The International Journal of High
Performance Computing Applications
0(0):1–13
©The Author(s) 2014
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI:doi number
hpc.sagepub.com

Productively Accelerating PET Image Reconstruction on GPUs with Julia

Michiel Van Gendt[*]
Ghent University, Belgium

Tim Besard[†]
Julia Computing, Belgium

Stefaan Vandenberghe[‡], **Bjorn De Sutter**[§]
Ghent University, Belgium

Abstract
Research in medical imaging is hampered by a lack of programming languages that support productive, flexible programming as well as high performance. In search for higher quality imaging, researchers can ideally experiment with novel algorithms using rapid-prototyping languages such as Python. However, to speed up image reconstruction, computational resources such as those of GPUs need to be used efficiently. Doing so requires re-programming the algorithms in lower-level programming languages such as CUDA C/C++ or rephrasing them in terms of existing implementations of established algorithms in libraries. The former has a detrimental impact on research productivity and requires system-level programming expertise, the latter puts severe constraints on the flexibility to research novel algorithms. Here, we investigate the use of the Julia scientific programming language in the domain of PET image reconstruction as a means to obtain both high performance (portability) on GPUs and high programmer productivity and flexibility, all at once, without requiring expert GPU programming knowledge.

Using rapid-prototyping features of Julia, we developed basic and performance-optimized GPU implementations of baseline MLEM PET image reconstruction algorithms, as well as multiple existing algorithmic

* Email: michiel.vangendt@ugent.be
† Email: tim@juliacomputing.com
‡ Email: stefaan.vandenberghe@ugent.be
§ Email: bjorn.desutter@ugent.be (corresponding author)

extensions. Thus we mimic the effort that researchers would have to invest to evaluate the quality and performance potential of algorithms. We evaluate the obtained performance and compare it to state-of-the-art existing implementations. We also analyze and compare the required programming effort.

With the Julia implementations, performance in line with existing GPU implementations written in the low-level, unproductive programming language CUDA C is achieved, while requiring much less programming effort, even less than what is needed for much less performant CPU implementations in C++.

Switching to Julia as the programming language of choice can therefore boost the productivity of research into medical imaging and deliver excellent performance at a low cost in terms of programming effort.

Keywords
GPU, Julia, high-level programming, performance, flexibility, productivity, PET image reconstruction

1. Introduction

Medical image reconstruction techniques are compute- and data-intensive. Traditionally, image reconstruction algorithms are executed on high-end Central Processing Units (CPUs), but Cui et al. (2011), Pratx et al. (2009), Xu and Mueller (2005) and Zhou and Qi (2011) have shown that a Graphics Processing Unit (GPU) can offer two orders of magnitude speedup compared to single-threaded CPU algorithms. The downside of GPUs is that they historically could only be, and today still mostly are, programmed in specific low-level programming languages such as CUDA C/C++ from NVIDIA (2020b) and OpenCL from Khronos Group (2020) that require expert programming skills. This makes it inconvenient for medical imaging researchers to explore new reconstruction algorithms, for which both image reconstruction times and quality are important optimization criteria. The use of external (hand-tuned) libraries is no satisfactory alternative, because those contain only established algorithms and rephrasing new algorithms around those often introduces needless overhead.

Medical imaging researchers hence commonly explore new reconstruction techniques in high-level languages such as Matlab or Python, where they focus on reconstruction quality at the cost of lower performance. This allows them to be productive, as they can work in their domain of expertise and their comfort zone at the algorithmic level. Before the code can be used in production, or before its practical potential as a fast enough reconstruction technique can be assessed, the algorithms are then reprogrammed using one of the mentioned lower-level language to achieve acceptable resource utilization on GPUs. This either requires the involvement of programming and computer architecture experts, or that the medical imaging researchers acquire sufficient such expertise themselves and invest into the reprogramming and performance tuning. The sketched work flow is clearly not optimal. Bezanson et al. (2014) called the resulting drop in overall productivity the "two-language problem". This problem is aggravated by the fact that implementations in the lower-level languages offer little performance portability: whenever a new generation of GPU surfaces, the code needs to be re-tuned.

The high-level Julia programming language provides a potential solution: Besard et al. (2019) added support for GPU programming with higher-level abstractions to the Julia ecosystem, and for at least some scientific computations, excellent GPU resource utilization has been demonstrated. It is still unclear, however, if Julia is sufficiently mature to enable high-performance execution of rapid-prototyping implementations of image reconstruction algorithms on GPUs. Additionally, it is uncertain to what extent image reconstruction algorithms can make use of higher-level abstractions in the language. The use of these abstractions is important: without it, there is only a small improvement in the developer's productivity compared to using low-level languages.

To assess the potential of Julia in this domain, we experimented with several Julia GPU implementations of Positron Emission Tomography (PET) image reconstruction algorithms. We programmed a basic and a performance-optimized implementation of a baseline Maximum Likelihood Expectation Maximization (MLEM)

reconstruction algorithm, as well as several algorithmic extensions from the state-of-the-art, thus covering the types of programming efforts that typically occur in medical imaging research and that can be performed by people with and without GPU programming expertise. We evaluated aspects of programmer productivity, such as the ease with which the extensions can be implemented and the required GPU expertise. We also evaluated performance portability over variations of algorithms and variations of hardware to assess the extent to which researchers in this domain can truly benefit from Julia to avoid the two-language problem.

This paper reports our experience in implementing the versions of the MLEM reconstruction algorithms, and our evaluation in terms of programmer productivity, achieved performance, and performance portability.

2. Background and Related Work

2.1. Baseline PET MLEM

As positrons are emited following a Poisson distribution, PET image reconstruction is often implemented by means of the statistical iterative reconstruction algorithm of MLEM as proposed by Lange and Carson (1984). Alternative approaches exist, such as algebraic reconstruction techniques. We refer the interested reader to Bailey et al. (2005b) for an overview and for more mathematical background on the MLEM method. In this work, we use the algorithm by Barrett et al. (1997) and Parra and Barrett (1998) for list-mode PET data, the mode proposed by Watabe et al. (2002). The MLEM algorithm is initialised with an arbitrary image estimate. Every iteration then updates that estimate based on the measured data until a point of convergence is reached.

Assume the tracer distribution in a brain is a continuous function $f(r)$ in the scanned 3D space. The scanner outputs a list of J coincidence events $A = \{A_1, ..., A_J\}$. The jth coincidence in the list is described by $A_j = (R_{1,j}, R_{2,j}, \Delta t_j)$, $R_{1,j}$ and $R_{2,j}$ are the 3D locations of points of interaction with the scanner, which form a so-called Line Of Response (LOR) and Δt_j is the Time-of-Flight (TOF) difference.

For the reconstruction, the 3D space is divided into N voxels r_n for $n = 1, ..., N$. The MLEM algorithm computes $f = \{f_1, ..., f_N\}$, a discrete approximation to $f(r)$ in the voxel space. The likelihood of the realisation f given the measurements A is defined as:

$$\mathcal{L}(f; A) = \prod_{j=1}^{J} p(A_j \,|\, f) \tag{1}$$

The conditional probability $p(A_j \,|\, f)$ is the probability of measuring coincidence event A_j given a discrete tracer distribution f. Assuming A_j and $A_{j'}$ are statistically independent as proposed by Parra and Barrett (1998), the conditional probability from Equation (1) equals:

$$p(A_j \,|\, f) = \sum_{n=1}^{N} \underbrace{p(A_j \,|\, r_n)}_{(1)} \underbrace{P(r_n \,|\, f)}_{(2)} \tag{2}$$

in which (1) is the probability of A_j given that a pair of photons is emitted from voxel r_n, and (2) is the probability of the emission of a photon pair in voxel r_n given a discrete tracer distribution f. With S_n being the sensitivity, i.e., the probability that two photons emitted from voxel r_n are collected by the detectors, the conditional probability is computed as:

$$P(r_n \,|\, f) = \frac{S_n f_n}{\sum_{n'=1}^{N} S_{n'} f_{n'}} \tag{3}$$

Berg et al. (2018) discussed that the sensitivity is a scanner-specific matrix and can be computed exactly from the system geometry or it can be measured.

Each iteration k of the MLEM algorithm first computes the expectation of the log-likelihood function from Equation (1) given the current image estimate $f^{(k)}$ and the measured data A. Subsequently, the algorithm maximises the expectation of the log-likelihood. The full derivation of the update equation by Shepp and Vardi (1982) can be found in the literature. It is as follows:

$$f_n^{(k+1)} = f_n^{(k)} \cdot c_n^{(k)} = f_n^{(k)} \cdot \left(\frac{1}{T} \sum_{j=1}^{J} \frac{p(A_j \,|\, r_n)}{\sum_{n'=1}^{N} p(A_j \,|\, r_{n'})\, S_{n'}\, f_{n'}^{(k)}} \right) \tag{4}$$

The update Equation (4) can be split in four steps.

