Contracting-grid Search Algorithm on CPU, GPU and FPGA

Luca Caucci
caucci@email.arizona.edu

Abstract

These notes summarize my efforts in implementing on different platforms the contracting-grid search algorithm for maximum-likelihood estimation of 2D position of interaction from raw photomultiplier tube (PMT) signals. The same algorithm was implemented for CPUs, GPUs (with NVIDIA's CUDA®) and FPGAs (using Xilinx® Vivado HLS). Many concepts and tricks I learned along the way are summarized in this document. The present discussion assumes familiarity with C++ concepts and a rudimentary understanding of how GPUs and FPGAs work.

1 Introduction

A typical gamma-ray camera [1] used in medical imaging consists of an array of photomultiplier tube whose voltage outputs are digitized and processed to estimate various parameters about the interaction between incoming gamma-ray photons and the camera's crystal [4]. Parameters we can estimate include photon energy and 2D or 3D position. Estimation typically makes use of calibration data (in the from of mean detector response function, or MDRF) and a statistical model that depends on the photon parameters we want to estimate.

In this work, we consider a maximum-likelihood approach for the estimation of the 2D position of interaction from raw PMT data $\boldsymbol{g} = \{g_1, \ldots, g_K\}$ (with $K = 9$) according to [3]:

$$\hat{\boldsymbol{r}}_{\mathrm{ML}} = \underset{\boldsymbol{r} \in D}{\arg\max} \big[\log L(\boldsymbol{g}; \boldsymbol{r}) \big],$$

in which D denotes the search space (the detector face), $\arg\max_{\boldsymbol{r} \in D}[\ldots]$ denotes the value of $\boldsymbol{r} = (x, y) \in D$ that maximizes the quantity $[\ldots]$ and $L(\boldsymbol{g}; \boldsymbol{r})$ denotes the likelihood of $\boldsymbol{g}$ given $\boldsymbol{r}$. For Poisson statistics, we have

$$L(\boldsymbol{g}; \boldsymbol{r}) = \prod_{k=1}^{K} \frac{[\overline{g}_k(\boldsymbol{r})]^{g_k}}{g_k!} \exp\big[- \overline{g}_k(\boldsymbol{r}) \big].$$

Notice that

$$\log L(\boldsymbol{g}; \boldsymbol{r}) = \sum_{k=1}^{K} \big[g_k \log \overline{g}_k(\boldsymbol{r}) - \overline{g}_k(\boldsymbol{r}) \big] - \sum_{k=1}^{K} \log(g_k!).$$

The quantity $\sum_{k=1}^{K} \log(g_k!)$ does not depend on $\boldsymbol{r}$ and it can be ignored during the search for the maximum. Moreover, for any integer n we can calculate $\log(n!)$ according to $\log(n!) = \log \Gamma(n+1)$, in which $\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} \, \mathrm{d}t$ is the Gamma function. Many programming languages provide a function to calculate $\log \Gamma(x)$ directly and without having to calculate $\Gamma(x)$ first. This avoids potential numerical overflows in the calculation of $\Gamma(x)$ for modestly-large values of x.

For reasons that will become clear later, it is useful to define

$$\log L'(\boldsymbol{g}; \boldsymbol{r}) = \sum_{k=1}^{K} \big[g_k \log \overline{g}_k(\boldsymbol{r}) - \overline{g}_k(\boldsymbol{r}) \big],$$

so that $\log L(\boldsymbol{g}; \boldsymbol{r}) = \log L'(\boldsymbol{g}; \boldsymbol{r}) - \sum_{k=1}^{K} \log(g_k!)$.

A particular search algorithm (to be discussed below) was implemented for the $\arg\max$ search. The same algorithm was implemented for three different computing architectures:

- a general-purpose central processing unit (CPU);
- a graphics processing unit (GPU);
- a field-programmable gate array (FPGA).

The code and sample data are available on Github at `https://github.com/caucci/get_data_proj`.

