Siddon–Jacobs ray tracing on GPUs: a technical note on a consistent and accurate implementation

Ander Biguri

Department of Applied Mathematics and Theoretical Physics (DAMTP), University of Cambridge

November 2026, revised for TIGRE 3.3.1

TIGRE technical note. Accompanies CERN/TIGRE pull requests #865 (issue #859) and, for Section 4.6, #913.

Abstract

This technical note documents a revision of the ray–voxel (Siddon) forward projector of the TIGRE toolbox, for anyone implementing the incremental variant of Siddon's algorithm by Jacobs et al. on a GPU. The variant, as usually implemented, has two flaws. Its published initialisation is inconsistent, in exact arithmetic, for rays that enter the image on a voxel edge, which then are integrated one voxel off; we characterise these rays exactly. And in single precision its incremental update accumulates rounding errors along the ray. Both have known remedies, the initialisation of Amanatides and Woo and the direct evaluation of crossings of Siddon's original method. We show how to apply them on a GPU at the cost of Jacobs' traversal. The revision also makes the projector 1.4–6.5 times faster on the GPUs tested, through the arrangement of the loop and the type of its indices. The three issues, consistency, accuracy and speed, have independent causes, and the speed could have been obtained while keeping Jacobs' traversal; but the revised traversal addresses all three at once, and the arrangement that makes it fast is also what makes the two fixes free. In TIGRE the projections are about 100 times more accurate and consistent for every ray. A revision for TIGRE 3.3.1 (Section 4.6) unrolls the loop into four branch-free steps, so that several memory reads are in flight at once, which makes the kernel a further 1.1–1.2 times faster with the same results.

Keywords: computed tomography, forward projection, Siddon's algorithm, ray tracing, voxel traversal, GPU, floating point.

1 Introduction

Iterative reconstruction in computed tomography (CT) rests on two operators, the forward projection $A\mathbf{x}$ and the backprojection $A^T\mathbf{b}$, applied hundreds of times per reconstruction. Ray-driven forward projectors integrate the image along each X-ray path; among them, the exact path-length method computes, for every ray, the length of its intersection with each voxel it crosses, i.e. the nonzero entries of the corresponding row of the system matrix. Siddon's algorithm [1] made this practical by working in the parameter of the ray: the ray crosses three families of equally spaced planes, so the intersection lengths are the differences between consecutive plane crossings, at a cost proportional to the number of planes crossed rather than to the number of voxels. Jacobs et al. [2] made it faster by replacing the sorting of crossings and the per-segment voxel index computation with an incremental traversal, which is essentially the voxel traversal of Amanatides and Woo [3] used in computer graphics. This incremental form is

the one usually implemented, and on GPUs the implementations are much alike: one thread per ray, the image in a texture, single-precision arithmetic, and the image processed in slabs when it does not fit in GPU memory. The TIGRE toolbox [4, 5, 6] is one such implementation, and the one we use as reference throughout.

This note documents what we found when an issue in TIGRE (#859: wrong projections when the image is split in slabs) led us to re-examine its implementation, and how we changed it. It is a technical note, not a new method: the ingredients of the revised traversal all exist in the literature, and we say where each comes from. Its content is:

1. An exact characterisation of the rays for which Jacobs' initialisation is inconsistent, independent of floating point (Section 3.1). This is the cause of #859, and it also affects whole images.

2. An analysis of the rounding errors of the incremental update in single precision, which grow with the number of steps (Section 3.2).

3. A traversal that avoids both, combining the initialisation of Amanatides and Woo [3] with the direct evaluation of crossings of Siddon [1], and arranged for SIMT execution on GPUs (Section 4).

4. An evaluation of correctness, accuracy and speed in TIGRE, including an ablation that separates the effect of each change on the GPU (Section 5).

The revision also made TIGRE's projector considerably faster, for reasons unrelated to the two flaws. Section 4.5 separates the three issues: they have independent causes, but one traversal addresses them together. Section 2 recalls both algorithms and the GPU setting, and Section 6 discusses why the problems went unnoticed, how other implementations deal with them, and the limitations of this study.

2 Background

2.1 Notation

We follow the notation of [7] (Section 4.3.1), in 3D. Coordinates are in voxel units, with the corner of the first voxel at the origin, so that voxel (i, j, k) occupies $[i, i+1) \times [j, j+1) \times [k, k+1)$. The image has $n_x \times n_y \times n_z$ voxels, bounded by the planes $x = 0, \ldots, n_x$ and likewise in y and z. A ray goes from the source $\mathbf{p}_1$ to a detector pixel $\mathbf{p}_2$,

$$\mathbf{p}(\alpha) = \mathbf{p}_1 + \alpha\,\mathbf{r}, \qquad \mathbf{r} = \mathbf{p}_2 - \mathbf{p}_1, \qquad \alpha \in [0, 1]. \tag{1}$$

The parameter at which it crosses the plane $x = b$ is

$$\alpha_x(b) = \frac{b - p_{x1}}{r_x}, \tag{2}$$

and likewise for y and z. The ray is inside the image for $\alpha \in [\alpha_{min}, \alpha_{max}]$,

$$\begin{aligned} \alpha_{min} &= \max_d \alpha_{d\,min}, \quad \alpha_{d\,min} = \min\big(\alpha_d(0), \alpha_d(n_d)\big), \\ \alpha_{max} &= \min_d \alpha_{d\,max}, \quad \alpha_{d\,max} = \max\big(\alpha_d(0), \alpha_d(n_d)\big), \end{aligned} \tag{3}$$

over the axes $d \in \{x, y, z\}$ with $r_d \neq 0$. We call $\mathbf{e} = \mathbf{p}(\alpha_{min})$ the entry point, and an axis a with $\alpha_{min} = \alpha_{a\,min}$ an entry axis: the ray enters through a face normal to a, the near face $a = 0$ if

$r_a > 0$ or the far face $a = n_a$ if $r_a < 0$. $s_d = \mathrm{sign}(r_d)$ is the direction of travel along d. With $\alpha_{min} = \alpha_0 < \alpha_1 < \cdots < \alpha_M = \alpha_{max}$ the parameters of the planes crossed, the line integral is

$$d_{ray} = \|\mathbf{r}\|_\Delta \sum_{m=1}^{M} (\alpha_m - \alpha_{m-1})\, \mathbb{I}(\mathbf{v}_m), \tag{4}$$

where $\mathbf{v}_m$ is the voxel between α_{m-1} and α_m, $\mathbb{I}$ is the image (zero outside it) and $\|\mathbf{r}\|_\Delta = (\sum_d r_d^2 \Delta_d^2)^{1/2}$ is the length of the ray in millimetres for voxel sizes Δ_d.

For floating point we use the standard model [8]: $\mathrm{fl}(\cdot)$ denotes an expression evaluated in floating point, each operation rounded to the nearest representable number, so that $\mathrm{fl}(a \circ b) = (a \circ b)(1+\delta)$ with $|\delta| \le \mathrm{u}$ for $\circ \in \{+, -, \times, /\}$. In single precision $\mathrm{u} = 2^{-24} \approx 6 \times 10^{-8}$. A fused multiply–add (FMA) rounds once, $\mathrm{fl}(ab + c) = (ab + c)(1 + \delta)$.

2.2 Siddon's algorithm and the incremental variant of Jacobs et al.

Siddon [1] computes all the crossings in $(\alpha_{min}, \alpha_{max})$ with Eq. (2), merges the three sorted sequences, and assigns each segment the voxel containing its midpoint $\alpha_{mid} = (\alpha_{m-1} + \alpha_m)/2$:

$$\mathbf{v}_m = \big(\lfloor p_x(\alpha_{mid}) \rfloor, \lfloor p_y(\alpha_{mid}) \rfloor, \lfloor p_z(\alpha_{mid}) \rfloor\big). \tag{5}$$

The midpoint of a segment of positive length is strictly inside its voxel, so Eq. (5) is exact; a segment of length zero (where the ray crosses two planes at once) may be assigned any voxel, since it contributes nothing. The cost is the merge and, per segment, the voxel index: for each axis a multiply–add and a conversion to an integer.