1. **Forward project** measurements into the data space with current image estimate. This is the computation of the sum in the denominator in eq. 4.
2. **Compare** estimations with measurements and compute an "error" in the data space. This is the fraction in eq. 4.
3. **Back project** the error from the data space into the image space. This is the summation over j in eq. 4.
4. **Update** the current image estimate with the normalised error in the image space. This is the remaining computation of eq. 4, i.e., the element-wise multiplication of the old image voxel values $f_n^{(k)}$ with the correction values $c_n^{(k)}$.

2.2. *Ray tracing*

As for Equation (4), it has been proven that $p(A_j \mid r_n) \propto l(A_j; r_n)$, i.e., the probability of A_j given the fact that a pair of photons is emitted from voxel r_n is proportional to the intersection length of the LOR of the jth coincidence event inside voxel r_n. One can further derive that $p(A_j \,|\, r_n) = c \cdot l(A_j; r_n)$ where c is a constant, unknown factor. When substituting this into the MLEM update equation 4, the factor c is cancelled out and the update equation becomes:

$$f_n^{(k+1)} = f_n^{(k)} \cdot c_n^{(k)} = f_n^{(k)} \cdot \left(\frac{1}{T} \sum_{j=1}^{J} \frac{l(A_j; r_n)}{\sum_{n'=1}^{N} l(A_j; r_{n'}) S_{n'} f_{n'}^{(k)}} \right) \tag{5}$$

For most values of j and n, $l(A_j; r_n) = 0$ because the LOR of event j does not pass through voxel n. The denominator in eq. 5 is hence in practice not computed by iterating over all voxels of the image, but by visiting only the voxels through which LOR j passes. The MLEM algorithm that iteratively updates the image estimate then becomes as shown in the pseudo-code in Algorithm 1. Lines 6–7 correspond to the forward projection of each event, lines 8–9 to the comparison and back projection, and line 10 performs the actual update of all elements of the estimated image.

Calculating the intersection length l of a LOR (a.k.a. ray) with every single voxel is known as ray tracing. It is done on-the-fly while visiting the voxels on a LOR, because too much storage would be required to store all non-zero pre-computed intersection lengths. The algorithm by Siddon (1985) is a frequently used ray tracing technique in list-mode MLEM image reconstruction. It iterates sequentially over the voxels on a LOR, and hence implicitly skips all voxels that contribute zero to eq. 5. Several improvements to the original algorithm are proposed in more recent works by Christiaens et al. (1999), De Sutter et al. (1998), Schretter (2007), Zhao and Reader (2002). However, because Siddon's algorithm and the improvements thereof are hard to parallelize

Algorithm 1 Baseline MLEM algorithm

```
1:  Initialize(image)
2:  for iterations 1 to K do
3:      zero-initialise corr (same dimensions as image)
4:      for all events j from 1 to J do
5:          forward_projection = 0
6:          for all voxels n on LOR of j do
7:              forward_projection += l(j,n) · image(n) · sensitivity(n)
8:          for all voxels n on LOR of j do
9:              corr[n] += l(j,n) / forward_projection
10:     image = image ⊙ corr        /* element-wise product in eq. 5 */
```

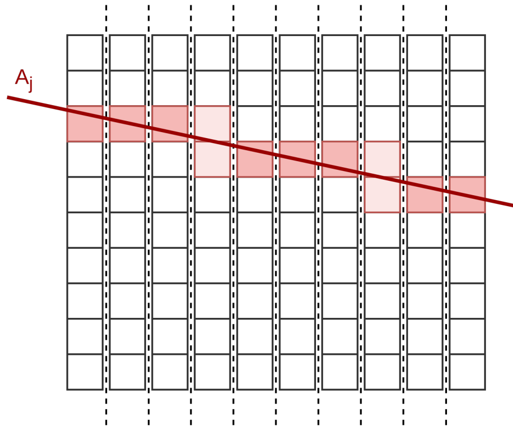


Figure 1. The ray tracing problem divided into slices along its main direction

over multiple threads as required to exploit the resources of GPUs, we build on another class of ray tracing techniques known as *slice-based ray tracing*. This class was proposed by Gao (2012) and Pratx et al. (2009), and has been shown to be much better suited for GPUs by Cui et al. (2011). In particular, it enables massive parallelization of the nested for loops of lines 4–9 of Algorithm 1 by handling multiple events in parallel and by handling multiple segments (called slices) of each event's ray in parallel.

Consider the 2D pixel space in Figure 1, where the example ray's *main direction* is along the x-axis. The pixel space is then divided into columns. In each column, the ray intersects with at most two pixels, called a slice. It is relatively straightforward to compute the intersection indices and lengths, as shown by Gao (2012), meaning that the slices and voxels that can contribute non-zero values in eq. 5 are identified on the fly, and only those are visited and accessed during the computation, as in Siddon's algorithm, thus implicitly exploiting the sparsity in $l(A_j; r_n)$. Similarly, if the main direction of the ray is along the y-axis, the pixel space can be divided into rows. For each row, the ray intersects with at most two pixels and the intersection indices and lengths can be computed. With analogue reasoning, the same principles can be applied to a 3D voxel space. The main direction of the ray is determined and the voxel space is divided into either x-planes, y-planes or z-planes. For every plane, the ray intersects with at most three different voxels as shown by Gao (2012).

The computational complexity per ray, i.e., the number of slices to be visited, is O(N), where N denotes the number of voxels in the main direction. Additionally, the computational complexity of each sub-problem, i.e., each slice, is O(1). These sub-problems are independent, meaning it is possible to compute the intersection lengths for each slice in parallel, thus exploiting the massively parallel computing resources of a GPU.

2.3. MLEM Extensions

We considered four improvements, extensions, and adaptations to the above baseline MLEM algorithm.

Ordered Subset Expectation Maximization The Ordered Subset Expectation Maximization (OSEM) algorithm, originally proposed by Hudson and Larkin (1994), makes a simple adjustment to the MLEM algorithm. The coincidence events are divided into S disjunct subsets $\{J_1, ..., J_S\}$ and the update equation is applied to each subset in a certain order. This comes down to replacing the J in eq. 4 by $J_{k \bmod S}$. Hudson and Larkin (1994) demonstrated that this improves the convergence speed by a factor $\simeq S$.

Correction for image degrading effects Due to the interaction of photons with matter, there is a certain probability that emitted photons do not reach the detectors. To correct for this effect called attenuation, every LOR must be weighted proportionally during the reconstruction process: LORs that were highly attenuated should have a higher contribution to the reconstructed image. The attenuation factor of a LOR can be computed as:

$$\alpha_j = exp\left(-\sum_{n=1}^{N} l(A_j; r_n)\, \mu_{n'}\right) \tag{6}$$

where $\mu_{n'}$ is the attenuation coefficient of the attenuation map in voxel r_n and $l(A_j; r_n)$ is the intersection length of the LOR inside voxel r_n according to Bailey et al. (2005a). The attenuation factor α_j can be incorporated in Equation (4), resulting in the attenuation-weighted MLEM algorithm by Hebert and Leahy (1990):

$$f_n^{(k+1)} = f_n^{(k)} \left(\frac{1}{T} \sum_{j=1}^{J} \frac{p(A_j \,|\, r_n)}{\boldsymbol{\alpha_j} \sum_{n'=1}^{N} p(A_j \,|\, r_{n'})\, S_{n'}\, f_{n'}^{(k)}} \right) \tag{7}$$

Time-of-Flight image reconstruction In the versions described thus far, the MLEM algorithm does not make use of the TOF information of the LOR. The TOF difference Δt_j is related to the distance d_j between the point of annihilation and the centre of the LOR according to:

$$d_j = c\frac{\Delta t_j}{2} \tag{8}$$

TOF measurements, however, are not exact and limitations in PET detectors and electronics limit the timing resolution. The uncertainty can be modelled as a Gaussian distribution, whose variance is a scanner-specific value. The probabilities $p(A_j \,|\, r_n)$ in Equation (4) must be re-weighted according to the Gaussian distribution that gives a higher weight to voxels that lie closer to the estimated point of annihilation. Mullani et al. (1980) and Efthimiou et al. (2019) present the full incorporation of the TOF principle in the MLEM update equation.