2 The Contracting-grid Search Algorithm

For fixed $\boldsymbol{g}$, the quantity $L(\boldsymbol{g};\boldsymbol{r})$ defined above is often a smooth function of $\boldsymbol{r}$. This suggest a coarse-to-fine approach to find the maximum of $L(\boldsymbol{g};\boldsymbol{r})$ over D. In this approach (referred to as "contracting-grid search", [3]), $L(\boldsymbol{g};\boldsymbol{r})$ is first evaluated for $\boldsymbol{r}$ over a $T \times T$ grid (in the case of a 2D estimation) of points that extends over the whole search space. Typically, T is between 3 and 10. The point of the grid that maximizes $L(\boldsymbol{g};\boldsymbol{r})$ becomes the center of the next $T \times T$ grid, which covers a smaller region of the search space. The grid spacing for this new grid is $1/\alpha$ (with $\alpha > 1$) times the grid spacing of the current grid. This process repeats for a fixed number of iterations. At the last iteration, the point of the grid that maximizes $L(\boldsymbol{g};\boldsymbol{r})$ is taken as the maximum-likelihood estimate $\hat{\boldsymbol{r}}_{\mathrm{ML}}$. Some care must be taken in the process above when $\boldsymbol{r}$ falls outside the search space D. For example, we can assume $L(\boldsymbol{g};\boldsymbol{r}) = -\infty$ if $\boldsymbol{r} \notin D$.

3 Spline functions

The contracting-grid search algorithm discussed above assumes that $L(\boldsymbol{g};\boldsymbol{r})$ can be evaluated at any $\boldsymbol{r} \in D$. This means that MDRF data $\overline{g}_k(\boldsymbol{r})$ for $k = 1, \ldots, K$ must be available for any $\boldsymbol{r} \in D$, which is rarely the case. In fact, the MDRF is typically available as collection of samples on a uniform lattice in D. In this work, we use numerical approximation with spline functions to obtain $\overline{g}_k(\boldsymbol{r})$ for any $\boldsymbol{r} \in D$. A brief introduction to spline functions is reported below.

Given integers $M > 0$ and $K > 0$ and a set of $2M+K$ non-decreasing real numbers (called "knots") $t_1, \ldots, t_{2M+K}$, a vector space of dimension $M+K$ is constructed from the basis set of functions $\{B_{1,M}(x), \ldots, B_{M+K,M}(x)\}$, in which $x \in [a, b]$, for $a = t_M$ and $b = t_{M+K+1}$. Basis vectors $B_{i,M}(x)$ are defined recursively as [5]

$$B_{i,1}(x) = \begin{cases} 1 & \text{if } x \in [t_i, t_{i+1}), \\ 0 & \text{otherwise} \end{cases}$$

and

$$B_{i,m}(x) = \frac{x - t_i}{t_{i+m-1} - t_i} B_{i,m-1}(x) + \frac{t_{i+m} - x}{t_{i+m} - t_{i+1}} B_{i+1,m-1}(x),$$

for $m = 2, \ldots, M$. It is possible to show that for any $x \in [a, b]$, $\sum_{i=1}^{M+K} B_{i,M}(x) = 1$ and that if $x \in [t_i, t_{i+1})$, then $B_{\ell,m}(x) > 0$ only for $\ell = i - m + 1, \ldots, i$; $B_{\ell,m}(x) = 0$ for all the other values of ℓ.

Given coefficients $c_1, \ldots, c_{M+K}$, we can consider the function

$$s(x) = \sum_{i=1}^{M+K} c_i B_{i,M}(x),$$

which we will refer to as "spline function" or "spline". Evaluation of $s(x)$ requires evaluation of at most M basis functions $B_{i,M}(x)$ according to the recursive formula above. Finally ND spline functions can be defined from a tensor product of 1D basis functions. For example, a 2D spline $s(x, y)$ takes the form

$$s(x, y) = \sum_{i=1}^{M_x+K_x} \sum_{j=1}^{M_y+K_y} c_{i,j} B^{(x)}_{i,M_x}(x) B^{(y)}_{j,M_y}(y).$$

In this work, we used least-square fitting of MDRF data with 2D spline functions defined over $[0, 1]^2$. Knots corresponding to the x and y variables were chosen uniformly spaced in $[0, 1]$. In other words, $t_M = 0$ and $t_{M+K+1} = 1$. All the other t_i's can be calculated on-the-fly in the recursive formula for $B_{i,m+1}(x)$ according to $t_i = \delta(i - m)$ for $\delta = (K + 1)^{-1}$. If $x \in [0, 1)$, then $x \in [t_i, t_{i+1})$ provided that $i = \lfloor x(K + 1) \rfloor + m$ (special handling is needed if $x = 1$). This choice of knots results in a very efficient evaluation of spline functions.