Jacobs et al. [2] avoid both by following the ray from voxel to voxel. We give the algorithm as in [7], where it is described in 2D, written here in 3D and in our notation. Its variables are shown in Figure 2(a), in Section 4, next to those of the revised traversal. For each axis, the first and last planes crossed are found from the entry and exit points. For $r_x > 0$,

$$i_{min} = \begin{cases} 1 & \alpha_{min} = \alpha_{x\,min} \\ \lceil p_x(\alpha_{min}) \rceil & \text{otherwise} \end{cases}, \qquad i_{max} = \begin{cases} n_x & \alpha_{max} = \alpha_{x\,max} \\ \lfloor p_x(\alpha_{max}) \rfloor & \text{otherwise} \end{cases}, \tag{6}$$

and for $r_x < 0$,

$$i_{max} = \begin{cases} n_x - 1 & \alpha_{min} = \alpha_{x\,min} \\ \lfloor p_x(\alpha_{min}) \rfloor & \text{otherwise} \end{cases}, \qquad i_{min} = \begin{cases} 0 & \alpha_{max} = \alpha_{x\,max} \\ \lceil p_x(\alpha_{max}) \rceil & \text{otherwise} \end{cases}, \tag{7}$$

and likewise $j_{min}, j_{max}, k_{min}, k_{max}$. The first crossing on each axis and the number of crossings are

$$\alpha_x = \begin{cases} \alpha_x(i_{min}) & r_x > 0 \\ \alpha_x(i_{max}) & r_x < 0 \end{cases}, \qquad N_p = (i_{max} - i_{min} + 1) + (j_{max} - j_{min} + 1) + (k_{max} - k_{min} + 1). \tag{8}$$

The first voxel is given by Eq. (5) applied to the first segment,

$$(i, j, k) = \big(\lfloor p_x(\alpha_{mid}) \rfloor, \lfloor p_y(\alpha_{mid}) \rfloor, \lfloor p_z(\alpha_{mid}) \rfloor\big), \qquad \alpha_{mid} = \tfrac{1}{2}\big(\alpha_{min} + \min(\alpha_x, \alpha_y, \alpha_z)\big). \tag{9}$$

Consecutive crossings of the same axis are a constant parameter step apart, and the index moves in the direction of travel:

$$\alpha_{xu} = \frac{1}{|r_x|}, \qquad i_u = s_x, \tag{10}$$

and likewise for y and z. Starting from $\alpha_c = \alpha_{min}$ and $d_{ray} = 0$, the traversal takes N_p steps. In each, the axis with the smallest α_d is crossed; if it is x,

$$d_{ray} \leftarrow d_{ray} + (\alpha_x - \alpha_c)\, \mathbb{I}(i, j, k), \qquad i \leftarrow i + i_u, \qquad \alpha_c \leftarrow \alpha_x, \qquad \alpha_x \leftarrow \alpha_x + \alpha_{xu}, \tag{11}$$

and likewise for y and z. Finally d_{ray} is multiplied by $\|\mathbf{r}\|_\Delta$. When the ray crosses two planes at once, the second crossing has length zero, so no special case is needed [7]. Algorithm 1 (left) collects Eqs. (6)–(11).

Amanatides and Woo's traversal [3] is the same loop (t_{Max} for α_x, t_{Delta} for α_{xu}), with the first voxel taken as the voxel containing the entry point. We compare the cost per step of the three algorithms in Section 4.4.

2.3 The usual GPU implementation

GPU implementations of Jacobs' algorithm share a common structure, which we describe with TIGRE's choices [4, 7]. Each thread traces one ray, with neighbouring threads tracing neighbouring rays so that they read nearby voxels; source and detector are expressed in voxel coordinates; the image is a 3D texture read with point sampling, which returns 0 outside it; and the arithmetic is in single precision. When the image does not fit in GPU memory, or is distributed over several GPUs, it is split into slabs of consecutive slices along z, each projected as an image of its own, and the projections are added [5]; a ray crossing from one slab to the next enters it through a z face. We call TIGRE's implementation before the changes described here the original kernel.

3 Two flaws of the incremental traversal

3.1 Inconsistent initialisation

Equations (6)–(8) and Eq. (9) compute two things that must agree, the next plane on each axis and the first voxel, by two independent routes: the planes from $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ of the entry point, the voxel from $\lfloor\cdot\rfloor$ of a midpoint. When the first segment has positive length its midpoint is inside the first voxel and the two agree. The first segment has zero length exactly when the entry point lies on a plane of an axis that is not an entry axis: that plane is then taken as the first crossing, at α_{min}, and the midpoint is the entry point itself. On a plane, $\lfloor\cdot\rfloor$ selects the voxel on the positive side, whatever the direction of travel.

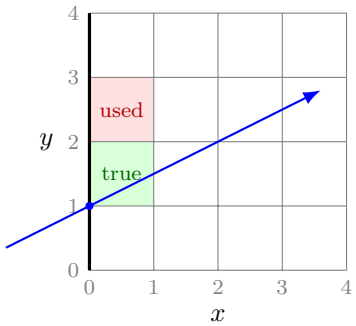
Proposition 1. *In exact arithmetic, let the entry point* $\mathbf{e}$ *lie on a plane of an axis* b *that is not an entry axis,* $e_b \in \mathbb{Z}$*, with* $r_b \neq 0$*. Then:*

(i) if $r_b > 0$*, the index along* b *starts correct, the zero-length crossing of the plane* e_b *increments it, and every later voxel is one voxel ahead along* b*;*

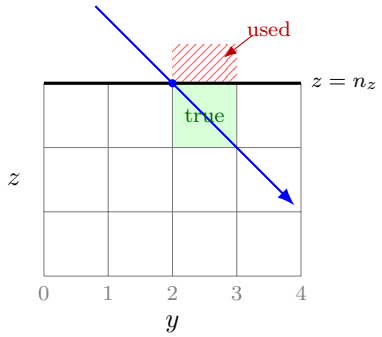
(ii) if the ray enters through a far face $a = n_a$ *(*$r_a < 0$*), the midpoint rule gives* $\lfloor e_a \rfloor = n_a$*, outside the image, while the crossings along* a *start from the plane* $n_a - 1$*; every later voxel is one voxel off along* a *(and the first segment reads zero).*

In all other cases, including $r_b < 0$ *with entry through a near face, the traversal is exact.*

Proof sketch. Let b be as stated, and write the argument for $b = y$. For $r_y > 0$, Eq. (6) gives $j_{min} = \lceil e_y \rceil = e_y$, so by Eq. (8) the first y crossing is $\alpha_y(e_y) = \alpha_{min}$. Then $\alpha_{mid} = \alpha_{min}$ in Eq. (9), which gives $j = \lfloor e_y \rfloor = e_y$, the correct voxel for $r_y > 0$. The first step, Eq. (11), processes the crossing at α_{min} (length zero) and increments j to $e_y + 1$ while the ray is still in voxel e_y; the next y crossing is the plane $e_y + 1$, so the offset is never corrected. For $r_y < 0$, Eq. (7) gives $j_{max} = \lfloor e_y \rfloor = e_y$ and Eq. (9) the index e_y, one too high; the zero-length crossing of the plane e_y then decrements it to $e_y - 1$, the correct voxel. For the entry axis a in case (ii), Eq. (7) sets $a_{max} = n_a - 1$, the far face of voxel $n_a - 1$, while Eq. (9) gives $\lfloor e_a \rfloor = n_a$; the first crossing along a then decrements the index to $n_a - 1$ when the ray enters voxel $n_a - 2$. For a near face, $\lfloor e_a \rfloor = 0$ is correct. If no non-entry coordinate of $\mathbf{e}$ is an integer, the first segment has positive length and its midpoint is inside the first voxel. □



(a) Case (i). Entry through the near face $x = 0$ at $y = 1$, moving up. The plane $y = 1$ is counted as a crossing of length zero, so after the first step the index is $j = 2$ while the ray is in $j = 1$.



(b) Case (ii). Entry through the far face $z = n_z$, moving down, at a point also on the plane $y = 2$. The midpoint is the entry point and $\lfloor n_z \rfloor$ selects the voxel above the face; the z crossings were computed for the voxel below.

Figure 1: The two cases of Proposition 1, in 2D sections.

We verified Proposition 1 exhaustively in rational arithmetic: for 52 811 rays through a $4 \times 5 \times 3$ grid, with entry points on a quarter-voxel lattice (so that many lie on planes) and random voxel values, Jacobs' traversal differed from Siddon's for exactly the rays predicted. Figure 1 illustrates the two cases. In both, the entry point lies on an edge of the voxel grid, where the entry face meets a plane of another axis (in 2D, a pixel corner); entering anywhere else on a face is harmless.

