Adaptive AIE–PL Systems for Efficient End-to-End Pyramidal 3D Image Registration

Giuseppe Sorrentino†, Paolo S. Galfano†, Claudio Di Salvo†, Eleonora D'Arnese⋆, Davide Conficconi†
†*Politecnico di Milano,* ⋆*University of Edinburgh*
{giuseppe.sorrentino,10927428,claudio.disalvo,davide.conficconi}@polimi.it, eleonora.darnese@ed.ac.uk

***Abstract*—Modern accelerators maximize throughput through aggressive specialization. However, in many real-world applications, workloads often vary at runtime, requiring multiple bitstreams to handle such changes. As a result, frequent reconfigurations introduce substantial overhead that can dominate end-to-end execution time. This issue is particularly evident in AIE–PL systems, where statically scheduled AI Engines (AIEs) achieve high performance through compile-time optimization and are therefore typically tailored to fixed workloads. Although AIEs support Runtime Parameters (RTPs) under Processing System (PS) orchestration, RTPs are impractical for discrete hosts. For this reason, we present a structured approach to designing single-bitstream, runtime-adaptable AIE–PL accelerators that does not rely on RTPs, suitable for discrete hosts. We exploit the Programmable Logic (PL) to generate and stream a compact metadata packet that distributes workload configuration across a directed AIE graph before computation. By doing so, we deliberately trade a fraction of fixed-instance efficiency for flexibility. We validate our approach by devising *PeterPan*, a software-programmable AIE–PL accelerator for 3D image registration. *PeterPan* supports runtime-varying problem sizes and integrates seamlessly into multi-stage pipelines, such as pyramidal (coarse-to-fine) registration. To maximize *PeterPan* utilization, we couple it with an ad-hoc software module that employs a novel heuristic to rapidly select informative sub-volumes, keeping the accelerator continuously fed and preventing input-side stalls. On a VCK5000, *PeterPan* matches state-of-the-art accelerator performance while retaining software programmability. In the end-to-end task, instead, *PeterPan* delivers a 3.06× speedup and a 2.74× higher energy efficiency than the state-of-the-art AIE-PL accelerator.**

***Index Terms*—hardware acceleration, AI Engine, FPGA, image registration, heterogeneous systems**

I. Introduction

The pursuit of performance in modern accelerators often leads to highly specialized, high-frequency designs that maximize throughput at the expense of flexibility [1]–[4]. While effective for fixed, well-characterized workloads, this specialization becomes problematic in pipelines that must adapt at runtime, as multiple bitstreams are required to handle different cases, introducing non-negligible reconfiguration overhead. When this happens, reconfiguration cannot be considered a *one-time cost* as generally assumed [3], [5]–[9].

Specialization is particularly relevant for AI Engines (AIEs), statically-scheduled Very Long Instruction Word (VLIW) processors with vector unit capabilities, which benefit from static-scheduling optimizations to enhance performance at the cost of fixing the workload shape [6], [10]–[13]. Aware of such limitations, our ***first contribution*** is **a strategy to design runtime-adaptable accelerators** featuring both AIE and Programmable Logic (PL), as in modern Versal accelerators (1) (§III-A). By following a structured approach, we trade a small fraction of static-scheduling performance for runtime flexibility.

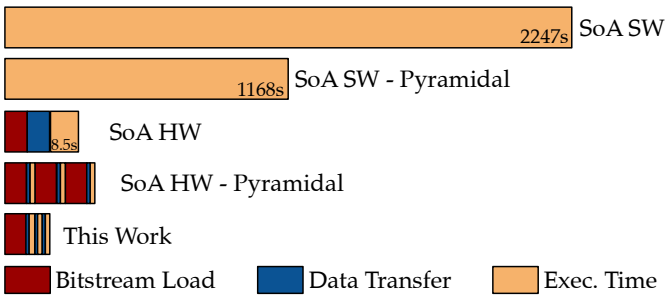


Fig. 1. Latency comparison for both software and hardware implementation of the literature solution, compared to the respective pyramidal versions. Notably, the hardware does not benefit from the pyramidal as the software version does.

Image registration is a representative case where runtime flexibility is essential. The goal of such a procedure is to align a floating 3D image with a reference, correcting acquisition distortions caused by differences in modality or time instants [14]–[17]. It is widely used in multiple scenarios, such as robotic surgery [18], atlas construction [19], and augmented reality [20]. Image registration pipelines typically combine heuristic pre-processing with deep-learning-based non-rigid alignment. However, the workload is highly unbalanced, as pre-processing can be orders of magnitude slower [21]. Accordingly, state-of-the-art AIE–PL accelerators focus on pre-processing, outperforming FPGA and GPU solutions [6], [12]. However, these approaches are unsuitable for modern applications requiring flexible, multi-stage workflows. In particular, modern pipelines adopt pyramidal, coarse-to-fine schemes [22], [23], in which the alignment starts at low resolution and is progressively refined at higher resolutions. As shown in the first two bars of Figure 1, a pyramidal strategy alone nearly doubles the performance of the state-of-the-art software implementation. In contrast, applying the same strategy to the hardware-accelerated pipeline (third and fourth bars in Figure 1) is unsatisfactory: the multi-level scheme is even slower than single-resolution registration because each pyramid level requires a different bitstream, and the resulting reconfiguration overhead dominates.