Filtering techniques The image estimate resulting from the MLEM can contain much statistical noise, especially at a large iteration number. Several filtering techniques that suppress the noise, increase the signal-to-noise ratio, and try to preserve the spatial resolution and contrast are used in practice. A good fraction of the state-of-the-art commercial methods use regularization as described by Ahn et al. (2015), but we consider two simple filtering techniques, which can be applied post-reconstruction or inter-iteration. A Gaussian filter is capable of lowering the noise in the reconstructed images as shown by Kim et al. (2014) and Arabi and Zaidi (2018). It does, however, add blurriness. As a result, the edges between different tissue are softened. Compared to a gaussian filter, a median filter results in less edge softening, while still lowering the amount of noise during image reconstruction as shown by Gui and He (2012). We hence used the latter in our research.

2.4. GPU Acceleration

Cui et al. (2011), Pratx et al. (2009), Xu and Mueller (2005) and Zhou and Qi (2011) have shown that the use of GPUs in the domain of medical image reconstruction can lead to up to two orders of acceleration over single-threaded CPU implementations. They obtained parallelization on GPUs by splitting the problem into smaller subtasks (i.e., by splitting the ray and/or image space into smaller parts) and calculating multiple forward and back projection operations in parallel. These works, however, all use a low-level programming language such as CUDA C/C++ or OpenCL. This is problematic for several reasons. CUDA C/C++ and OpenCL code is quite verbose and requires a significant amount of boilerplate code around the code that specifies the actual computations to be performed. This requires additional expertise from the researchers outside of the domain of image reconstruction, and lowers their productivity by distracting them from their research at the algorithmic level. Unlike in rapid-prototyping languages like Python, CUDA C/C++ and OpenCL code is typed statically, which hinders code re-use. Furthermore, that code does not offer performance portability: code tuned for performance on one specific GPU device needs to be re-tuned, i.e., (partially) rewritten, for achieving good performance on other devices.

While the use of third-party libraries can free application programmers from some of that burden, such as when scientific computing applications re-use cuBLAS libraries from NVIDIA (2020a), such libraries are not useful for researchers studying alternative algorithms. By construction, third-party libraries only contain optimized implementations of established, widely-used algorithm components. Rephrasing novel algorithms in terms of those components often introduces additional overhead, typically because extra kernel launches are then needed, with a detrimental impact on memory traffic and data locality.

2.5. The Julia programming language

This work is the first in studying the use of a high-level programming language for researching MLEM algorithms on the GPU, which allow programmers to become much more productive. However, whereas using high-level languages such as Matlab, R, or Python typically do not offer acceptable utilization of a GPUs resources once the researcher explores algorithms beyond the established ones in libraries, we will show that with the Julia language, both productive programming and good GPU utilization can be achieved together.

The Julia programming language is designed for scientific and technical computing. It aims to be a rapid prototyping language according to Bezanson et al. (2012). Hence Julia is a high-level and dynamically-typed language. Code written in Julia is compiled to machine code via a Just-In-Time (JIT) compiler based on the LLVM compiler initially developed by Lattner and Adve (2004). Despite Julia being dynamically-typed, extensive type inference is done prior to the JIT compilation. This allows the JIT compiler to generate highly optimized code that omits the overhead of dynamic typing. Consequently, Julia's performance on standard CPUs is comparable to that of statically compiled language such as C and Fortran.

Besard et al. (2019) have previously extended the Julia ecosystem with support for NVIDIA GPUs. All CUDA programming functionality (i.e., all functionality need to program GPUs from NVIDIA that are normally programmed in CUDA C) is bundled into one package: CUDA.jl[1]. Besard et al. (2019) have also demonstrated that the performance of Julia on GPUs is similar to the performance of CUDA C/C++ for at least some benchmarks. Whether Julia's GPU support, being relatively young and far from complete, already suffices for use in specific domains such as medical image reconstruction, was an open question before this research. Ray tracing, for

[1] `https://github.com/JuliaGPU/CUDA.jl`

example, is not a type of computation that easily maps onto already supported operations such as algebraic computations on vectors and matrices. In this paper, we try to answer this question.

In parallel with our research, but unknown to us until after the initial submission of this work, Knopp and Grosser developed and presented MRIReco.jl, an MRI reconstruction framework Knopp and Grosser (2021). Whereas we focus our work described below focuses on combining high programmer productivity with efficient use of the massively parallel GPU resources, Knopp and Grosser focused on CPUs, only briefly mentioning preliminary GPU support. Their MRI image reconstructions operators build on FFT and NFFT, while we focus on ray tracing, a very different computation. Their and our work is hence complementary.

2.6. QETIR

We use the existing image reconstruction software package QETIR as a basis for comparison and to validate the correctness of our implementations in Julia. QETIR has been used in research before, notably by Thoen et al. (2013) and by Kolstein and Chmeissani (2016). It contains implementations of the baseline MLEM algorithm, using an improved version of Siddon's algorithm, plus several of the extensions we experiment with. Being written in C++, QETIR only runs on modern x86-64 CPUs. It does not support GPU acceleration.

3. Basic Implementation

First, we implemented the baseline MLEM algorithm in Julia in combination with the slice-based ray tracing technique. This basic implementation excludes sensitivity correction, which is instead studied as a possible extension, and stays close to the mathematical descriptions, featuring no GPU-specific optimizations. This implementation can be seen as an implementation done by someone who knows very little about GPUs apart from the fact that they need to be programmed with kernels. The source code, as well as that of all other MLEM versions evaluated in this paper, can be found at `https://github.com/michielvangendt/PET-Julia`.

To evaluate the performance of this basic implementation, we ran it on a simulated scan of a phantom image containing three radioactive line sources. The simulation was performed with GATE, the simulation toolkit developed by Jan et al. (2004). As the number of iterations through the four steps of the MLEM reconstruction algorithm is identical for our Julia implementations and for the QETIR implementation independent of the image content, one such image at different dimensions suffices for analyzing and comparing the performance of the implementations.

The performance of the basic GPU implementation was evaluated and compared to a basic version of the CPU-only single-threaded QETIR C++ code[2]. The latter was benchmarked on a machine with two Intel Xeon E5-2637 v2 @3.50GHz CPUs. The Julia GPU code is benchmarked on an NVIDIA Tesla V100 16GB and on an NVIDIA RTX 2080 Ti. The specs of the GPUs can be found in Table 1.

Figure 2 shows a somewhat simplified code of the main GPU computation kernel of the basic implementation of the baseline MLEM algorithm. This kernel is executed in parallel for all slices on all events, thus implementing the computations of lines 4–9 of Algorithm 1 (without the sensitivity correction). No complex constructs or GPU specific code such as those on lines 11 and 12 have been omitted or simplified, which by the way are the only GPU-specific code lines. Only the computations on lines 16–36 have been reduced from three planes (x,y,z) into one case to avoid redundant complication of the code.

[2] This basic QETIR version was obtained by omitting extensions from the MLEM implementation in QETIR.

	Tesla V100	RTX 2080 Ti
GPU architecture	NVIDIA Volta	NVIDIA Turing
CUDA cores	5,120	4,352
Double-precision performance	7 TFLOPS	0.4 TFLOPS
Single-precision performance	14 TFLOPS	14 TFLOPS
GPU memory	16GB HBM2	11GB GDDR6
Memory bandwidth	900GB/sec	616GB/sec

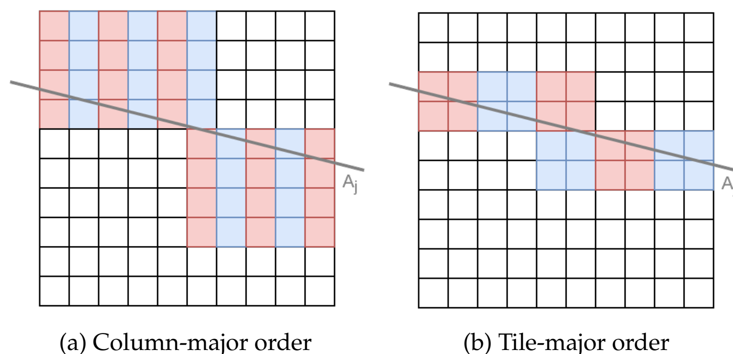
Table 1. Specification of the NVIDIA GPUs

```
1  using CUDA: threadIdx, blockIdx, sync_threads, reduce_block, @cuStaticSharedMem, @atomic
2
3  """
4      @cuda threads=... blocks=... gpu_kernel!(events, image, corr, tmp_forward_projections, dimx, dimy, dimz)
5
6  Perform forward projection, compare, and back projection steps.
7  Output is written to `corr`, which is expected to be zero-initialised.
8  `tmp_forward_projections` is used for temporary storage and is also expected to be zero-initialized.
9  """
10 function gpu_kernel!(events, image, corr, tmp_forward_projections, dimx, dimy, dimz)
11   s_i = threadIdx().x # The index of the current slice
12   e_i = blockIdx().x # The index of the current event
13
14   event = events[e_i]
15
16   # Ray tracing
17   Ym, Yp, Zm, Zp, l1, l2, l3, l4 = ray_tracing(event, s_i, dimx, dimy, dimz)
18
19   value = 0.0
20   # Forward project
21   value += image[s_i, Ym+1, Zm+1] * l1
22   value += image[s_i, Ym+1, Zp+1] * l2
23   value += image[s_i, Yp+1, Zm+1] * l3
24   value += image[s_i, Yp+1, Zp+1] * l4
25
26   # Sum forward projection values for all slices
27   @atomic tmp_forward_projections[e_i] += Float32(value)
28   # Wait for all slices to be processed
29   sync_threads()
30   forward_projection = tmp_forward_projections[e_i]
31
32   # Compare and back project
33   corr[s_i, Ym+1, Zm+1] += l1 / forward_projection
34   corr[s_i, Ym+1, Zp+1] += l2 / forward_projection
35   corr[s_i, Yp+1, Zm+1] += l3 / forward_projection
36   corr[s_i, Yp+1, Zp+1] += l4 / forward_projection
37
38   return nothing
39 end
```