Frequency. The rays of Proposition 1 form a set of measure zero, and for a generic geometry no ray lies on it exactly. In TIGRE's default cone-beam geometry (256^3 voxels, a 256^2 detector, 360 angles), not one of the 23.6 million rays of the whole image satisfies the condition in exact arithmetic. Two things nevertheless put rays on it in practice. First, floating point turns the set into a band: an entry point within rounding of a grid edge behaves as if it were on it, and so does a comparison such as $\alpha_{min} = \alpha_{z\,min}$ in Eq. (7) that rounding decides the other way. The band is about one unit in the last place of the coordinates wide, $\sim 10^{-5}$ voxels, and in the same geometry 372 rays have an entry point within one such unit of a failing edge, about one in 60 000, in line with the 424 wrong rays observed (Section 5.2). Second, geometries aligned with the grid can place rays on the set exactly. When the image is split into slabs, the slab faces are an integer number of voxels from the plane of the source, and at angles that are multiples of 90° the detector axes are parallel to the image axes; the coordinates of entry points on a slab face are then rational numbers with small denominators, often integers. Split into three slabs, the same geometry has 1 074 rays that satisfy the condition exactly, all at 0°, 90°, 180° and 270°, in addition to the floating-point band. Splitting also multiplies the opportunities: a whole image is entered once per ray, whereas a ray crossing S slabs enters S of them. Splitting is therefore not the cause of #859, but it is what made it frequent.

In short, exact hits are an edge case of symmetric, grid-aligned geometries, which a calibrated scanner geometry will practically never produce. The floating-point band, however, does not depend on the geometry: in single precision, a small fraction of rays, of the order of 10^{-5}, lies within rounding of the set in any geometry, and each of them is integrated one voxel off for the rest of its path.

Remark. Siddon's original algorithm is not affected: every segment gets its own midpoint, Eq. (5), and the segment that follows a zero-length one has a midpoint inside its voxel again. The inconsistency is introduced by taking a single midpoint, Eq. (9), and then only incrementing.

3.2 Accumulation of rounding errors

In floating point the update of Eq. (11) is $\alpha_x \leftarrow \mathrm{fl}(\alpha_x + \alpha_{xu})$, which adds a small number to a large one. In voxel coordinates $\|\mathbf{r}\|$ is of the order of the source–detector distance in voxels, e.g. ≈ 3000 for a 512^3 image of 0.5 mm voxels and a source–detector distance of 1536 mm, so one voxel step changes α by $\alpha_{xu} = 1/|r_x| \gtrsim 3 \times 10^{-4}$, while inside the image α is about DSO/DSD ≈ 0.65. Each addition is rounded to a multiple of $\mathrm{ulp}(0.65) \approx 6 \times 10^{-8}$, an error of up to $\sim 10^{-4}$ of a voxel step. After m steps the computed crossing has accumulated up to m such errors, and typically $\sim \sqrt{m}$ of them; at 512^3, with a few hundred steps per axis, crossings end up 10^{-3} to 10^{-2} of a voxel step from their exact position.

A boundary displaced by $\delta\alpha$ moves a length $\|\mathbf{r}\|_\Delta\, \delta\alpha$ from one voxel to its neighbour, so by Eq. (4) the error in d_{ray} is

$$\delta d_{ray} \approx \|\mathbf{r}\|_\Delta \sum_m \delta\alpha_m \big(\mathbb{I}(\mathbf{v}_m) - \mathbb{I}(\mathbf{v}_{m+1})\big), \tag{12}$$

proportional to the jumps of the image between neighbouring voxels: largest for rough images, small for smooth ones. Evaluating each crossing directly from Eq. (2), as Siddon did, gives an error of a few roundoffs of α per crossing, independent of its position along the ray. The incremental update is a speed optimisation that trades this for one addition instead of a multiplication and a division.

4 A consistent traversal for GPUs

4.1 Consistent initialisation and direct crossings

The first voxel is the one the ray is in just after entering. On a plane, the direction of travel decides the side:

$$i_0 = \begin{cases} \lfloor e_x \rfloor & r_x \geq 0 \\ \lceil e_x \rceil - 1 & r_x < 0 \end{cases}, \qquad i_0 \leftarrow \min(\max(i_0, 0),\, n_x - 1), \tag{13}$$

and likewise for j_0, k_0. This replaces Eq. (9). It is the initialisation of Amanatides and Woo's traversal [3], written out for entry points on planes, where the direction of travel decides the side. The clamp only absorbs entry points that rounding places a hair outside the image. The next crossing along each axis is the face of the current voxel ahead of the ray, computed from its plane with Eq. (2), as in Siddon's original formulation [1]: $\alpha_x(i+1)$ for $r_x > 0$ and $\alpha_x(i)$ for $r_x < 0$. This replaces Eqs. (6)–(8) and the incremental update in Eq. (11). Deriving the crossings from the current voxel, rather than from separately computed plane limits, is also what several CPU implementations do (Section 6); it is what makes the first voxel and the crossings agree.

Lemma 1. *In exact arithmetic, the traversal defined by Eq.* (13)*, the crossings* $\alpha_x(i+1)$ *or* $\alpha_x(i)$ *of the current voxel, and the update of Eq.* (11) *without the increment* α_{xu}*, computes Eq.* (4) *exactly for every ray with* $\alpha_{min} < \alpha_{max}$*.*

Proof sketch. For every axis, $\mathbf{p}(\alpha)$ lies in the slab $[i_0, i_0 + 1)$ (resp. $(i_0, i_0 + 1]$ for $r_x < 0$) for α just above α_{min}, by construction of Eq. (13). The crossing computed for the current index is therefore the next plane the ray meets along that axis, and crossing it moves the ray into the slab of the updated index. By induction, the index always names the voxel the ray is in, and the crossings are visited in increasing order, so the segments are those of Eq. (4); segments of zero length (edges and corners) contribute nothing. □

The loop ends when the next crossing reaches α_{max}, the last segment ending exactly at α_{max}, so N_p and the plane limits of Eqs. (6)–(7) are no longer needed. In floating point, a step limit $n_x + n_y + n_z + 3$ guarantees termination even if rounding never lets a crossing reach α_{max}.

4.2 What is costly on a GPU

Siddon's and Jacobs' algorithms minimise arithmetic, the appropriate measure on the scalar processors for which they were designed. A GPU executes threads in warps of 32 that issue each instruction together [22], and its costs differ in four ways that matter here. None of them is specific to ray tracing; they are the standard considerations of GPU programming, applied to this loop.

A texture fetch takes tens to hundreds of cycles, and independent arithmetic issued while it is in flight is nearly free: the memory reads, not the arithmetic, set the pace. When the threads of a warp take different branches, the warp executes each branch in turn with the other threads masked. Neighbouring rays cross similar planes, but not at the same step: at a given iteration one ray crosses an x plane while its neighbour crosses a y plane. In the original kernel the texture fetch was inside each of the three branches of Eq. (11), so a warp could issue up to three fetches per step.

Integer–float conversions have a fraction of the throughput of a fused multiply–add. The texture is addressed with floating-point coordinates, so the original kernel converted three indices for every fetch, and it updated the index in floating point, $i = \mathrm{fl}(i + i_u)$, converting back. In TIGRE the indices were `unsigned long`, 64-bit on Linux, and 64-bit conversions proved far slower still (Section 5.4); the type of the indices is easy to overlook in any implementation.

Finally, an FMA costs the same as an addition. Evaluating Eq. (2) as $\mathrm{fl}(b\, r_x^{-1} + o_x)$, with $r_x^{-1} = 1/r_x$ and $o_x = -p_{x1}/r_x$ computed once per ray, costs one FMA, as much as the addition in Eq. (11). Computing crossings directly is therefore not more expensive than accumulating them.

4.3 Implementation

Two choices follow. The texture fetch is placed at the start of each step, identical for all threads, and only the update of the axis crossed is in the branch. And after the initialisation the voxel is kept as the coordinates of its centre, $c_x = i + \frac{1}{2}$, which is what the texture fetch takes: the loop needs no conversion. Centres are exact in single precision for fewer than 2^{23} voxels per axis. The face ahead of the voxel is at $c_x + h_x$, $h_x = s_x/2$, so the next crossing is

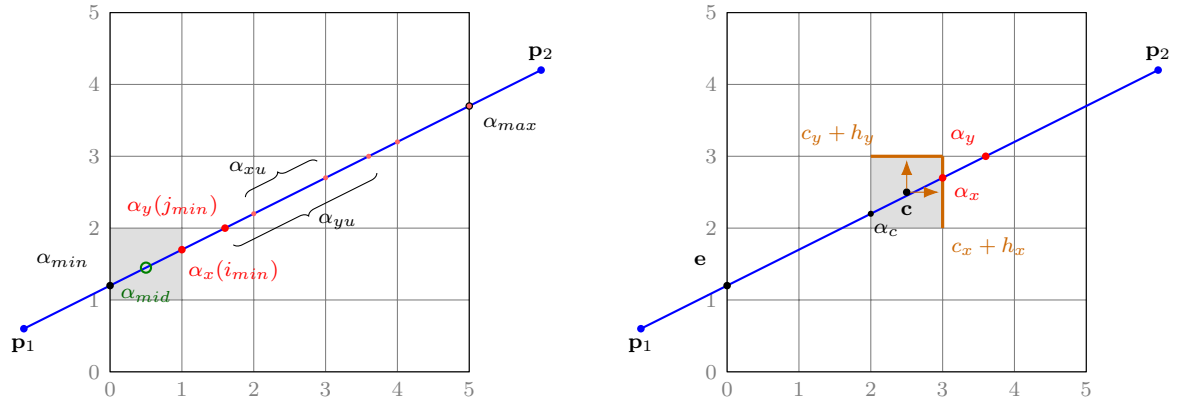
$$\alpha_x = \mathrm{fl}\left((c_x + h_x)\, r_x^{-1} + o_x\right). \tag{14}$$

Figure 2(b) shows these variables. Algorithm 1 sets the new traversal next to Jacobs', with the lines that differ highlighted, and Appendix A gives the CUDA code.