C++ templated classes were written to implement 1D, 2D and 3D spline functions. For example, the definition of the class `spline_1D` looks like:

```
template<class _V, class _C, int _M, int _K> class spline_1D {
  public:
    spline_1D();
    spline_1D(const _V my_coefs[_M + _K]);
```

```
    _V operator()(const _C & x) const;
    void get_coefs(_V output[_M + _K]) const;

  private:
    _V coefs[_M + _K];
};
```

Type `_C` corresponds to the type of the variable x, while type `_V` is the type of the value $s(x)$. This design provides great flexibility as long as the standard mathematical operators for and between types `_C` and `_V` are defined. For example, `_V` could be defined to store MDRF values for all K PMTs. That way, the evaluation of $\overline{\boldsymbol{g}}(x,y) = \{\overline{g}_1(x,y), \ldots, \overline{g}_K(x,y)\}$ would only require calculation of M_x spline basis functions in x and M_y spline basis functions in y. Finally, the definition of `operator()` adds clarity to the code by allowing evaluation of a spline function as if it were a mathematical function.

As a fun exercise, development and testing of the spline classes included the development of a class to store a fraction represented as $\frac{n}{d}$, in which n and d were 1024-bit integer numbers with no common factors. In the testing of `spline_1D`, type `_C` was take as this fraction type, while type `_V` was defined as the complex numbers type over `_C`.

4 CPU Implementation

The CPU implementation follows the basic description of the contracting-grid algorithm. Raw PMT data are first corrected for PMT gain and stored in a temporary array:

```
for(pmt = 0; pmt < NUM_PMTS; ++pmt) {
  tmp_data[pmt] = PMT_data[event_index].val[pmt] / calibr_funct.gain[pmt];
}
```

Contracting grid variables are initialized (the estimation assumes first $D = [0,1]^2$) and the iterative process is set up:

```
current_x = current_y = float(1) / float(2);
step = (float(1) - float(0)) / float(SIZE_CONTR_GRID);
for(iter = 0; iter < NUM_CONTR_GRID_ITER; ++iter) {
  ...
  ...
  ...
}
```

Inside the for loop above, the values of $\log L'(\boldsymbol{g};\boldsymbol{r})$ for all points of the grid are calculated

```
for(index_x = 0; index_x < SIZE_CONTR_GRID; ++index_x) {
  test_x = current_x + (float(index_x) - (float(SIZE_CONTR_GRID - 1) / 2.00f)) * step;
  for(index_y = 0; index_y < SIZE_CONTR_GRID; ++index_y) {
    test_y = current_y + (float(index_y) - (float(SIZE_CONTR_GRID - 1) / 2.00f)) * step;
    ...
    ...
    ...
  }
}
```

The calculation of $\log L'(\boldsymbol{g};\boldsymbol{r})$ avoids "not-a-number" situations by avoiding multiplying a non-zero number by $\log(0)$:

```
log_like = float(0);
for(pmt = 0; pmt < NUM_PMTS; ++pmt) {
  camera_MDRF = calibr_funct.mdrf[pmt](test_x, test_y);
  if((tmp_data[pmt] != float(0)) || (camera_MDRF != float(0))) {
    log_like += tmp_data[pmt] * std::log(camera_MDRF) - camera_MDRF;
  }
}
```

Once the values of $\log L'(\boldsymbol{g};\boldsymbol{r})$ are calculated for each point on the grid, the indices of the maximum are found and stored in the variables `max_index_x` and `max_index_y`. The new grid center is calculated and the grid step is adjusted:

```
current_x = current_x + (float(max_index_x) - (float(SIZE_CONTR_GRID - 1) / 2.00f)) * step;
current_y = current_y + (float(max_index_y) - (float(SIZE_CONTR_GRID - 1) / 2.00f)) * step;
step /= CONTR_FACTOR;
```

At the end of the contracting grid process, likelihood thresholding is applied to the estimate. To that end, the previously-ignored term $\sum_{k=1}^{K} \log(g_k!)$ is accounted for and $\log L(\boldsymbol{g};\boldsymbol{r})$ is compared to a position-depended threshold:

```
log_like = max_log_like;
for(pmt = 0; pmt < NUM_PMTS; ++pmt) {
  if(tmp_data[pmt] > float(0)) {
    log_like -= std::lgamma(tmp_data[pmt] + float(1));
  }
}
estim_event[event_index].valid = log_like > calibr_funct.thresh(current_x, current_y);
```