Figure 2. (Simplified) source code of the kernel function that performs a forward projection, compare, and back projection.

4. Performance Optimizations

The basic implementation discussed above suffers from a memory bottleneck that results from the layout of the data in memory not matching the data access pattern of the algorithm. With the slice-based ray tracing approach, each thread computes the intersection indices and lengths of a single row or column, depending on the LOR's main direction. In the simplified 2D example in Figure 3a, ten threads are computing the intersection indices and lengths of the ten columns in parallel. During this computation, and according to the MLEM forward projection formula, each thread fetches the current image values of the intersecting pixels. Assuming the array is stored in column-major order and four array values can be fetched by a single memory operation, a total of



(a) Column-major order (b) Tile-major order

Figure 3. Visualization of the requested memory for the two array storage methods

```
1 shared = @cuStaticSharedMem(Event, 1)
2 if s_i == 1
3     shared[1] = events[e_i]
4 end
5 sync_threads()
6 event = shared[1]
```

Figure 4. Using shared memory to fetch the event data

11 memory operations are needed that each load a group of four coloured pixels, consuming the space for 44 values in the cache. Had row-major order been used, a similar number of memory accesses would have been needed, and a similar amount of cache space would have been devoted

Instead of storing the array values in a row- or column-major order, we switched to a novel storage method. As shown in Figure 3b, the array is now stored in tiles, hence the name *tile-major order*. Thanks to the memory coalescing principle in CUDA, the same kernel now only requires six memory operations in this example, and only space for 24 values need occupy space in the cache.

Other options to reduce the pressure on memory and to increase the resource utilization include the use of the shared memories, texture memories, and constant memories of GPUs, as previously demonstrated by Cui et al. (2011), Zhou and Qi (2011) and Xu and Mueller (2005). We only explored optimizations using shared memory, the small and fast memories (one per Streaming Multiprocessor (SM)) that can be used to store frequently used data in a block of threads.

In line 14 of Figure 2, all threads in a block fetch the same coincidence event from the global memory. Placing the event in shared memory by replacing line 14 with the code in Figure 4 results in fewer global memory accesses. Notice how the optimized code Figure 4 only consists of high-level operations such as allocating an array, fetching data, and synchronization. Importantly, this template for using shared memory does not require expert low-level GPU knowledge at all.

Lines 27-30 of Figure 2 constitute the straightforward GPU-kernel-style expression of the reduction operation corresponding to the summation of the forward projection. The accumulated values (`tmp_forward_projections`) are stored in global memory throughout the computation, the slowest GPU memory. The existing support for CUDA devices in Julia provides a `reduce_block` function that can be used to perform the same computation, while storing the intermediate results in the much faster shared GPU memory. The replacement code of lines 27-30 is listed in Figure 5. Notice that this code uses the same template as in Figure 4, as it also involves shared memory. Again, no low-level GPU expertise is needed.

In addition to those memory optimizations, the occupancy of the kernel can be optimised, i.e., its utilization of the many cores on the GPU device. The occupancy is the ratio of active warps per SM to the maximum

```
1  # Forward project
2  forward_projection = reduce_block(+, convert(Float32, value), 0.0f0)
3
4  # Share total length across all threads in the block
5  shared_forward_projection = @cuStaticSharedMem(Float32, 1)
6  if thread_i == 1
7      shared_forward_projection[1] = forward_projection
8  end
9  sync_threads()
10 forward_projection = shared_forward_projection[1]
```

Figure 5. Using block reduction to compute the forward projection value

number of possible active warps per SM. Volkov (2010) showed that increasing the occupancy can have a positive effect on the kernel execution time by hiding the latencies of slow accesses to global memory. We first adapted the basic implementation to support a number of threads that differs from the numbers of voxels in the x-direction. Then we manually evaluated multiple so-called launch configurations that specify how many concurrent threads should be created and how they should scheduled onto the GPUs computing resources, and we selected the best one. This optimization does require some expert GPU knowledge , but encoding the result in the code is relatively simple.

Finally, we relied on existing Julia macros `@inline` to ensure that code is inlined maximally, and `@inbounds` to let the compiler omit bounds checking on the accessed arrays.

5. Algorithmic Extensions

To assess the true potential of Julia for overcoming the two-language problem, we assessed to what extent the optimized Julia MLEM and ray tracing implementations for GPUs can serve as flexible research vehicles. We do so by evaluating how easy it is to introduce extensions into the optimized implementation without needing to re-tune or re-optimize the code.

For this research, we chose the existing extensions discussed in Section 2.3 By re-implementing these existing extensions, we mimicked the programming effort that the original inventors of those extensions would have had to invest if they would already have had Julia at their disposal. We also evaluated whether those changes impact the good GPU performance that was obtained with the optimizations discussed in the previous section. Obviously, if relatively minor algorithmic updates would require major changes in the deployed performance optimizations, medical image researchers would still not be able to focus on their domain knowledge, instead having to invest in code tuning again and again.

Our experiments with the algorithmic extensions were performed on a simulated, life-like phantom from previous work by Thyssen et al. (2018) involving a 133x86x580 image reconstructed with 4 MLEM iterations. We do not report on the quality of the reconstructed images as our goal is not to improve reconstruction algorithms beyond the existing state of the art. However, to validate the correctness of our implementations, we did verify that the quality of the images reconstructed with QETIR and with our Julia implementations were similar. They were not completely identical because the GPU and CPU versions feature different floating-point precision, accuracy, and rounding modes, as well as different orders for iterating over all coincidence events. For all versions of the baseline and extended algorithms reported in this paper, we observed that the root mean square error was in fact smaller (by 1-2%) for the images reconstructed with the Julia implementations than those reconstructed with QETIR. We hence consider our Julia GPU implementations correct and not suffering from a degradation in reconstruction quality.

Sensitivity Correction First, we added sensitivity correction, which we had originally excluded from our basic implementation. This correction can be implemented by adding only two lines of code to the MLEM algorithm. One line `c_sensmap = CuArray(sensmap)` initializes the sensitivity map on the GPU. Secondly, the MLEM update step is extended with the line `image = image .* correction ./ c_sensmap`. The multiplication and division operators with the dot syntax denote element-wise operations. Despite being two different operations, the element-wise multiplication and the element-wise division are merged into a single broadcast kernel by the CUDA.jl package, i.e., the updates to all voxels are parallelized automatically over all available computational resources on the GPU. By contrast, in CUDA C/C++ the programmer would have to write a new kernel that performs these array operations manifestly.

Ordered Subset Expectation Maximization Replacing MLEM by OSEM requires only trivial changes to the code.

Correction for image degrading effects To add support for attenuation correction with minimal code changes, we relied on Julia's dynamic typing and multiple-dispatch. First, we added a new custom type `Att_event` on top of the already used type `Event`. The new type stores the event and its attenuation correction factor. Then in between the forward and back projections of the kernel, we inserted an inlinable method call to a method with two instantiations: one that multiplies with the computed result with the embedded attenuation factor when invoked on arguments of type `Att_Event`, and one that does nothing when invoked on arguments of the original type `Event`. Of course we also added a kernel to pre-compute the attenuation coëfficients. This uses only similar constructs as the MLEM kernel itself. All other code for the MLEM algorithm and the raytracing embedded it in remains unchanged. In CUDA C, this would not at all be the case, as that code is typed statically. Combining CUDA with C++ templating could reduce the burden to some extent, but it is precisely the cumbersomeness of that programming style that makes dynamically typed languages such as Julia shine for rapid-prototyping. In the domain of image reconstruction, our support for attenuation correction is yet another demonstration of this.