The two loops have the same shape. The differences are where the voxel is read, now once per step outside the branch; how the crossing is advanced, now evaluated from the plane instead of accumulated; how the voxel is held, now as a floating-point centre; and how the loop ends, now at α_{max} instead of after N_p steps. Reading the voxel before the branch requires the length of the segment before knowing which axis is crossed, hence α_n, and the last segment is clipped at α_{max}.

Rounding. Each crossing carries the error of one evaluation of Eq. (14): the rounding of the FMA, and those of r_x^{-1} and o_x. The latter two are common to all crossings of the ray, so they shift and scale them together by a relative $\sim 10^{-7}$, which is equivalent to an equally small change of geometry and does not affect how length is shared between voxels. No error grows with the number of steps.

Indices. The integer indices exist only in Eq. (13), as 32-bit signed integers: signed because $\lceil e_x \rceil - 1$ can be -1 before the clamp. (The original kernel converted such values to an unsigned type, which is undefined for negative values; on the GPU it produced the out-of-image first voxel of case (ii).)



(a) Jacobs. The first crossing on each axis (red) comes from the plane limits, Eqs. (6)–(8); the first voxel (grey) from the midpoint α_{mid} of the first segment, Eq. (9). Later crossings are found by adding α_{xu} or α_{yu}, Eq. (11).

(b) Revised. The first voxel comes from the entry point **e** and the direction of travel, Eq. (13). The voxel is kept as its centre **c**; each crossing is computed from the face ahead of it (orange), $\alpha_x = \mathrm{fl}\left((c_x + h_x)r_x^{-1} + o_x\right)$, Eq. (14).

Figure 2: Variables of the two traversals, in 2D, for the same ray. Adapted from the diagram of [7].

Parallel beam. The same traversal applies unchanged to parallel beam, where each ray has its own source point; an axis with $r_d = 0$ is never crossed. TIGRE's parallel-beam kernel had the same loop as its cone-beam one and received the same change.

4.4 Operation counts

Siddon designed his algorithm to minimise floating-point operations, and Jacobs' variant reduces them further. Table 1 counts the operations per crossing of the three formulations, as written above (not from machine code). Jacobs' step needs three arithmetic operations, the subtraction and the multiply–add of the accumulation and the increment of α_x, against about eight for Siddon's merge, midpoint and voxel index, plus three float-to-integer conversions. The revised step has the same arithmetic as Jacobs': the increment becomes an FMA, which costs the same, and the integer increment of the index becomes a floating-point one. What it adds are two or three comparisons: the minimum α_n, the clamp to α_{max} and the termination test, which replace Jacobs' integer step counter. In operation counts, then, the revised step is marginally more expensive than Jacobs' and about half the cost of Siddon's. On a GPU these counts do not decide the time (Section 4.2): the revised step is faster because it issues one memory read per warp and no conversions (Section 5.4).

Table 1: Operations per plane crossing, in the loop, excluding the per-ray setup. FMA counts as one operation. "GPU": the conversions needed to address a texture with integer indices, as in the original kernel.

	Siddon (merge, midpoint)	Jacobs	revised
floating-point arithmetic	≈ 8	3	4
comparisons / min	≈ 2	2–3	5–6
integer operations	—	1	—
float–integer conversions	3	0 (3 on GPU)	0
memory reads	1	1	1

Algorithm 1 Jacobs' traversal, Eqs. (6)–(11) (left), and the revised one (right), for one ray after α_{min} and α_{max} are computed. Only the x branch of the update is written out. Highlighted lines differ. $\alpha_d = \infty$ for an axis with $r_d = 0$.

$i_{min}, i_{max}, \dots$ ▷ Eqs. (6)–(7)
$\alpha_d \leftarrow \alpha_d(d_{min})$ or $\alpha_d(d_{max})$ ▷ Eq. (8)
$N_p \leftarrow \sum_d (d_{max} - d_{min} + 1)$
$(i, j, k) \leftarrow \lfloor \mathbf{p}(\alpha_{mid}) \rfloor$ ▷ Eq. (9)
$\alpha_c \leftarrow \alpha_{min}$, $d_{ray} \leftarrow 0$
for N_p steps **do**
  if $\alpha_x \le \min(\alpha_y, \alpha_z)$ **then**
    d_{ray} += $(\alpha_x - \alpha_c)\, \mathbb{I}(i, j, k)$
    $i \leftarrow i + i_u$; $\alpha_c \leftarrow \alpha_x$
    $\alpha_x \leftarrow \alpha_x + \alpha_{xu}$
  else if $\alpha_y \le \alpha_z$ **then** ...
  else ...
  end if
end for
return $\|\mathbf{r}\|_\Delta\, d_{ray}$

$(c_x, c_y, c_z) \leftarrow (i_0, j_0, k_0) + \frac{1}{2}$ ▷ Eq. (13)
$h_d \leftarrow s_d/2$, $r_d^{-1} \leftarrow 1/r_d$, $o_d \leftarrow -p_{d1} r_d^{-1}$
$\alpha_d \leftarrow \mathrm{fl}\left((c_d + h_d) r_d^{-1} + o_d\right)$ ▷ Eq. (14)

$\alpha_c \leftarrow \alpha_{min}$, $d_{ray} \leftarrow 0$
for at most $n_x + n_y + n_z + 3$ steps **do**
  $\alpha_n \leftarrow \min(\alpha_x, \alpha_y, \alpha_z)$
  d_{ray} += $\left(\min(\alpha_n, \alpha_{max}) - \alpha_c\right) \mathbb{I}(\mathbf{c})$
  if $\alpha_n \ge \alpha_{max}$ **break**
  if $\alpha_x = \alpha_n$ **then**
    $c_x \leftarrow c_x + s_x$; $\alpha_c \leftarrow \alpha_x$
    $\alpha_x \leftarrow \mathrm{fl}\left((c_x + h_x) r_x^{-1} + o_x\right)$
  else if $\alpha_y = \alpha_n$ **then** ...
  else ...
  end if
end for
return $\|\mathbf{r}\|_\Delta\, d_{ray}$

4.5 Three issues, one traversal

The original kernel had three independent problems: an inconsistent initialisation (Section 3.1), accumulated rounding errors (Section 3.2), and a slow loop (Section 4.2). Their causes are unrelated, and each could be fixed alone (Table 2). In particular the speed has nothing to do with the two flaws: moving the fetch out of the branches and using 32-bit indices would have given Jacobs' traversal, flaws included, most of the gain (Table 6).

Table 2: The three issues of the original kernel, their causes, and the change that addresses each. Times are from the ablation of Section 5.4, relative to the original kernel.

issue	cause	change	if done alone
consistency	first voxel from a midpoint, Eq. (9), crossings from plane limits, Eqs. (6)–(8)	first voxel from entry point and direction, Eq. (13); crossings from the current voxel (Amanatides and Woo)	no measurable cost (once per ray)
accuracy	crossings accumulated, α_x += α_{xu}	each crossing from its plane, Eq. (14) (Siddon)	10% slower with an integer index; free with float centres
speed	fetch inside the branches; integer–float conversions; 64-bit indices on Linux	one fetch before the branch; float voxel centres; 32-bit indices	0.55–0.74 of the time; 32-bit indices alone remove a 5–8× slowdown on Linux

The changes do interact, and this is why one traversal is the natural way to make them. Deriving the crossings from the current voxel is what makes the initialisation consistent, and it also removes the plane limits and N_p. Computing each crossing from its plane costs a multiply–add per step, but keeping the voxel as a floating-point centre, which the speed change introduces to remove conversions, makes that multiply–add take the place of Jacobs' addition, so the accuracy comes at no cost. Conversely, the fixes cost nothing only in this arrangement: computing the crossings from an integer index made the loop about 10% slower. The result is a single loop, Algorithm 1 (right), that is consistent, accurate and fast; but readers who only want the speed

of Section 5.4 can obtain most of it with the changes in the last row of Table 2.

