multimedia Networks (WOWMOM)*, pages 1–4, June 2018.

[20] Sivan Toledo, David Shohami, Ingo Schiffner, Emmanuel Lourie, Yotam Orchan, Yoav Bartan, and Ran Nathan. Cognitive map-based navigation in wild bats revealed by a new high-throughput tracking system. *Science*, 369(6500):188–193, 2020.

[21] Adi Weller-Weiser, Yotam Orchan, Ran Nathan, Motti Charter Anthony J. Weiss, and Sivan Toledo. Characterizing the accuracy of a self-synchronized reverse-GPS wildlife localization system. In *Proceeings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12, Vienna, Austria, April 2016.

[22] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 20:1–20:21, 2018.

תקציר

בעבודה זו אנו מציגים מימוש בעל תפוקה גבוהה של אלגוריתמי עיבוד האותות של מערכת איכון לחיות בר שנקראת אטלס. מערכת זו מנטרת את מיקומם של משדרי רדיו שמחוברים לחיות בר ע"י הערכת זמן הגעת האות מהמשדר למספר תחנות בסיס שמיקומם ידוע מראש. חישובים אילו דורשים כח חישוב גדול, במיוחד כשזמן שידור האות מהמשדר אינו ידוע (אפילו לא בקירוב). חישובים אילו הם צוואר הבקבוק שמגביל את מספר האיכונים שהמערכת יכולה לבצע ביחידת זמן. בתזה זו אנו מראים שני מימושים של אלגוריתמי עיבוד האותות של אטלס. מימוש ראשון מיועד למעבד והשני ליחידת עיבוד גרפי. כמו כן אנו מראים את תוצאות מדידות הביצועים שנערכו לכל אחד מהמימושים. התוצאות מראות שהרצת האלגוריתם על יחידת עיבוד גרפית משפרת את הביצועים ומקטינה את צריכת ההספק יחסית למימוש שרץ על השרתים הטיפוסיים שמריצים את אטלס (שרתים עם מעבדים חזקים). השיפור בביצועים הוא פי 50 כשמריצים את הקוד על כרטיס גרפי עוצמתי ופי 4 כשמריצים את הקוד על פלטפורמה המכילה יחידת עיבוד גרפי, ובעלת דרישת ההספק קטנה פי 5 מזה של שרת טיפוסי. כלומר יש שיפור בכמות משדרי הרדיו עליהם ניתן לעקוב לאותו הספק, וגם שיפור לעומת מחיר החומרה.

אוניברסיטת תל-אביב
הפקולטה למדעים מדויקים על שם ריימונד ובברלי סאקלר
בית הספר למדעי המחשב על שם בלווטניק

עיבוד אותות למערכת איכון חיות בר בקצב גבוה על גבי מעבדי CPU ו-GPU

התזה מוגשת כחלק מהדרישות לקבלת תואר שני
מאוניברסיטת תל-אביב ע"י

יניב רובינפור

המחקר לתזה זו בוצע באוניברסיטת תל-אביב תחת הנחייתו
של פרופ' סיון טולדו
ינואר2021