Time-of-Flight image reconstruction Similar to the previous extension, we defined a new type of event that includes the TOF information and updated the kernel function to invoke a function performing the necessary re-weighting. So in terms of programming ease, this was as easy as the attenuation correction.

TOF correction requires an array that contains the weights of the Gaussian distribution. These weights do not change throughout the execution of the algorithm and, depending on the TOF parameters, only contain $\sim$100 elements or so. This type of data is hence well-suited to be stored in the so-called constant memory space of the GPU, which is a low-latency memory, and could result in a speedup of the TOF algorithm. The Julia GPU stack, however, does not yet support the allocation of constant memory.

Filtering techniques Median filtering, the technique we have chosen for reasons explain in Section 2.3 is such commonly used that support and abstractions for it exists in packages in many high-level programming languages. In Julia, the ImageFiltering.jl package offers linear and non-linear filtering operations. Unfortunately, these operations are not (yet) implemented for the CuArray types used in Julia GPU stack.

One alternative would have been to reprogram such functionality ourselves on top of —and actually in— the Julia GPU stack. That would have required much more system-level Julia and GPU expertise than we can and should assume the average medical image processing researcher to have. As an alternative, we decided therefore that the best approach was to re-use (and slightly adapt) an existing median filtering algorithm in

CUDA C/C++[3]. The `ccall` foreign function interface of Julia makes this very easy, as it allows passing the CUArray GPU-pointer to a C code wrapper that invokes the CUDA kernel on that pointer. It is hence not necessary to copy the data back and forth between GPU and CPU memory in between our own Julia kernels and the re-used C CUDA kernels.

All extensions combined Finally, we combined all extensions. To do so, we created a new event structure that contains both the attenuation and TOF information. Beyond that change, it sufficed to change the event input type of the existing function definitions to the new type.

6. Evaluation

6.1. Performance and Performance Portability

For the baseline MLEM algorithm, Table 2 shows the execution times of the basic CPU QETIR version and of the basic and optimized Julia implementations on the two used GPUs. For the GPU versions of the baseline algorithm, we noticed that switching between 32-bit and 64-bit floats did not alter the performance in a practically significant manner. Here we report the times measured with the 64-bit floating point precision.

Depending on the GPU and the dimensions, the basic Julia implementation is between 19 and 43 times faster. Larger speedups are obtained for larger dimensions (because the overhead of invoking the GPU becomes relatively smaller), and for the more performant Tesla V100, as expected. These speedups are below the two orders of magnitude speedup that Cui et al. (2011) and Zhou and Qi (2011) have reported Cui et al. (2011), Zhou and Qi (2011), but then again, they are obtained mostly for free as we will discuss in the next section.

Part of the reason for achieving lower performance gains than reported in other literature is that the resource utilization of our main MLEM kernel is sub-optimal in this basic implementation. For example, the Tesla V100's SM utilization, i.e., the utilization of its computational resources, is only 3%.[4] The main culprit is the bad utilization of the memory hierarchy. Bad data access locality in the basic implementation resulted in extremely low L1 and L2 cache hit rates and in a relatively high amount of off-chip memory transfer (1531 GB). Off-chip memory operations take several hundreds of cycles to complete. As a result, it takes an average of 258 cycles to complete a single instruction on the GPU. In GPU-specific terms, the consequence is that there are only 0.04 eligible warps[5] per scheduler on average, which leaves the hardware resources heavily underutilised. For the RTX 2080 Ti, similar observations were made. We can conclude that although a basic, straightforward implementation of the algorithm in Julia suffices to gain a huge speedup on a GPU over equivalent code running on a CPU, that basic implementation is too basic to utilize the resources of the GPU satisfactorily.

With the optimized Julia implementation, the resource utilization improves drastically. For example, the SM and memory utilization on the Tesla V100 increased from 3% and 34% to 23% and 58% respectively, resulting in an additional speedup by a factor 2.85 over the basic GPU implementation for the largest dimensions. On the RTX 2080 Ti, an additional speedup by a factor 2.16 was reached, because of similar increases in utilization. The achieved memory utilization of the optimised algorithm is not perfect at around 60%, but that level is considered acceptable in practice according to Bavoil (2019).

[3] `https://github.com/detel/Median-Filtering-GPU/`

[4] We used NVIDIA Nsight Compute and NVIDIA Nsight Systems from the NVIDIA GPU programmer's standard toolbox to measure and study all GPU performance characteristics discussed in this paper.

[5] An eligible warp can be seen as the threads that are ready to be scheduled whenever another thread gets halted because it is waiting for data to arrive from memory. For a more precise definition, we refer to the CUDA programming model description by Sanders and Kandrot (2010).

	32x32x32	128x128x128	512x512x512
QETIR C++ on CPU (32-bit floats)	21.52s	109.68s	2455.42s
Basic Julia on Tesla V100 GPU (64-bit floats)	0.72s	3.05s	57.17s
Basic Julia on RTX 2080 Ti GPU (64-bit floats)	1.13s	4.88s	71.15s
Optimised Julia on Tesla V100 GPU (64-bit floats)	1.45s	4.48s	20.03s
Optimised Julia on RTX 2080 Ti GPU (64-bit floats)	1.74s	7.14s	33.02s

Table 2. End-to-end reconstruction time for different dimensions and 20 iterations of the baseline MLEM algorithm.

For smaller dimensions, the optimized implementation is, perhaps surprisingly at first sight, slower than the basic implementation. This slowdown results from the extra transformation code that is needed for the tile-major array indexing. This indexing method has no added value for such small dimensions because the arrays fit entirely in the cache and accessing them is fast anyway, hence the transformation only adds overhead to the complexity of the computations.

Most of the achieved speedup over the basic implementation is due to the novel tile-based access to the image arrays. On the Tesla V100, e.g., the SM utilization improved from 3% to 14%. This increase in performance is thanks to the fact that the L1 and L2 hit rates increased from 18% and 64% to 44% and 84% respectively. As a result, the threads spend much less time waiting for memory transactions to complete and can better utilise the compute resources of the GPU.

The extremely simple edit to inline functions resulted in 10% faster execution on the Tesla V100. Placing events in shared memory resulted in 47% fewer global memory accesses, but unfortunately this yielded less than 1% reduction in execution time. The use of the `reduce_block` function and shared memory for intermediate results decreased the execution time with another 15% and the simple addition of the `@inbounds` macro to line 1 in Figure 2 reduced the run time with another 22%. Finally, optimizing the occupancy of the kernel by selecting a good launch configuration results in a speedup with another 9% on the Tesla V100. For the RTX 2080 Ti, similar speedups were obtained with the different optimizations.

With the performance optimization, execution on our Tesla V100 is 122 times faster than on our CPU, and on the RTX 2080 Ti, it is 74 times faster. As such, we achieve speedups with Julia in the same ball park as the two order of magnitude speedups already reported for lower-level programming languages by Cui et al. (2011) and Zhou and Qi (2011). As we will discuss in the next section, we achieve that speedup with considerably less programming effort, however, which was our main objective.

Furthermore, for both GPUs we used exactly the same code, without manual device-specific re-tuning. The performance results on the two GPUs indicate that Julia offers a decent amount of performance portability.

To study performance portability over algorithmic variations, Table 3 lists the reconstruction times for the life-like phantom and a 133x86x580 image reconstructed in 4 MLEM iterations for the different extensions for which we implemented support building on the optimized implementation of the baseline MLEM. The following observations can be made on these results.

First, small additions like sensitivity correction result in similar overheads for the CPU versions and the GPU versions. Secondly, switching to MLEM from OSEM results in a (slightly) larger speedup on the CPU than on the GPUs. This is not due to Julia or our implementations, however. Instead it is a fundamental scalability issue of GPUs. Being accelerators with separate memories, GPUs can only shine when the overhead of transferring data between CPU memory and GPU memory and of launching kernels on the GPU is compensated by the parallelization of the computations. When fewer computations need to be performed, but the same amounts of data need to be transferred, as is the case when moving from MLEM to OSEM, the memory transfer and kernel launch overhead becomes more dominant in the overall execution times. Hence the speedup that GPU versions

can achieve over CPU versions inherently becomes smaller. This effect plays out exactly the same in CUDA C/C++ implementations.