4.6 Revision for TIGRE 3.3.1: several reads in flight

The loop above issues one texture fetch per step, identical for all threads of a warp. But each step also ends with a branch on the axis crossed, and the fetch of the next step cannot be issued before that branch is resolved and the loop comes round. A warp therefore has one read in flight at a time, and the latency of each read, tens to hundreds of cycles, is hidden only by other warps. The sum does not depend on the branch: the coordinates of the next voxel do, but only through comparisons and additions.

TIGRE 3.3.1 removes the branch and walks several steps per pass, four in what follows (the number is chosen per problem, as the end of this section explains). Each step selects instead of branching: with $b_x = (\alpha_x = \alpha_n)$, $b_y = \neg b_x \wedge (\alpha_y = \alpha_n)$ and $b_z = \neg b_x \wedge \neg b_y$ (the same tie-breaking as before), it updates $c_x \leftarrow b_x \,?\, c_x + s_x : c_x$ and $\alpha_x \leftarrow b_x \,?\, \mathrm{fl}((c_x + h_x) r_x^{-1} + o_x) : \alpha_x$, and likewise for y and z. The four steps are straight-line code, so the compiler issues their four fetches back to back and waits once. The loop ends after a pass, not after a step, so up to three steps run past α_{max}; a flag set when $\alpha_n \geq \alpha_{max}$ keeps them from adding to the sum. They still fetch a voxel, outside the ray's path and with no effect, which is cheaper than the branch it replaces. Every operation that reaches the sum is the same as before, in the same order, so the projections are bit-for-bit identical (Appendix A gives the code).

Table 3 shows the effect in TIGRE. For these problems, the kernel becomes 1.1–1.2 times faster in 3D. In 2D (fan beam) the gain is smaller, 3–13%: there the time is set by the number of steps itself, about 1.3 times the number of samples an interpolating (Joseph-type) projector takes along the same ray. The same release also prepares the constants of the next kernel launch on the host while the current kernels run, instead of after waiting for them; that change is outside the traversal, but it is included in the whole-call times of Table 3 for many angles.

Table 3: TIGRE 3.3.1 against 3.3.0's kernel (Sections 4–4.5): time of the whole forward projection relative to ASTRA's forward projector measured in the same run (below 1, TIGRE is faster), PyTorch tensors, RTX 4070, Windows, two alternating rounds. "Four steps": this section; "+ launches": with the host-side change as well.

problem	one step per pass	four steps	+ launches
cone 256^3, 360 angles	0.84–0.87	0.70–0.71	0.70
cone 512^3, 360 angles	0.59–0.60	0.51	0.50
parallel 256^3, 360 angles	0.97–0.98	0.86–0.87	0.84
parallel 512^3, 360 angles	0.83–0.84	0.75	0.74
fan 1024^2, 360 angles	1.13–1.14	1.05	1.02–1.12
fan 1024^2, 1440 angles	1.64–1.66	1.54	1.39–1.40
fan 2048^2, 360 angles	1.32–1.33	1.17	1.16–1.18

One more variant was tried and rejected: reading the image from a layered 2D texture (one layer per slice, a point fetch at an integer layer) instead of a 3D texture. In TIGRE's backprojectors the same change halves the cost of every fetch, because a 3D linear fetch is trilinear even when one weight is zero; but the projector uses point fetches, and with the layered texture it was 1.3–1.5 times *slower* for most problems, in 3D and in 2D (faster only with a detector much coarser than the image).

How many steps per pass. Four steps per pass are not faster everywhere. When neighbouring rays are several voxels apart where they cross the image, and the image is large, the three extra fetches in flight evict from the texture cache the voxels that each ray reads next: for a 1024^3 image and rays 4 voxels apart, Nsight Compute measures an L1 hit rate of 41% with four steps per pass against 57% with one, and the four-step kernel takes 1.5 times as long as 3.3.0's.

Table 4 measures one, two and four steps per pass over the ray spacing r (a detector pixel scaled to the rotation axis, in voxels) and the image side n, on both GPUs. For the usual detector ($r = 1$, a detector pixel per voxel) four steps are the fastest on the Quadro RTX 6000 at every size, by up to 20%, and within 3% of the fastest on the RTX 4070. With coarser detectors, two steps and then one are faster. TIGRE 3.3.1 (pull request #916) compiles the kernel for the three and chooses on the host: four steps when $r < 1.25$ or $rn < 800$, two when $rn < 2700$, one above (n is the side of the piece of the image in one texture, when it is split). Since the steps of a pass add to the sum in the same order whatever their number, the projections are bit-for-bit identical for any choice. Over a sweep of 39 problems (cone beam 256^3–1024^3, parallel beam 512^3 and 1024^3, r from 0.5 to 6), the choice is within 3% of the fastest of the three on the RTX 4070 and within 5% on the Quadro RTX 6000 for all but three problems (5–8%), and it is never slower than the 3.3.0 kernel on the RTX 4070. These thresholds come from two GPUs; others may favour slightly different ones, and we would welcome measurements on them (the 3.3.1 speed note [21] says how to run the suite).

Table 4: One, two and four steps per pass: time of the whole forward projection, 360 angles, PyTorch tensors, median of two alternating rounds; RTX 4070 relative to the 3.3.0 kernel (one step per pass with branches), Quadro RTX 6000 relative to one step per pass. n^3 image, r: distance between neighbouring rays at the rotation axis, in voxels; the detector just covers the image. In bold, the choice of TIGRE 3.3.1.

				RTX 4070			Quadro RTX 6000		
beam	n	r	rn	one	two	four	one	two	four
cone	256	1	256	0.86	0.82	**0.81**	1.00	0.74	**0.63**
cone	512	1	512	0.89	0.86	**0.84**	1.00	0.71	**0.59**
cone	512	1.5	768	0.91	0.83	**0.83**	1.00	0.70	**0.61**
cone	512	4	2048	0.98	**0.95**	1.02	1.00	**0.98**	1.65
cone	1024	1	1024	0.89	0.87	**0.89**	1.00	0.70	**0.58**
cone	1024	2	2048	0.98	**0.91**	1.06	1.00	**0.85**	1.11
cone	1024	4	4096	**1.00**	1.15	1.49	**1.00**	0.98	1.16
cone	1024	6	6144	**1.00**	1.16	1.35	**1.00**	1.00	1.08
parallel	1024	1	1024	0.92	0.90	**0.92**	1.00	0.76	**0.66**
parallel	1024	4	4096	**1.01**	1.13	1.21	**1.00**	1.11	1.38

5 Experiments

The experiments answer three questions, one for each claim of the previous sections. First, correctness: does the original kernel fail on the rays predicted by Proposition 1, often enough to matter, and does the new kernel compute those rays correctly? Second, accuracy: how large is the rounding error of the incremental update described in Section 3.2, and how much does the direct evaluation of Eq. (14) reduce it? Third, speed: does the new traversal cost anything, and if it is faster, which of its changes make it so?

The experiments were made with TIGRE 3.3.0's kernel. TIGRE 3.3.1's (Section 4.6) computes the same sums in the same order, and its projections are bit-for-bit identical to 3.3.0's, so the results on correctness and accuracy hold for it unchanged; Section 4.6 gives its speed, and Section 5.5 its comparison with ASTRA.

5.1 Setup

All experiments use TIGRE, before and after the changes, as the implementation. Two systems were used: Windows with an NVIDIA RTX 4070 (Ada) and CUDA 12.3, and Linux with a Quadro RTX 6000 (Turing) and CUDA 12.5. Geometries are TIGRE's default cone-beam

geometry (source–origin 1000 mm, source–detector 1536 mm, image 256 mm wide, detector pixels of 0.8 mm) at n^3 voxels with an $n \times n$ detector. Accuracy was measured with a CPU emulation of the kernels in single precision, which reproduces the GPU outputs exactly, against the same traversal evaluated in double precision. Timings compare builds of TIGRE before and after the change in separate processes, in alternating order over several rounds (after a warm-up on Linux), with image and projections as GPU arrays; kernel time is taken from the PyTorch profiler and the whole call is synchronised. We report the mean ratio over rounds with a 95% confidence interval.

5.2 Correctness

We first compared the original and the new kernel on the GPU for a whole 256^3 image and 360 angles. Of 23.6 million rays, 424 differ by more than 0.1%, the worst by 14%. To decide which kernel is right we evaluated the four worst rays in double precision: in every case the double-precision value agrees with the new kernel (for example, original 127.07, new 145.38, exact 145.38), and the single-precision emulation of the original kernel reproduces its wrong value, so the error is in the traversal and not in the hardware. When the same image is split into slabs, as TIGRE does when memory is short, errors grow to about 20% on some rays, as expected from the larger number of rays that enter a slab on a plane.