As a final step, the estimated position is scaled to the detector physical size and stored to memory:

```
estim_event[event_index].x_pos = CAMERA_MIN_POS + current_x * (CAMERA_MAX_POS - CAMERA_MIN_POS);
estim_event[event_index].y_pos = CAMERA_MIN_POS + current_y * (CAMERA_MAX_POS - CAMERA_MIN_POS);
```

5 GPU Implementation

A typical GPU code includes one or more kernels, which can be described as pieces of code executed in parallel by the GPU device as threads. Threads are organized in blocks (of dimensionality from 1 to 3) and blocks form a grid (also of dimensionality from 1 to 3). Hence, a hierarchy of threads exists in a GPU device: a grid is made of blocks and each block is made of threads. Built-in variables are available inside a kernel to retrieve block and thread indeces along their three dimensions. The programming tool of choice for the GPU implementation is CUDA®. A special programming construct is used by the CUDA® programmer to create a grid of blocks and to populate each block with threads running on the GPU. Being an extension of the C++ programming language, CUDA® allows classes and objects being used inside a kernel.

The GPU implementation of the contracting-grid algorithm [2] is based on its CPU version. Notable differences are:

- a 1D grid of blocks is used and the block index along the first dimension is used as the event index;
- a 2D thread block of threads of the same size as the contracting grid is used to calculate $\log L'(\boldsymbol{g};\boldsymbol{r})$ in parallel for each point of the contracting grid;
- the implementation uses shared memory to share among the threads in the same block the gain-corrected PMT values and the values of $\log L'(\boldsymbol{g};\boldsymbol{r})$ for each point in the contracting grid.

The implementation allocates in shared memory only the following variables:

```
__shared__ float log_like_values[SIZE_CONTR_GRID][SIZE_CONTR_GRID];
__shared__ float tmp_data[NUM_PMTS];
```

For example, it is not necessary that threads in the same block share the center of the contracting grid (variables `current_x` and `current_y` in the code) because these two variables are initialized to $1/2$ at the beginning of the search and they are updated deterministically and based on the same data (i.e., the content of the 2D array `log_like_values`) shared among threads in the same block.

Towards the end of the kernel code, one of the threads (conventionally thread 0) in each block calculates the previously-ignored term $\sum_{k=1}^{K}\log(g_k!)$ and compares the value of $\log L(\boldsymbol{g};\boldsymbol{r})$ to the position-depended likelihood threshold.

Minor modifications to the spline library were needed so that spline objects could be used inside kernels.

6 FPGA Implementation

The following text was reproduced with minor modifications from the Xilinx® website:

> *Advanced algorithms used today in wireless, medical, defense, and consumer applications are more sophisticated than ever before. Vivado® High-Level Synthesis (HLS) accelerates IP creation by enabling C and C++ specifications to be directly targeted into Xilinx programmable devices without the need to manually create register-transfer logic (RTL). Vivado HLS provides abstraction of algorithmic description, data type specification (integer, fixed-point or floating-point) and interfaces. Vivado HLS accelerates verification using C/C++ test bench simulation.*

6.1 Vivado HLS and Test Benching

Even for the most elementary FPGA design, synthesis of register-transfer logic from a high-level description can take hours to complete. To reduce development time, Xilinx® included in Vivado HLS a simulation step in which the FPGA design is run through a CPU-based simulation of the FPGA design. Because no synthesis is required, the correctness of an FPGA design can be verified in a matter of minutes (or less) on a small dataset. A test bench functionality is available wherein the output of the simulated FPGA design is compared against a known correct output. In Vivado HLS, the code to be synthesized is implemented as a function called the "top function". The top function can call other functions, which will be included in the synthesis as well. Running the simulation of the FPGA design can be thought of as calling the top function as if it were a regular CPU-based function.

Vivado HLS includes a command-line interface (CLI), which is very useful when Vivado HLS is used remotely. Moreover, a tool command language (TCL) file can be used to collect Vivado HLS CLI commands in a script file so that many repetitive steps (including set up a project, add test bench files, specify compilation flags, select the top functions, etc.) can be listed in a single TCL file and perform as a batch job.