Thirdly, the addition of attenuation correction has a larger negative impact on the GPU execution times than on the CPU times. This larger increase in execution time is caused by the overhead of an extra kernel launch to compute the attenuation correction factors and by the extra overhead to copy the attenuation map to the GPU device memory. These forms of overhead are fundamental in GPU computing, they are not a side-effect of using Julia instead of lower-level languages like CUDA C/C++. For TOF correction, similar overheads have to be paid on the GPU.[6] It is because of these extra computations and data transfers that using 32-bit floating point precision on the GPU becomes beneficial over using 64-bit precision. As stated before, the quality of the 32-bit reconstructed images was still as good (or even slightly better) on the GPU as with QETIR on the CPU.

Fourthly, for the last considered extension of median filtering, the extended QETIR algorithm suffers a 38% slowdown, while the GPU slowdowns are only 7% and 4%. The reason for this much smaller slowdown is that the median filtering on the GPU benefits from the tile-based array indexing method. The implementation of the median filtering algorithm in QETIR is fairly straightforward and data locality is not taken into account, which causes the larger slowdown.

Finally, we consider the combination of all extensions. Interestingly, the reconstruction times of the implementation on the Tesla V100 almost doubled compared to the baseline OSEM algorithm without any extensions ($0.60 \rightarrow 1.17$). This 95% increase is larger than the sum of all increases (2%+35%+31%+7%=75%) observed for individual extensions on the baseline MLEM versions. The reason for this somewhat unexpected performance scaling is entirely architectural. Each extension increases the register pressure in the GPU code (i.e., the amount of GPU registers used in the code) a little bit because more computations need to be performed. For individual extensions, these increases are small enough not to have an effect on the utilization of computational resources while kernels are executing: each SM core can still handle 4 active blocks of concurrently running threads. When all extensions are combined, however, the register pressure increase above the threshold for running 4 active blocks of threads concurrently on the Tesla V100, instead dropping to 3 active blocks. In short, the observed performance hit originates from a resource limitation in the GPU architecture, not from an issue with the compiler or used programming language. On the RTX 2080 Ti, we similarly observe that the 68% increase in execution time for all extensions combined ($0.77 \rightarrow 1.29$) is also larger than the sum of their individual increases (1%+25%+10%+4%=41%). This follows from exactly the same cause.

In summary, except for reasons fundamental to GPU computing, we observed that adding extensions to the optimized Julia implementation of the baseline algorithm did not undo any of the optimizations and did not

[6] We have no execution times for the CPU version of MLEM and OSEM with TOF correction, due to a bug in the TOF correction part of QETIR, of which the fixing was out of scope of our work.

Algorithm **Processor**	**QETIR** **CPU**	**Julia** **Tesla V100**	**Julia** **RTX 2080 Ti**	**CPU vs.** **Tesla V100**	**CPU vs.** **RTX 2080 Ti**
Baseline MLEM without extensions	87.46s	0.97s	1.43s	x90	x61
MLEM with sensitivity correction	89.65s	0.99s	1.44s	x91	x62
OSEM without extensions	43.16s	0.60s	0.77s	x72	x56
MLEM with attenuation correction	93.60s	1.31s	1.79s	x71	x52
MLEM with TOF correction	N/A	1.27s	1.58s	N/A	N/A
MLEM with median filtering	120.89s	1.04s	1.49s	x116	x81
OSEM with all extensions	N/A	1.17s	1.29s	N/A	N/A

Table 3. Comparison of QETIR CPU and optimized Julia GPU reconstruction times. All versions use 32-bit floating point precision.

result in larger performance penalties than the ones in C++ CPU versions. We can hence conclude that our Julia implementations feature great performance portability over algorithmic variations.

Overall, we can conclude that the optimised GPU algorithms in Julia are well suited for image reconstruction of large dimensions, which is favourable for, e.g., total-body PET systems.

6.2. Programmer Productivity

Quantitative Evaluation. Table 4 presents the lines of code needed for four implementations of baseline MLEM algorithms: the QETIR C++ CPU implementation, our basic and optimized Julia GPU implementations, and the CUDA C implementation of the original slice-based ray tracing paper by Gao (2012). The latter only includes the ray-tracing kernels, however, as that was the focus of that paper. We note that lines of code is only a rough metric, which is biased by programming style. Furthermore, it only measures the final result of the developer's effort, not the amount of effort that was needed to get there. Still, we consider it useful as a first-order quantitative approximation of programmer effort.

The basic Julia implementation needs much fewer lines than QETIR. This is primarily caused by the ray tracing algorithm in C++: it contains many if-statements and temporary variables, resulting in 258 lines of code. The slice-based ray tracing algorithm in Julia is more compact and only has 95 lines of code. When comparing the MLEM algorithm of both implementations, they are about the same length. The profit of using Julia in terms of lines of code is cancelled out by the extra code that is needed for GPU-specific instructions. Still, as stated before, this implies that with Julia we get GPU support and optimized performance at virtually no cost in code complexity over C++.

The pre-existing slice-based CUDA C implementation from Gao (2012) has large, almost identical code blocks and contains 619 lines of code. Besides deduplication, their code is amendable to some space-saving refactorings, which would, by our estimates, bring the size down to approximately 120 lines. With 95 lines, our basic implementation in Julia is considerably simpler. Moreover, invoking kernels in CUDA C/C++ requires lots of boilerplate code outside the kernels to allocate memory on the GPU and to transfer data from CPU memory to GPU memory. In Julia, much less such GPU-specific code is necessary, as is clear from the work by Besard et al. (2019) and Besard et al. (2019), as well as from the already discussed comparison of Julia GPU code with QETIR C++ code for the part of the code implementing the MLEM algorithm. In summary, the basic Julia implementation is much smaller than the CUDA C implementation.

Compared to the basic Julia implementation, only 50 lines of additional code were needed for all the performance optimizations. The ray tracing function remained the same but 15 extra lines of code were added to the MLEM algorithm. These extra lines of code are needed to interact with shared memory and to call tile-based array re-indexing functions. The re-indexing functions themselves cause 24 extra lines of utility code. Finally, an extra structure definition was needed to store temporary results of slices.

We can conclude that even performance-optimized Julia GPU code is on par with C++ CPU-only code in terms of needed lines of code, as well as with CUDA C code, if not better. This, together with the fact that a single Julia code version could be used for different generations of GPUs, without manual re-tuning, provides strong evidence of the programming productivity boost that Julia can bring for the programming medical image reconstruction algorithms on GPUs over the use of lower-level languages.

For the algorithmic extensions, the Julia implementations required no tuning for specific GPUs either. As shown in Table 5, the extended Julia implementation still contains significantly fewer lines of code than the

	Lines of code
QETIR C++ for CPU (total)	**394**
↳ MLEM	60
↳ Ray tracing	258
↳ Utility functions	36
↳ Definition of classes	40
Basic Julia for GPU (total)	**228**
↳ MLEM	63
↳ Ray tracing	95
↳ Utility functions	50
↳ Definition of structures	20
Optimized Julia for GPU (total)	**278**
↳ MLEM	78
↳ Ray tracing	95
↳ Utility functions	74
↳ Definition of structures	31
CUDA C for GPU (total)	**N/A**
↳ MLEM	N/A
↳ Ray tracing	619
↳ Ray tracing with code compaction refactorings and deduplication	$\sim$ 120

Table 4. Lines of code of the C++, CUDA, and Julia implementations of the baseline MLEM algorithm

equivalent QETIR C++ implementation.[7] Our overall conclusion is that Julia enables GPU performance portability and good resource utilization for PET MLEM-based image reconstruction while at the same time offering much more productive programming than plain C++ and a fortiori than CUDA C/C++, in particular when domain researchers want to explore algorithmic variations.

All in all, the Julia language proved to be mature enough to implement the most basic version of the baseline MLEM algorithm and run it on a GPU. We were able to use some of the high-level features offered by the CUDA.jl package. Compared to CUDA C, the CuArrays data type freed us from manually managing device memory and the element-wise multiplication on the GPU is completely abstracted away.

Nevertheless, the Julia GPU ecosystem does not fully relieve the programmer from the task of performance optimization. The forward projection, compare, and back projection steps require the collaboration and the communication between multiple threads. As the GPU abstractions offered by CUDA.jl build on a set of high-level array operations as proposed in Besard et al. (2019) and Besard et al. (2019), and as ray-tracing does not map directly onto those array operations, we needed to program GPUs at a slightly lower level of abstraction, i.e., at that of kernels. To do so, some knowledge of GPUs and their programming model was still needed, such as the memory hierarchy and how the CUDA execution model maps onto the GPU hardware. This is much less and still more abstract than what needs to be known for programming in CUDA C, and therefore much easier to grasp for non-GPU experts.