Tracing one of these rays shows case (ii) of Proposition 1 at work. For a 256^3 image in 3 slabs of 86 slices, at angle 0, the ray of pixel $(v, u) = (36, 158)$ enters the first slab through its top face at $\mathbf{e} = (246.688,\ 142.000000,\ 86.000000)$, exactly on the plane $y = 142$. This is the second mechanism of Section 3.1: at angle 0 the face $z = 86$ is 42 voxels from the plane of the source, the pixel is 30.5 and -91.5 pixels from the detector centre, and $e_y = 128 + 42 \times 30.5/91.5 = 142$. The first y crossing is at $\alpha_{min} + 1.2 \times 10^{-7}$, so the midpoint of Eq. (9) is the entry point and gives $k = 86$, outside the slab, while the z crossings of Eq. (8) start at $z = 85$. The slab contributes 104.80 instead of 122.86, 15% less.

These comparisons look at individual rays; TIGRE's test suite checks the projector as a whole. On both systems, split and unsplit projections of a 512^3 random image agree to 10^{-4} of the maximum, a test the original kernel fails. The projector tests that check known line integrals, linearity and the agreement between the ray–voxel and interpolated projectors pass, and projections computed from GPU arrays are identical to those computed from host arrays. These tests cover parallel beam too, and pass with the new parallel-beam kernel (Windows).

5.3 Accuracy

To isolate the rounding error from the initialisation, we compared each traversal in single precision against itself in double precision, on 150 random rays through a 512^3 image. We used two images, chosen to test the prediction of Section 3.2 that the error of the incremental update scales with the jumps of the image between neighbouring voxels: a rough one, with voxel values uniform in $[0, 1)$, and a smooth one, a sum of two Gaussians.

Table 5: Relative error of single-precision line integrals against double precision, 150 random rays through a 512^3 image.

image	traversal	median	95th percentile	max
rough	incremental, Eq. (11)	1.1×10^{-4}	6.3×10^{-4}	1.7×10^{-3}
	direct, Eq. (14)	2.5×10^{-6}	8.1×10^{-6}	1.1×10^{-5}
smooth	incremental	3.2×10^{-5}	1.3×10^{-4}	2.1×10^{-4}
	direct	4.4×10^{-7}	1.5×10^{-6}	2.6×10^{-6}

Table 5 confirms both predictions. The incremental update loses three to four significant

digits, up to 1.7×10^{-3} on the rough image, and about five times less on the smooth one. The direct evaluation is 40–150 times more accurate on both, with errors of 10^{-5} or less, close to what single precision allows for a sum of a few hundred terms. These errors are below the noise of measured data, which is why they had gone unnoticed (Section 6); they matter when projections are compared with each other, as in the split-versus-unsplit test, and in algorithms that rely on the operator being reproducible.

5.4 Speed

The new kernel changes several things at once, and comparing it with the original only tells whether it is faster, not why. We therefore first built an ablation, in which each variant changes one thing, and then measured the whole change in TIGRE, which is what users see.

Ablation. The kernel was copied into a standalone CUDA program with variants that each change one aspect of the loop, run interleaved in one process with TIGRE's texture settings, launch configuration and geometry. Since a variant can appear faster by doing less work, each variant's output is compared with the new kernel's (Table 6).

Table 6: Ablation: kernel time relative to the original kernel, standalone variants, RTX 4070, Windows, 360 angles, median of 15–30 rounds. "Max diff": largest difference from the new kernel's output, relative to its maximum. On Windows `long` is 32-bit, so the original indices are 32-bit there; the 64-bit rows correspond to the original kernel on Linux.

variant	256^3	512^3	max diff
original	1.00	1.00	8–9×10^{-2}
original, index step as an integer add	1.00	0.99	8–9×10^{-2}
original, 64-bit indices	8.09	7.60	8–9×10^{-2}
one convergent fetch; incremental crossings	0.71	0.74	$\leq 2 \times 10^{-3}$
one convergent fetch; direct crossings from an integer index	0.80	0.81	0
new: one convergent fetch; direct crossings from float centres	**0.55**	**0.67**	0
new, with 64-bit indices	5.29	4.69	0

Moving the fetch out of the branches gives most of the gain, about 30%, and keeping the voxel as floating-point centres a further 10–15%, by removing the conversions that computing the crossing from an integer index would add. Doing the original index step as an integer addition changes nothing, and 64-bit indices make any variant 5–8 times slower on this GPU. Appendix B relates these numbers to the machine code.

In TIGRE. The standalone program leaves out everything around the kernel, so we also compared complete builds of TIGRE, before and after the change, measuring both the kernel time and the whole projection call, which includes memory transfers and setup. Because GPU clocks vary with temperature and with other work, the two builds were run alternately, each in a fresh process, over several rounds, and we report the ratio with its confidence interval (Table 7).

For cone beam, the new kernel takes 54–69% of the original's time on Windows and 15–18% on Linux (at 512^3, 2054 ms of kernel time become 364 ms), consistent with the 64-bit rows of the ablation. It was faster in every round for every size. The gain is smaller for larger images, plausibly because more texture reads miss the cache and the kernel becomes more memory-bound; we did not verify this. For parallel beam the new kernel takes 51–59% of the original's time on Windows and 21–23% on Linux (at 512^3, 1569 ms of kernel time become 355 ms); on Windows its gain does not shrink with image size, and on Linux only slightly. TIGRE's original parallel-beam kernel had extra corrections in its initialisation.

Table 7: New against original kernel in TIGRE: time ratio, mean with 95% confidence interval over 4–6 rounds. Windows: RTX 4070; Linux: Quadro RTX 6000.

		cone beam		parallel beam	
image, angles		Windows	Linux	Windows	Linux
128^3, 360	kernel	0.54 ± 0.02	0.152 ± 0.001	0.593 ± 0.005	0.209 ± 0.001
	whole call	0.76 ± 0.06	0.24 ± 0.01	0.76 ± 0.06	0.32 ± 0.04
256^3, 360	kernel	0.56 ± 0.02	0.158 ± 0.001	0.530 ± 0.008	0.220 ± 0.001
	whole call	0.64 ± 0.02	0.175 ± 0.001	0.58 ± 0.04	0.237 ± 0.001
512^3, 360	kernel	0.67 ± 0.01	0.177 ± 0.000	0.543 ± 0.010	0.227 ± 0.002
	whole call	0.69 ± 0.01	0.182 ± 0.001	0.57 ± 0.03	0.231 ± 0.002
768^3, 180	kernel	0.69 ± 0.01	—	0.506 ± 0.012	—
	whole call	0.74 ± 0.01	—	0.55 ± 0.01	—

5.5 Comparison with ASTRA

The comparisons above are between two versions of TIGRE. To place the result among GPU projectors in common use, we also timed TIGRE's Siddon projector against the forward projector of the ASTRA Toolbox [13] (version 2.5.0) on the same problems. ASTRA's cone-beam GPU forward projector is ray-driven with Joseph-type interpolation [12], not an exact intersection-length method, so this compares projectors used for the same purpose rather than two implementations of one algorithm.

Both toolboxes ran in one process, on the cone-beam geometry of the previous sections at n^3 voxels with 360 angles, on one GPU and, on the Linux system, on two. Inputs and outputs were host arrays, so the time includes the copies to and from the GPUs, which is how most users call either toolbox; on one GPU we also timed PyTorch GPU tensors in and out, with no host copies. ASTRA's angle and detector conventions were matched to TIGRE's on a small problem, by trying the eight angle conventions and four detector flips and keeping the one under which the projections of an asymmetric image agree, and at every size the outputs of the two toolboxes were compared, to check that both solved the same problem. Each round ran every operation of both toolboxes, alternating which went first; we report the median time and the ratio of the TIGRE time to the ASTRA time, as a mean over rounds with a 95% confidence interval. The script, `tigre_vs_astra.py`, is next to this note; it also times TIGRE's interpolated projector, backprojection and FDK reconstruction, which are not the subject of this note.

Table 8: TIGRE against ASTRA, NVIDIA GeForce RTX 4070 (Windows): forward projection, TIGRE's Siddon projector against ASTRA's, 360 angles. Time of the whole call, median over 4 rounds. Ratio: TIGRE time over ASTRA time, mean and 95% confidence interval over rounds (below 1: TIGRE faster). NumPy: arrays in host memory, copies to and from the GPUs included; tensors: PyTorch GPU tensors in and out, no copies.