6.2 Ports in the Design Interface

Contrary to C++ design, in which all input and output operations to functions are performed through formal function arguments, in an RTL design these same input and output operations must be performed through ports in the design interface. These ports typically operate using a specific I/O protocol. Ports can be specified in an RTL design using `#pragma` compiler directives. In this project, a Xilinx® Alveo™ U250 Data Center accelerator card with a 16-lane PCI Express edge connector was used. The following list of facts was learned from various sources:

- the return type for the top function must be `void` and `extern "C"` must be added in front of the declaration;
- access to arrays requires the creation of an AXI4 interface (`m_axi`) for each memory buffer;
- each function parameter must have an AXI4-Lite (`s_axilite`) interface associated to it;
- an additional AXI4-Lite interface must be defined for the `return` statement;
- each element of an array must have a size in bits that is a power of 2 (data padding might be required);
- there is limited support for `structs` containing `structs` in an AXI4 interface;
- if a `struct` containing arrays is passed by value, the arrays within the `struct` are replaced by pointers.

As a reference, the signature for the top function is:

```
extern "C" void contr_grid_kernel(estim_event_t *estim_event_dev, PMT_data_t *PMT_data_dev,
  unsigned int num_events, float mdrf_spline_coefs_dev[(MY + KY) * (MX + KX) * NUM_PMTS],
  float thresh_spline_coefs_dev[(MY + KY) * (MX + KX)], float gain_values_dev[NUM_PMTS]);
```

The list of `#pragma` used to define the interfaces is:

```
#pragma HLS INTERFACE m_axi port=estim_event_dev offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=PMT_data_dev offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=mdrf_spline_coefs_dev offset=slave bundle=gmem2
#pragma HLS INTERFACE m_axi port=thresh_spline_coefs_dev offset=slave bundle=gmem3
#pragma HLS INTERFACE m_axi port=gain_values_dev offset=slave bundle=gmem4
#pragma HLS INTERFACE s_axilite port=estim_event_dev bundle=control
#pragma HLS INTERFACE s_axilite port=PMT_data_dev bundle=control
#pragma HLS INTERFACE s_axilite port=num_events bundle=control
#pragma HLS INTERFACE s_axilite port=mdrf_spline_coefs_dev bundle=control
#pragma HLS INTERFACE s_axilite port=thresh_spline_coefs_dev bundle=control
#pragma HLS INTERFACE s_axilite port=gain_values_dev bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
```

6.3 Loop Unrolling and Pipelining

Loop unrolling is a technique to exploit parallelism between loop iterations. It creates multiple copies of the loop body and adjust the loop iteration counter accordingly. In Vivado HLS a loop can be unrolled by placing `#pragma HLS unroll` at the beginning of the body of the loop. Loops with a small trip count are typically automatically unrolled by default.

Pipelining is a technique for implementing instruction-level parallelism within a single processor. Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps (source: Wikipedia). A pipelined function or loop can process new inputs every `N` clock cycles, where `N` is the initiation interval (II) of the loop or function. In Vivado HLS, a block of instructions can be pipelined by placing `#pragma HLS PIPELINE II=N` within the body of the function or loop. The integer `N` is typically 1. An important concept related to pipelining is the "depth" of the pipeline, which can be interpreted as the number of steps needed to fill up the pipeline. Once the pipeline is full, a new result is available every `N` clock cycles.

It is important to realize that "loop carry dependencies" can prevent pipelining inside a loop. As an example, consider the following piece of code for the calculation of the M non-zero spline basis functions:

```
template<class _C, int _M, int _K> void evaluate_basis(_C basis[_M], const _C & x, int ell) {
  _C saved, tmp;
  int m, j;

  if(ell >= 0) {
    basis[0] = _C(1);
    for(m = 1; m < _M; ++m) {
      saved = _C(0);
      for(j = 0; j < m; ++j) {
        tmp = basis[j] / (_C(m) / _C(_K + 1));
        basis[j] = saved + (_C(ell + j + 1) / _C(_K + 1) - x) * tmp;
        saved = (x - _C(ell + j - m + 1) / _C(_K + 1)) * tmp;
      }
      basis[m] = saved;
    }
  }
  return;
}
```