Ease of Programming. Fortunately, the Julia programming environment also helps the programmer with the writing and optimization of code, i.e., on the path from scratch to the final optimized code that we analyzed so

[7] The QETIR line count is an estimate. The QETIR code base includes more algorithmic extensions than we evaluated in our research. We excluded the lines needed for those additional algorithmic extensions, but it might very well be that the code would have looked different, possibly with somewhat fewer lines of code, when the developers would not have included those additional extensions at all. The exact number of lines is of no importance with respect to our overall conclusions.

	Without extensions	With extensions
QETIR C++ on CPU (total)	394	~ 636
Julia on GPU (total)	228	416

Table 5. Lines of code of the MLEM algorithm without and with extensions

far. Firstly, the REPL (read-eval-print loop), Julia's interactive command-line interface, provides an easy way to write and debug code. The REPL is comparable to the Python Interpreter. For example, the REPL provides a much faster way to make changes to a kernel and test the result of the change compared to CUDA C. In CUDA C, code must be re-compiled and the entire application must be re-executed. In Julia, the programmer only needs to re-run the affected function. Through its seamless interaction with existing profiling tools, such as NVIDIA Nsight, the REPL also enables iteractive tuning, where adjusted kernels can be profiled immediately to analyse the impact of the adjustments. This is definitely beneficial in a research domain in which execution times and output quality are both important optimization objectives.

Secondly, the visualization of a 3D image is much more programmer-friendly in Julia: thanks to the Images.jl package, only one line of code is needed after installing and importing the package. The PyPlot.jl package offers an interface to the well-known Matplotlib library in Python. This allows people familiar with Matplotlib to get started quickly without learning to work with a new visualization tool.

Thirdly, Julia also comes with a fully featured built-in package manager, a feature that is missing in most low-level languages. In most low-level languages, every package has its own installation procedure. The programmer has to keep track which version of each library is installed. Recreating the same environment on a different computer is a daunting task.

These benefits of the Julia ecosystem enabled the first author of this work to perform the reported research and to obtain the presented result within the course of a 24-credit master thesis, even though the student was unfamiliar with Julia, CUDA, PET, and MLEM before starting his thesis project in September 2019.

The results of this section and the previous section hence clearly show that Julia can solve the two-language problem in this domain: an expert in the field of image reconstruction can first write an existing or new algorithm in Julia without having to take performance into account. Afterwards, a somewhat more GPU savvy developer can tune the new algorithm without having to re-write it into a low-level language, instead staying in the Julia language. To implement the extension on the baseline MLEM algorithm, the multiple-dispatch mechanism in Julia proved to be useful and enabled efficient code re-use. Because most code could be re-used, the resource utilization remained fairly insensitive to the extensions. It must, however, be taken into account that register usage can change, which has an influence on the optimal launch configuration.

Unfortunately, support for constant memory, which could improve the execution time of the TOF kernel, was missing in Julia when we conducted our research. Since then, development of constant memory support has started (in our research group), but is as of yet immature.

Finally, we come back to the re-use of an existing median filtering implementation written in CUDA C in our Julia code base. With this reuse, we showed that research groups that have an existing medical image reconstruction codebase in CUDA C/C++ do not need to rewrite all their kernels in Julia to make the switch. With the Julia GPU ecosystem, existing CUDA C/C++ kernels can be called and used in Julia. In this way, their existing work is not lost, while they can already benefit from the advantages that Julia offers during the further development of their application.

Higher-Level Programming Outlook. As discussed above, we implemented MLEM PET image reconstruction by writing code at the level of GPU kernels. With this form of implementation, researchers that want to explore algorithmic extensions or alternatives still need to implement those at the same, relatively low level of abstraction, which does require GPU knowledge, and which requires knowledge of how equation 4 is implemented by means of on-the-fly ray tracing to enable massive parallelism and to avoid storage of large sparse matrices.

By supporting the overloading (i.e., redefinition) of array index functions, by means of its multiple dispatch support, and by means of its meta-progamming capabilities, Julia also enables lazy array operations and matrix-free methods for iterative solvers. Existing packages such as LazyArrays.jl, LinearMaps.jl, LinearMapsAA.jl, and LinearOperators.jl provide higher-level interfaces and data types that allow Julia programmers to express their computations at the mathematical level of abstraction, i.e., in terms of (system) matrices and matrix operators, and to have those operators and compositions thereof implemented by means of various optimized algorithms that avoid storing large (intermediate) matrices.

Several examples exist of such high-level programming in the Julia ecosystem. Knopp and Grosser (2021) developed such interfaces, data types, and operator implementations for CPU-based MRI image reconstruction. Besard et al. (2019) demonstrated that one can enable programmers to write high-level code in terms of arrays, matrices, and operators such as Kronecker products, while still having that code automatically compiled into efficient GPU code. Faingnaert et al. (2020) provides an example where general matrix multiplications (GEMM) computed on GPUs automatically get optimized when they are invoked on, e.g., diagonal matrices.

It remains future work to design and implement comparable high-level interfaces to the algorithms and their implementations presented in this paper, which would allow researchers to express their PET image reconstruction algorithms at the mathematical level of abstraction we used throughout Section 2 and using high-level notation similar to the one we mention in the paragraph on sensitivity correction in Section 5. Given the successes in the aforementioned computational application domains, we have no doubt that such a design and implementation is feasible. The design requires a more extensive study of existing variations of (MLEM) PET image reconstruction algorithms to ensure that all interesting algorithmic explorations are covered. That study is out-of-scope of this paper. Once a high-level interface design would be chosen, however, we envision that implementing the functionality beneath the interfaces will involve only relatively straightforward software engineering and that the GPU kernels developed in this paper will help to implement this functionality for use on GPUs.

7. Discussion and Conclusion

In the Julia programming language, we first developed a basic GPU-enabled implementation of the baseline MLEM algorithm for PET image reconstruction. We then optimized that code for performance, after which we also extended it with five existing algorithmic improvements.

We thus showed that the Julia programming language is mature enough to implement image reconstruction algorithms and to execute them on a GPU with good resource utilization. The Julia GPU ecosystem offers enough high-level functionality to optimise the performance to a more than acceptable level without requiring expert knowledge about GPU architectures and their programming models. Performance portability across GPU generations and algorithmic variations was achieved. Support for some GPU features that could increase the performance even further is still missing, but the impact is probably minor.

The two-language problem in the domain of medical image reconstruction can therefore mostly be avoided by using the Julia programming language. Different high-level abstractions and language features enable fast-prototyping. The two-language problem is not completely solved within Julia, however. For media filtering, the

available alternatives were reuse of non-Julia code or system-level programming in Julia that required expertise in GPU programming models and Julia's underlying GPU support, neither of which are optimal.

Reusing existing CUDA C median filtering implementations, we demonstrated that switching to Julia as the programming language of choice does not prevent re-use of existing functionality written in other languages. This is particularly relevant as long as Julia is an upcoming language that lacks the maturity of more established languages, and in which domain-specific libraries, although expanding rapidly, are often not as feature-complete as those for other languages, and or not yet developed to be compatible with GPU support libraries.

In our approach so far, there was still a need for the manual implementation of kernels, rather than expressing the algorithms at more abstract, mathematical levels, but this could be done more efficiently in Julia than in CUDA C/C++ because less boilerplate code is needed.

During our investigation in the usability of Julia for medical image reconstruction on GPUs, we added a new array storage and indexing technique that improved performance significantly. We also showed that our implementation with current state-of-the-art hardware scales well for large image dimensions. This is beneficial in the context of total-body pet systems.

Overall, our conclusion is that switching to Julia will likely boost the productivity of researchers in the domain of PET image reconstruction, and hence quite possibly also in other domains of medical imaging.

In the future, when high-level interfaces are developed on top of our current implementations, an additional boost in productivity can likely be achieved.

Declaration of Conflicting Interest

The Author(s) declare(s) that there is no conflict of interest.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Research Foundation Flanders (Fonds voor Wetenschappelijk Onderzoek) [grant number 3G051318].

References

S. Ahn, S. G. Ross, E. Asma, J. Miao, X. Jin, L. Cheng, S. D. Wollenweber, and R. M. Manjeshwar. Quantitative comparison of OSEM and penalized likelihood image reconstruction using relative difference penalties for clinical PET. *Physics in Medicine and Biology*, 60(15):5733–5751, jul 2015. . URL `https://doi.org/10.1088/0031-9155/60/15/5733`.

H. Arabi and H. Zaidi. Improvement of image quality in PET using post-reconstruction hybrid spatial-frequency domain filtering. *Physics in Medicine & Biology*, 63(21):215010, Oct. 2018. ISSN 0031-9155. . URL `https://doi.org/10.1088%2F1361-6560%2Faae573`. Publisher: IOP Publishing.