		1 GPU		
image	arrays	TIGRE ms	ASTRA ms	ratio
256^3	NumPy	37	55	0.68 ± 0.05
	tensors	34	37	0.92 ± 0.03
512^3	NumPy	267	507	0.53 ± 0.01
	tensors	238	406	0.58 ± 0.01

Tables 8 and 9 give the results, for TIGRE 3.3.0. On one GPU, TIGRE's exact projector is faster than ASTRA's interpolating one in every case but one: it takes 0.53–0.92 of ASTRA's time on the RTX 4070, and 0.55–0.89 on the Quadro RTX 6000; the exception is GPU tensors at 512^3 on the Quadro RTX 6000, where the two are equal within the confidence interval (1.07 ± 0.10). The advantage grows with the image: at 1024^3, TIGRE takes 0.55 of ASTRA's time. ASTRA

Table 9: TIGRE against ASTRA, Quadro RTX 6000 (Linux): forward projection, TIGRE's Siddon projector against ASTRA's, 360 angles. Time of the whole call, median over 3 rounds. Ratio: TIGRE time over ASTRA time, mean and 95% confidence interval over rounds (below 1: TIGRE faster). NumPy: arrays in host memory, copies to and from the GPUs included; tensors: PyTorch GPU tensors in and out, no copies.

		1 GPU			2 GPUs		
image	arrays	TIGRE ms	ASTRA ms	ratio	TIGRE ms	ASTRA ms	ratio
512^3	NumPy	417	477	0.89 ± 0.07	325	300	1.10 ± 0.24
	tensors	379	359	1.07 ± 0.10	—	—	—
1024^3	NumPy	1974	3607	0.55 ± 0.00	1654	2128	0.77 ± 0.04
	tensors	1696	3023	0.56 ± 0.01	—	—	—

gains more from a second GPU, whose time drops by a factor of 1.6–1.7 against 1.2–1.3 for TIGRE, so that on two GPUs the two are on par at 512^3 (1.10 ± 0.24) and TIGRE is faster at 1024^3 (0.77 ± 0.04); we did not investigate this difference. The outputs of the two toolboxes agreed to within 2.2×10^{-3} (relative L_2 difference after a least-squares scale), the difference between the two projection models.

Table 10 repeats the one-GPU comparison with TIGRE 3.3.1, on the same geometry (detector pixels of 0.8 mm), as part of a wider comparison of the two toolboxes made for that release [21]. TIGRE 3.3.1's projector takes 0.32–0.77 of ASTRA's time on one GPU. Where the two releases were measured on the same problem, 3.3.1 is further ahead: with host arrays, 0.55 of ASTRA's time at 256^3 on the RTX 4070 (0.68 with 3.3.0) and 0.47 at 1024^3 on the Quadro RTX 6000 (0.55 with 3.3.0). For problems that fit in one GPU, ASTRA still gains more from more GPUs: on the Linux system at 1024^3, with a detector that just covers the image, TIGRE 3.3.1 takes 1.44 s on two GPUs and ASTRA 2.17 s, but on four 2.08 and 1.68 s [21].

Table 10: TIGRE 3.3.1 against ASTRA, one GPU: forward projection, TIGRE's Siddon projector against ASTRA's, 360 angles, detector pixels of 0.8 mm. Time of the whole call in ms, median over three rounds; ratio as in Table 8. *At 1024^3 with GPU tensors, the image and the projections nearly fill the RTX 4070's 12 GB, and ASTRA's call slows down.

		RTX 4070 (Windows)			Quadro RTX 6000 (Linux)		
image	arrays	TIGRE ms	ASTRA ms	ratio	TIGRE ms	ASTRA ms	ratio
256^3	NumPy	31.5	56.8	0.55 ± 0.05	51.2	66.6	0.77 ± 0.02
	tensors	27.4	37.3	0.73 ± 0.04	25.1	44.1	0.57 ± 0.00
1024^3	NumPy	1353	4276	0.32 ± 0.02	1679	3604	0.47 ± 0.00
	tensors	1013	17949*	0.06 ± 0.00	1356	2950	0.46 ± 0.00

6 Discussion

Why these problems went unnoticed. The inconsistent initialisation affects, in any geometry, a fraction of rays of the order of 10^{-5} (one in 50 000 in our example), each wrong by an amount comparable to the difference between neighbouring voxels, and scattered across projections: invisible in a reconstruction. The drift affects every ray, but by 10^{-4}–10^{-3}, below the noise of measured data. Neither changes a reconstruction in practice; what they break is consistency. Both became visible only when projections of the same image computed in slabs and in one piece were compared, which changes where the traversal starts and so where the accumulation restarts.

Generality. Proposition 1 concerns the incremental algorithm as published, not TIGRE's code: any implementation that initialises with a single midpoint, Eq. (9), and then only increments shares it, on CPU or GPU, in single or double precision. The drift affects any implementation that accumulates crossings in single precision. Double precision is not a good alternative on GPUs, where it runs at a small fraction of the single-precision rate, and the direct evaluation already achieves an error of about 10^{-5}. The arrangement around one convergent fetch is specific to SIMT hardware.

Related implementations. Appendix C summarises how the open-source implementations we examined deal with these issues. Each ingredient of Section 4 exists elsewhere: the consistent initialisation in Amanatides and Woo and in several CPU codes, the direct evaluation of crossings in Siddon's method. To the best of our knowledge, none of the implementations we examined combines a consistent initialisation, crossings computed without accumulation at the cost of Jacobs' traversal, and a loop arranged for SIMT execution. We did not survey implementations systematically, and compared speed only with ASTRA (Section 5.5).

Rays inside a plane. A ray that lies in a plane between voxels ($r_b = 0$, $e_b \in \mathbb{Z}$) has an ambiguous line integral. STIR traces it twice, half a voxel to each side, and averages. Our traversal, like Siddon's midpoint rule, takes the voxel on the positive side. This is a modelling choice rather than an error, but a symmetric treatment may be preferable for geometries that produce many such rays.

Limitations. Speed was measured on two GPUs of two generations, accuracy by emulation on 150 rays per image, and correctness in exact arithmetic on a small grid. We did not study backprojection: TIGRE's backprojectors are voxel-driven and do not use this traversal, but a ray-driven matched backprojector built on Jacobs' traversal would inherit both flaws.

7 Conclusion

The incremental form of Siddon's algorithm, as published by Jacobs et al. and commonly implemented, has two flaws: an initialisation that is inconsistent for an exactly characterised set of rays, and, in single precision, rounding errors that accumulate along the ray. Both have known remedies, the initialisation of Amanatides and Woo and the direct evaluation of crossings of Siddon's original method. This note shows how to combine them on a GPU without extra cost, by organising the step around one convergent memory read, with floating-point voxel centres and 32-bit indices. In TIGRE the revised projector is consistent for every ray, about 100 times more accurate, and 1.4–6.5 times faster, depending on the platform; most of the speed comes from the GPU arrangement, not from the algorithm. On one GPU it is also faster than the interpolating forward projector of the ASTRA Toolbox, by up to a factor of 1.9 at the sizes tested with TIGRE 3.3.0, and of 3.2 with TIGRE 3.3.1, whose kernel walks several steps of the ray per pass.

Code availability

The implementation is part of TIGRE, `https://github.com/CERN/TIGRE`, from pull request #865, revised in #913 and #916 for TIGRE 3.3.1. The cone-beam kernel is in `Common/CUDA/Siddon_projection.cu` and the parallel-beam kernel in `Common/CUDA/Siddon_projection_parallel.cu`. The comparison with ASTRA is `docs/technical-notes/siddon/tigre_vs_astra.py`.

Use of generative AI

This work and this note were done with substantial help from Claude (Anthropic), a large language model, used as a coding and writing assistant (Claude Opus, through Claude Code). Under the author's direction, Claude carried out much of the investigation and analysis, wrote the revised kernels and the host-side code, wrote the floating-point emulations and the benchmark and test scripts, searched and read the literature and other implementations, and wrote the first draft of this note, including the proofs and figures.

The author conceived and directed the work, decided what to change and how to present it, ran every test and benchmark on GPU hardware (the results reported here), checked the derivations and the results, and revised the text. The author takes full responsibility for the content of this note, including any errors. Claude is not an author: it cannot take responsibility for the work or for its publication.

Acknowledgements

The author thanks Claude (Anthropic) for its assistance, as described above.

References

[1] R. L. Siddon, "Fast calculation of the exact radiological path for a three-dimensional CT array," *Medical Physics* 12(2):252–255, 1985.

[2] F. Jacobs, E. Sundermann, B. De Sutter, M. Christiaens, I. Lemahieu, "A fast algorithm to calculate the exact radiological path through a pixel or voxel space," *Journal of Computing and Information Technology* 6(1):89–94, 1998.

[3] J. Amanatides, A. Woo, "A fast voxel traversal algorithm for ray tracing," *Eurographics '87*, 1987.

[4] A. Biguri, M. Dosanjh, S. Hancock, M. Soleimani, "TIGRE: a MATLAB-GPU toolbox for CBCT image reconstruction," *Biomedical Physics & Engineering Express* 2(5):055010, 2016. doi:10.1088/2057-1976/2/5/055010.