In the code above, the array used to return the result (`basis`) is also used as a temporary buffer. This introduces a read-after-write (RAW) dependency on `basis[j]` in the body of the innermost loop, and the body of the loop exposes limited pipelining potential. Better pipelining can be achieved if temporary buffers are used to store intermediate results as shown below:

```
template<class _C, int _M, int _K> void evaluate_basis(_C basis[_M], const _C & x, int ell) {
  #pragma HLS ARRAY_PARTITION variable=basis complete
  _C saved, tmp, t;
  _C buff[2][_M];
  int m, j;

  if(ell >= 0) {
    buff[0][0] = _C(1);
    for(m = 1; m < (_M - 1); ++m) {
      #pragma HLS PIPELINE II=1
      saved = _C(0);
      for(j = 0; j < m; ++j) {
        tmp = buff[(~m) & 1][j] * _C(_K + 1) / _C(m);
        t = _C(ell + j + 1) / _C(_K + 1);
        buff[m & 1][j] = saved + (t - x) * tmp;
        t = _C(ell + j - m + 1) / _C(_K + 1);
        saved = (x - t) * tmp;
      }
      buff[m & 1][m] = saved;
    }
    saved = _C(0);
    for(j = 0; j < (_M - 1); ++j) {
      #pragma HLS PIPELINE II=1
      tmp = buff[_M & 1][j] * _C(_K + 1) / _C(_M - 1);
      t = _C(ell + j + 1) / _C(_K + 1);
      basis[j] = saved + (t - x) * tmp;
      t = _C(ell + j - _M + 2) / _C(_K + 1);
      saved = (x - t) * tmp;
    }
    basis[_M - 1] = saved;
```

```
  }
  return;
}
```

This improved implementation uses a 2D array (named `buff`) of size $2 \times M$ so that its rows are alternatively used as reading and writing buffers. The bit-level C operators `&` and `~` are used to calculate the row index from the loop variable `j`. Finally, the last iteration of the loop over `m` was manually unrolled and each write to the temporary buffer was replaced with a write to the array used to return the result.

6.4 Vivado HLS Streams and Producer-Consumer Design

Vivado HLS supports first-in, first-out blocking queues through "streams". Streams can be used in a producer-consumer scenario by specifying `#pragma HLS DATAFLOW` in the body of a function. As an example, the top function declared two streams:

```
#pragma HLS STREAM variable=PMT_data_stream depth=32
#pragma HLS STREAM variable=estim_event_stream depth=32
#pragma HLS DATAFLOW
static hls::stream<PMT_data_t> PMT_data_stream("PMT_data_stream");
static hls::stream<estim_event_t> estim_event_stream("estim_event_stream");
```

These streams are used in two instances of the producer-consumer paradigm inside the top function as follows:

```
read_input(PMT_data_stream, PMT_data_dev, num_events);
compute(estim_event_stream, PMT_data_stream, num_events,
  mdrf_spline_coefs_dev, thresh_spline_coefs_dev, gain_values_dev);
write_result(estim_event_dev, estim_event_stream, num_events);
```

The `#pragma HLS DATAFLOW` design directive instructs the compiler to schedule the three function above in parallel. Function `read_input(...)` reads from global memory (pointer `PMT_data_dev`) PMT data and inserts (produces) them into the `PMT_data_stream` stream. Function `compute(...)` consumes PMT data from the `PMT_data_stream` stream, performs the contracting-grid estimation and writes (produces) estimates to the `estim_event_stream` stream. Finally, function `write_result(...)` consumes data from the `estim_event_stream` stream and writes them to global memory (pointer `estim_event_dev`). To reiterate, two instances of the producer-consumer paradigm take place:

1. between `read_input(...)` and `compute(...)`; and
2. between `compute(...)` and `write_result(...)`.

Because these function calls are scheduled in parallel in the final design, this strategy effectively overlaps memory transfers (inside `read_input(...)` and `write_result(...)`) with actual computation (inside `compute(...)`).

6.5 (Re)-Implementing $\log\Gamma(x)$

During implementation and testing of the top function, it was discovered that the Vivado HLS library function to calculate $\log\Gamma(x)$ presents some bugs. Although the library code for $\log\Gamma(x)$ is based on the C implementation of `lgamma`, some instruction were omitted, resulting in uninitialized variables and unspecified behavior. For this reason, I decided to implement my own `lgamma` function for FPGA. This function, which I called `my_lgamma`, was templated so that it can be used with any numerical type, including the fixed-point data types `ap_[u]fixed<W,I,Q,O,N>` available in Vivado HLS. Testing of `my_lgamma` was performed by comparing its outputs for single and double precision against the values of the standard C functions `lgammaf` and `lgamma`, respectively.