D. L. Bailey, J. S. Karp, and S. Surti. Physics and Instrumentation in PET. In D. L. Bailey, D. W. Townsend, P. E. Valk, and M. N. Maisey, editors, *Positron Emission Tomography: Basic Sciences*, pages 13–39. Springer London, London, 2005a. ISBN 978-1-84628-007-8. . URL `https://doi.org/10.1007/1-84628-007-9_2`.

D. L. Bailey, M. N. Maisey, D. W. Townsend, and P. E. Valk. *Positron emission tomography*, volume 2. Springer, 2005b.

H. H. Barrett, T. White, and L. C. Parra. List-mode likelihood. *Journal of the Optical Society of America. A, Optics, image science, and vision*, 14(11):2914–2923, Nov. 1997. ISSN 1084-7529. URL `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2969184/`.

L. Bavoil. The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload, June 2019. URL https://devblogs.nvidia.com/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/.

E. Berg, X. Zhang, J. Bec, M. S. Judenhofer, B. Patel, et al. Development and Evaluation of mini-EXPLORER: A Long Axial Field-of-View PET Scanner for Nonhuman Primate Imaging. *Journal of Nuclear Medicine: Official Publication, Society of Nuclear Medicine*, 59(6), 2018. ISSN 1535-5667. .

T. Besard, V. Churavy, A. Edelman, and B. De Sutter. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, 132:29 – 46, 2019. ISSN 0965-9978. . URL http://www.sciencedirect.com/science/article/pii/S0965997818310123.

T. Besard, C. Foket, and B. De Sutter. Effective extensible programming: Unleashing julia on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, April 2019. ISSN 2161-9883. .

J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. URL http://arxiv.org/abs/1209.5145.

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014. URL http://arxiv.org/abs/1411.1607.

M. Christiaens, B. De Sutter, K. De Bosschere, J. Van Campenhout, and I. Lemahieu. A fast, cache-aware algorithm for the calculation of radiological paths exploiting subword parallelism. *Journal of Systems Architecture*, 45(10):781–790, 4 1999.

J.-Y. Cui, G. Pratx, S. Prevrhal, and C. S. Levin. Fully 3D list-mode time-of-flight PET image reconstruction on GPUs using CUDA. *Medical Physics*, 38(12):6775–6786, 2011. ISSN 2473-4209. . URL https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.3661998.

B. De Sutter, M. Christiaens, K. De Bosschere, and J. Van Campenhout. On the use of subword parallelism in medical image processing. *Parallel Computing*, 24(9-10):1537–1556, 1998.

N. Efthimiou, E. Emond, P. Wadhwa, C. Cawthorne, C. Tsoumpas, and K. Thielemans. Implementation and validation of time-of-flight PET image reconstruction module for listmode and sinogram projection data in the STIR library. *Physics in Medicine & Biology*, 64(3):035004, Jan. 2019. ISSN 0031-9155. . URL https://doi.org/10.1088%2F1361-6560%2Faaf9b9. Publisher: IOP Publishing.

T. Faingnaert, T. Besard, and B. De Sutter. Flexible Performant GEMM Kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2020. URL https://arxiv.org/abs/2009.12263. Paper under submission.

H. Gao. Fast parallel algorithms for the x-ray transform and its adjoint. *Medical Physics*, 39(11):7110–7120, Nov. 2012. ISSN 0094-2405. . URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3505201/.

Z.-G. Gui and J. He. Regularized maximum likelihood algorithm for PET image reconstruction using a detail and edges preserving anisotropic diffusion. *Optik*, 123(6):507–510, Mar. 2012. ISSN 0030-4026. . URL http://www.sciencedirect.com/science/article/pii/S0030402611002488.

T. Hebert and R. Leahy. Fast methods for including attenuation in the EM algorithm. *IEEE Transactions on Nuclear Science*, 37 (2):754–758, Apr. 1990. ISSN 1558-1578. . Conference Name: IEEE Transactions on Nuclear Science.

H. M. Hudson and R. S. Larkin. Accelerated image reconstruction using ordered subsets of projection data. *IEEE transactions on medical imaging*, 13(4):601–609, 1994. ISSN 0278-0062. .

S. Jan, G. Santin, D. Strul, S. Staelens, K. Assié, D. Autret, et al. GATE: a simulation toolkit for PET and SPECT. *Physics in Medicine and Biology*, 49(19):4543–4561, Oct. 2004. ISSN 0031-9155. .

Khronos Group. OpenCL: An open standard for parallel programming of heterogeneous systems, 2020.

H. S. Kim, S.-G. Cho, J. H. Kim, S. Y. Kwon, B.-i. Lee, and H.-S. Bom. Effect of Post-Reconstruction Gaussian Filtering on Image Quality and Myocardial Blood Flow Measurement with N-13 Ammonia PET. *Asia Oceania Journal of Nuclear Medicine and Biology*, 2(2):104–110, 2014. ISSN 2322-5718. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4937694/.

T. Knopp and M. Grosser. MRIReco.jl: An MRI reconstruction framework written in Julia. *Magnetic Resonance in Medicine*, 86 (10):1633–1646, 2021.

M. Kolstein and M. Chmeissani. Using triple gamma coincidences with a pixelated semiconductor compton-pet scanner: a simulation study. *Journal of Instrumentation*, 11(01):C01039, 2016.

K. Lange and R. Carson. EM Reconstruction Algorithms for Emission and Transmission Tomography. *Journal of computer assisted tomography*, 8:306–316, May 1984.

C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

N. A. Mullani, J. Markham, and M. M. Ter-Pogossian. Feasibility of time-of-flight reconstruction in positron emission tomography. *Journal of Nuclear Medicine: Official Publication, Society of Nuclear Medicine*, 21(11):1095–1097, Nov. 1980. ISSN 0161-5505.

NVIDIA. cuBLAS: CUDA toolkit documentation, 2020a.

NVIDIA. CUDA C++ programming guide, 2020b.

L. Parra and H. Barrett. List-mode likelihood: EM algorithm and image quality estimation demonstrated on 2-D PET. *IEEE Transactions on Medical Imaging*, 17(2):228–235, Apr. 1998. ISSN 1558-254X. . Conference Name: IEEE Transactions on Medical Imaging.

G. Pratx, G. Chinn, P. Olcott, and C. Levin. Fast, Accurate and Shift-Varying Line Projections for Iterative Reconstruction Using the GPU. *IEEE transactions on medical imaging*, 28:435–45, Apr. 2009. .

J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, Upper Saddle River, NJ, 2010. ISBN 9780131387683 0131387685.

C. Schretter. A fast tube of response ray-tracer. *Medical physics*, 33:4744–8, Jan. 2007. .

L. A. Shepp and Y. Vardi. Maximum likelihood reconstruction for emission tomography. *IEEE transactions on medical imaging*, 1(2):113–122, 1982.

R. L. Siddon. Fast calculation of the exact radiological path for a three-dimensional CT array. *Medical Physics*, 12(2):252–255, Apr. 1985. ISSN 0094-2405. .

H. Thoen, V. Keereman, P. Mollet, R. Van Holen, and S. Vandenberghe. Influence of detector pixel size, tof resolution and doi on image quality in mr-compatible whole-body pet. *Physics in Medicine & Biology*, 58(18):6459, 2013.

C. Thyssen, R. Van Holen, and S. Vandenberghe. Monte Carlo simulations of total-body PET systems, 2018. URL `http://lib.ugent.be/catalog/rug01:002494587`.

V. Volkov. Better Performance at Lower Occupancy, Sept. 2010. URL `https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf`.

H. Watabe, S.-K. Woo, K.-M. Ki, H. Matsuura, K. Matsumoto, P. Bloomfield, and H. Iida. Performance of list-mode data acquisition with ECAT EXACT HR positron emission scanner. In *2002 IEEE Nuclear Science Symposium Conference Record*, volume 2, pages 970–973 vol.2, Nov. 2002. .

F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science*, 52(3):654–663, June 2005. ISSN 1558-1578. . Conference Name: IEEE Transactions on Nuclear Science.

H. Zhao and A. Reader. Fast projection algorithm for voxel arrays with object dependent boundaries. In *2002 IEEE Nuclear Science Symposium Conference Record*, volume 3, pages 1490–1494 vol.3, Nov. 2002. .

J. Zhou and J. Qi. Fast and efficient fully 3D PET image reconstruction using sparse system matrix factorization with GPU acceleration. *Physics in medicine and biology*, 56(20):6739–6757, Oct. 2011. ISSN 0031-9155. . URL `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4080908/`.