[5] A. Biguri, R. J. Lindroos, R. Bryll, H. Towsyfyan, H. Deyhle, I. Harrane, R. Boardman, M. Mavrogordato, M. Dosanjh, S. Hancock, T. Blumensath, "Arbitrarily large tomography with iterative algorithms on multiple GPUs using the TIGRE toolbox," *Journal of Parallel and Distributed Computing* 146:52–63, 2020. doi:10.1016/j.jpdc.2020.07.004.

[6] A. Biguri, T. Sadakane, R. Lindroos, Y. Liu, M. Sabaté Landman, Y. Du, M. Lohvithee, S. Kaser, S. Hatamikia, R. Bryll, E. Valat, S. Wonglee, T. Blumensath, C.-B. Schönlieb, "TIGRE v3: efficient and easy to use iterative computed tomographic reconstruction toolbox for real datasets," *Engineering Research Express* 7(1):015011, 2025. doi:10.1088/2631-8695/adbb3a.

[7] A. Biguri, *Iterative Reconstruction and Motion Compensation in Computed Tomography on GPUs*, PhD thesis, University of Bath, 2017. `https://github.com/AnderBiguri/PhDThesis`.

[8] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, 2002.

[9] Insight Toolkit (ITK), `SiddonJacobsRayCastInterpolateImageFunction`, module TwoProjectionRegistration (originally J. Wu, *Insight Journal*, publication 784), now in `Modules/Registration/TwoProjectionRegistration`, `https://github.com/InsightSoftwareConsortium/ITK`.

[10] G. Fattori, `itkgSiddonJacobsRayCastInterpolateImageFunction`, reg23Topograms, Paul Scherrer Institut, `https://gitea.psi.ch/fattori_g/reg23Topograms`.

[11] pyRadPlan, `pyRadPlan.raytracer._siddon`, `https://pyradplan.readthedocs.io`.

[12] P. M. Joseph, "An improved algorithm for reprojecting rays through pixel images," *IEEE Transactions on Medical Imaging* 1(3):192–196, 1982.

[13] W. van Aarle, W. J. Palenstijn, J. Cant, E. Janssens, F. Bleichrodt, A. Dabravolski, J. De Beenhouwer, K. J. Batenburg, J. Sijbers, "Fast and flexible X-ray tomography using the ASTRA toolbox," *Optics Express* 24(22):25129–25147, 2016.

[14] S. Rit, M. Vila Oliva, S. Brousmiche, R. Labarbe, D. Sarrut, G. C. Sharp, "The Reconstruction Toolkit (RTK), an open-source cone-beam CT reconstruction toolkit based on the Insight Toolkit (ITK)," *Journal of Physics: Conference Series* 489:012079, 2014.

[15] parallelproj, `https://github.com/gschramm/parallelproj`.

[16] D. Gürsoy, F. De Carlo, X. Xiao, C. Jacobsen, "TomoPy: a framework for the analysis of synchrotron tomographic data," *Journal of Synchrotron Radiation* 21(5):1188–1193, 2014.

[17] K. Thielemans, C. Tsoumpas, S. Mustafovic, T. Beisel, P. Aguiar, N. Dikaios, M. W. Jacobson, "STIR: software for tomographic image reconstruction release 2," *Physics in Medicine and Biology* 57(4):867–883, 2012.

[18] TomograPy, `https://github.com/nbarbey/TomograPy`.

[19] plastimatch, `https://gitlab.com/plastimatch/plastimatch`.

[20] LEAP, LivermorE AI Projector for computed tomography, `https://github.com/llnl/LEAP`.

[21] A. Biguri, "TIGRE 3.3.1 and ASTRA 2.5: a speed comparison of the projection operators on one and several GPUs," TIGRE technical note, November 2026.

[22] NVIDIA, *CUDA C++ Programming Guide*, sections on SIMT architecture and arithmetic instruction throughput.

A CUDA code

Cone-beam kernel, `Common/CUDA/Siddon_projection.cu`, after α_{min} (`am`) and α_{max} (`aM`) are computed:

```
int i=(ray.x<0.0f)? (int)ceilf(pX)-1 : (int)floorf(pX);      // Eq. (start); j, k alike
i=(i<0)? 0 : ((i>(int)geo.nVoxelX-1)? (int)geo.nVoxelX-1 : i);
float cX=(float)i+0.5f;                                       // voxel centre
float sX=(ray.x<0.0f)? -1.0f : 1.0f;  float hX=0.5f*sX;      // direction, face offset
float rX=(ray.x!=0.0f)? 1.0f/ray.x : 0.0f;  float oX=-source.x*rX;
float ax=(ray.x!=0.0f)? fmaf(cX+hX,rX,oX) : INFINITY;        // next x crossing
// ... same for y, z ...
float ac=am, sum=0.0f;
const int maxsteps=(int)geo.nVoxelX+(int)geo.nVoxelY+(int)geo.nVoxelZ+3;
for (int step=0; step<maxsteps; step++){
    float aminc=fminf(fminf(ax,ay),az);
    sum+=(fminf(aminc,aM)-ac)*tex3D<float>(tex, cX, cY, cZ);  // one fetch, all threads
    if (aminc>=aM) break;
    if (ax==aminc){      cX+=sX; ac=ax; ax=fmaf(cX+hX,rX,oX); }
    else if(ay==aminc){  cY+=sY; ac=ay; ay=fmaf(cY+hY,rY,oY); }
    else{                cZ+=sZ; ac=az; az=fmaf(cZ+hZ,rZ,oZ); }
```

```
}
detector[idx]=sum*maxlength;
```

TIGRE 3.3.1 (Section 4.6), the same set-up, then `STEPS` branch-free steps per pass (a template parameter: 1, 2 or 4, chosen on the host):

```
bool out=false;
for (int step=0; step<maxsteps; step+=STEPS){
#pragma unroll
    for (int s=0; s<STEPS; s++){
        const float aminc=fminf(fminf(ax,ay),az);
        const float t=tex3D<float>(tex, cX, cY, cZ);       // STEPS fetches in flight
        if (!out) sum+=(fminf(aminc,aM)-ac)*t;             // nothing added after the exit
        out=out || (aminc>=aM);
        const bool bx=(ax==aminc), by=!bx && (ay==aminc), bz=!bx && !by;
        ac=aminc;
        cX=bx? cX+sX : cX;  ax=bx? fmaf(cX+hX,rX,oX) : ax;
        cY=by? cY+sY : cY;  ay=by? fmaf(cY+hY,rY,oY) : ay;
        cZ=bz? cZ+sZ : cZ;  az=bz? fmaf(cZ+hZ,rZ,oZ) : az;
    }
    if (out) break;
}
detector[idx]=sum*maxlength;
```

B Machine code of the ablation variants

The machine code (sm_89) explains the ablation of Table 6. The original loop, unrolled four times by the compiler, has 220 instructions for four steps, 12 of them texture fetches (one in each branch); the restructured loop has 43 instructions per step and one fetch. Computing the crossing from an integer index costs 49 instructions per step, six of them conversions, which keeping the voxel as floating-point centres removes. Doing the original index step as an integer addition changes nothing because the conversions it saves are not on the critical path while the fetch is divergent. The slowdown with 64-bit indices is most likely due to 64-bit integer–float conversions taking a slow path; we did not isolate this further.

C Related implementations

We read the ray–voxel code of several open-source packages for tomography and registration. Many use other projection models, such as Joseph's interpolation [12] (RTK [14], parallelproj [15], and ASTRA's 3D GPU projectors [13]), and are not concerned here. Of those that compute exact intersection lengths, TomoPy [16] and vectorised codes such as pyRadPlan [11] implement Siddon's original method: every crossing is computed directly and each segment gets its own midpoint, so they are immune to Proposition 1 and have no drift, at about twice the arithmetic per crossing of the incremental form and with per-ray storage for the merge. STIR [17], TomograPy [18] and plastimatch [19] use the incremental traversal but derive the crossings from the faces of the first voxel, as Amanatides and Woo do, so their initialisation is consistent; they then accumulate the crossings, or the distances to them, and so have the drift of Section 3.2 (in single precision for STIR and TomograPy), and some add tolerances or explicit branches for ties. LEAP [20] avoids a traversal altogether: it steps through the image slice by slice along the dominant axis and computes the exact lengths in the at most four voxels each slice can contain, with more arithmetic and memory reads per voxel but no data-dependent branching; its positions are also accumulated. The Siddon–Jacobs ray caster of ITK's TwoProjectionRegistration module [9] takes the first voxel as the floor of the entry point whatever the direction of travel, so rays that enter on a plane take the zero-length step of case (i), and it advances the voxel index before reading the voxel, so each segment is credited to the next voxel

along the ray and the first segment is lost; codes derived from it add special cases and tolerances for rays aligned with the grid [10].