In such scenarios, the only alternatives are to improve the reconfiguration mechanism to reduce overhead or to ensure that a single bitstream supports multiple workloads. Thus, our

second contribution is ***PeterPan*****,** a novel runtime-adaptable AIE–PL accelerator for image registration ❷(§III-B). Shaped according to ❶, *PeterPan* delivers comparable fixed-instance performance to state-of-the-art AIE-PL solutions, and ensures higher flexibility – thus suiting the integration into pyramidal schemes. For deploying, we use the same device as the current state-of-the-art solution: a Versal VCK5000 featuring Field Programmable Gate Array (FPGA) and 400 AIEs.

However, the sole software programmable accelerator is insufficient to deliver enhanced end-to-end performance. Indeed, integrating optimized hardware modules into pyramidal registration requires careful HW/SW co-design, as the surrounding software can easily become a bottleneck. For example, in pyramidal registration, each pyramid level operates on a different image resolution, thus implying multiple input resizes. Similarly, prior work shows that registration can operate on an informative 3D sub-volume [24], reducing the number of processed pixels and overall latency. However, this shifts part of the cost to software pre-processing: automatically identifying such a sub-volume remains an **open challenge**, requiring manual, time-consuming user selection. To preserve the gains obtained by avoiding reconfiguration, the input preparation delay should be lower than the bitstream reconfiguration overhead (∼ 2 s on PCIe-based devices). Thus, ***our third contribution*** ❸ is a co-designed high-performance software module, namely Data Pre-Processing Unit (DPPU), to prepare the pyramidal levels input. Our DPPU supports both manual and automatic modes: this latter relies on **a novel heuristic** to detect the informative sub-volume in input images. Paired with *PeterPan*, it allows for efficient scheduling of hardware runs across pyramidal levels, ensuring no stalls from delaying software steps (§III-C).

Overall, *PeterPan* suits pyramidal registration, delivering an end-to-end 3.06× speedup and 2.74× energy-efficiency improvement against the current state-of-the-art solutions.

For reproducibility, we open-source and provide artifacts for both *PeterPan* design and the entire pyramidal registration [1].

II. Background

AIE–PL Systems Overview – Our work targets AIE-PL SoCs, such as the VCK5000, which feature the PL with a cluster of 400 AIEs, interconnected via a Network on Chip (NoC) [25]–[27]. Each AIE is a statically scheduled VLIW processor equipped with 32 KB of local data memory, vector units for both floating- and fixed-point arithmetic, and dedicated communication ports. The AIEs are arranged in a two-dimensional grid, enabling highly parallel, multi-kernel workloads.

AIE–PL Terminology – According to vendor documentation, we define a single AIE VLIW processor as an *AIE core*. Each AIE core can execute one or more functions, namely *AIE kernels*, through time multiplexing. In this work, each AIE core executes a single AIE kernel. AIE cores reside in the *AIE region* and are interconnected through a directed graph, referred to as the *AIE graph*. Similarly, we denote the FPGA fabric as the PL, and each function mapped onto it as a *PL kernel*.

[1]https://github.com/necst/peterpan

Listing 1. AIE Kernel GEMM structure.

```
1  // includes and other constants
2
3  const int boundary_i = h1 / 2;
4  const int boundary_j = w2 / 8;
5  const int boundary_k = w1 / 8 - 1;
6
7  void kernel_gemm(...) {
8    for (unsigned int i = 0; i <
         boundary_i; i++)
9      chess_loop_range( boundary_i ,
           boundary_i ){
10       for (unsigned int j = 0; j <
             boundary_j; j++)
11           chess_loop_range(boundary_j,
                 boundary_j){
12           for (unsigned int k = 0; k <
                 boundary_k; k++)
13               chess_loop_range(boundary_k,
                     boundary_k){
14             ...
15           }
16           // Final reduction iter ...
17       }
18   }
19 }
```

Pyramidal Image Registration – The algorithm aligns a floating image against a reference through a multi-resolution approach [23]. The process starts from a coarse resolution level to estimate large transformations efficiently and robustly, and progressively refines the alignment at higher resolutions to capture fine-grained details. This hierarchical strategy improves computational efficiency and mitigates the risk of converging to incorrect local minima, particularly in the presence of large displacements, noise, or complex deformations [28].

III. Pyramidal Registration Co-Design

To assess the flexibility gains, a suitable case-study is pyramidal image registration; since each level uses a different workload size, software programmability avoids costly reconfigurations. First, §III-A introduces our structured approach for software programmability, focusing on different AIE kernel examples and highlighting the root causes of flexibility limitations. Then, §III-B applies this method to design *PeterPan*, an open-source hardware module for accelerating image registration. We describe its components, key design challenges, and its implementation of the proposed flexibility-oriented approach. Finally, §III-C describes the pyramidal image registration application, using *PeterPan* accelerator and shows the need for an optimized software counterpart.