6.6 Generating the Design

A TCL file (`contr_grid.tcl`) has been written to:

1. compile and run pre-synthesis C++ simulation using the provided C test bench;
2. synthesize the Vivado HLS database for the active solution;
3. export and package the synthesized design in RTL.

This TCL file can be executed as `vivado_hls contr_grid.tcl`. In absence of errors, file `contr_grid.xo` is created. This process should take just a few minutes. Kernel accelerator functions can be compiled and linked with the Xilinx® OpenCL Compiler (XOCC):

```
xocc -l --platform xilinx_u250_xdma_201830_2 --optimize 2 contr_grid.xo -o contr_grid.xclbin
```

The step above, which can take several hours to complete, produces the file `contr_grid.xclbin` which can be loaded with OpenCL function calls. For an example, please refer to the code.

7 Performance Results

Tests were conducted with a data set consisting of 901398 events. The size of the contracting grid was set to $T \times T = 6 \times 6$. For each estimation, 12 iterations were performed and the contracting factor α was set to 1.75. MDRF and log-likelihood threshold samples were for a 79×79 sample grid, and the (physical) spacing between contiguous samples was $1.75\ \mathrm{mm}$. In the least square fitting with 2D splines we used $M_x = M_y = 3$ and $K_x = K_y = 10$.

Performance results are summarized in Table 1 and Figure 1.

Architecture	Details	Events/s	Speedup
CPU	Intel® Xeon® Gold 6134	2831.38	1.00
GPU	NVIDIA® Tesla® V100 SXM2	804896	284.27
FPGA	Xilinx® Alveo™ U250	30578200	10799.75

Table 1: Comparison of contracting-search algorithm on CPU, GPU and FPGA

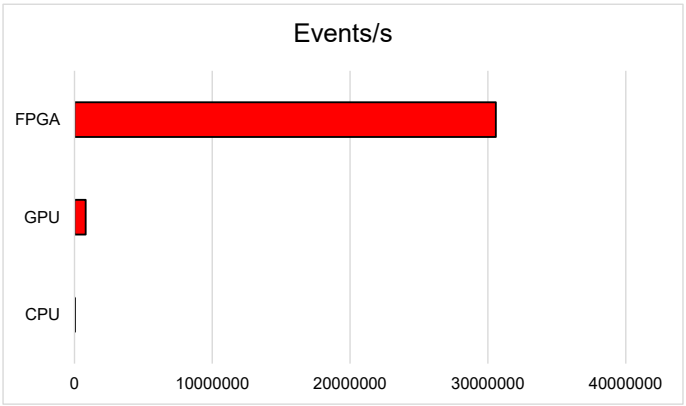

Figure 1: Comparison of contracting-search algorithm on CPU, GPU and FPGA

References

[1] ANGER, H. O. Scintillation camera. *Review of Scientific Instruments 29*, 1 (Jan. 1958), 27–33.

[2] CAUCCI, L., FURENLID, L. R., AND BARRETT, H. H. Maximum likelihood event estimation and list-mode image reconstruction on GPU hardware. In *IEEE Nuclear Science Symposium Conference Record* (Orlando, FL, Oct. 2009), pp. 4072–4076.

[3] FURENLID, L. R., HESTERMAN, J. Y., AND BARRETT, H. H. Real-time data acquisition and maximum-likelihood estimation for gamma cameras. In *14th IEEE-NPSS Real Time Conference* (Stockholm, Sweden, June 2005), pp. 498–501.

[4] FURENLID, L. R., WILSON, D. W., CHEN, Y.-C., KIM, H., PIETRASKI, P. J., CRAWFORD, M. J., AND BARRETT, H. H. FastSPECT II: A second-generation high-resolution dynamic SPECT imager. *IEEE Transactions on Nuclear Science 51*, 3 (June 2004), 631–635.

[5] PIEGL, L. A., AND TILLER, W. *The $\mathcal{NURBS}$ Book*. Springer, Berlin, Germany, 1997.