A. Towards FPGA-based Software Programmability for AIE

To propose a structured method, we first analyze different AIE kernels to highlight the root cause behind the lack of flexibility. We consider two different examples, Listing 1 and Listing 2, showing two state-of-the-art implementations; code is taken from the respective public repositories and presents the AIE flexibility limitations. Listing 1 shows a matrix-multiplication kernel from ARIES [10], a tool that automatically generates Versal-based accelerators from Python code. ARIES applies vectorization and scheduling pragmas to improve performance, while configuring the FPGA logic to stream data to the AIE.

Listing 2. AIE kernel for image transformation with compile-time bounds.

```
constexpr int DIM = 512;
constexpr float H_DIM = DIM / 2.f;
constexpr int DIM_DIV_32 = DIM / 32;

void transform(
 input_window_float* coords_in_cc,
 input_window_float* coords_in_rr,
 output_window_float* coeff_R) {

  // === Loop over rows ===
  for (int r = -H_DIM; r < -H_DIM + DIM;
       ++r) {
    chess_loop_range(32,)
     // Loop column batches
    for (int c = 0; c < DIM_DIV_32; c
         += 32) {
     chess_loop_range(32,)
      // compute
    }
  }
}
```

Since AIEs are statically scheduled, ARIES uses pragmas to optimize the design, applying, for example, software pipelining and unrolling; in this case, CHESS_LOOP_RANGE fixes loop bounds at compile time, enables efficient instruction scheduling, and permits software pipelining for dynamic loop ranges [29]. While enhancing performance, such optimizations also impose constraints on input sizes, as the minimum data specified in the CHESS_LOOP_RANGE must be provided to avoid deadlocks. Similarly, loop sizes are defined at compile time to ease static scheduling, thereby fixing the input shape. In other words, for a GEMM-intensive workload, multiple bitstreams are required to handle different matrix sizes.

Similarly, focusing on the image registration case-study, Listing 2 presents the transformation kernel in TRILLI, the state-of-the-art accelerator for image registration. It targets both AIE and PL in a VCK5000, using constant expressions computed at compile time. Based on such values, it fixes dimensions for both loops and pragmas. In other words, to support different resolutions, such a design needs multiple bitstreams.

On a similar path, many other examples follow the same strategy, favouring compile-time optimizations rather than flexibility [30]–[34]. In contrast, our method follows a different strategy: we do not fix the workload size at compile time. Instead, to feature software programmability and cover scenarios where runtime parameters are not supported, we suggest four guidelines (① – ④), relying on a main assumption: the PL can forward data to the AIE layer.

① Each run starts in the PL: a kernel retrieves the configuration from the host, assembles a metadata packet, and injects it ahead of the payload stream. The packet carries the runtime workload information for the AIE graph; since different cores may require different information, the metadata embodies all required entries. Figure 2 depicts an example of a metadata packet in an accelerator for image registration, which needs as information: the transformation parameters to be applied, the 3D image resolution ($Depth, N_{row}, N_{col}$), and other application-specific values easily retrieved in PL. Notably, since AIE kernels typically perform vectorized reads/writes (and window-based

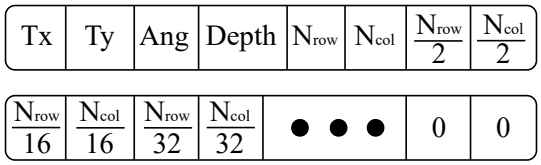


Fig. 2. Metadata packet for *PeterPan*. It embodies transformation parameters (T_x, T_y, Ang), the 3D image depth, and other parameters for each AIE core.

kernels consume fixed-size windows), the metadata packet size must be aligned to the *maximum* access granularity among the AIE kernels, to prevent any one from over-reading into the payload stream. Let B_{meta} be the metadata size in bytes. In **stream-based** AIE kernels, the injected packet must occupy an integer number of vector reads, thus its transmitted size is: $B_{\text{pkt}} = \left\lceil \frac{B_{\text{meta}}}{V_{\max}} \right\rceil \cdot V_{\max}$, where $V_{\max}$ is the *maximum vector access size* (in bytes) among the AIE kernels along the metadata path. For example, considering three cores reading respectively 32-, 64-, and 128-bytes per vector, the metadata must be a multiple of 128 bytes, to ensure each core can perform an integer amount of reads without erroneously consuming payload data. In **window-based** AIE kernels [35], which consume inputs as fixed-size windows, the metadata packet must occupy an integer number of window reads; hence: $B_{\text{pkt}} = \left\lceil \frac{B_{\text{meta}}}{W_{\max}} \right\rceil \cdot W_{\max}$, where $W_{\max}$ is the *largest window size* (in bytes) among the AIE kernels that read the metadata. In both cases, metadata is *zero-padded* up to B_{pkt}, ensuring that the consumption process targets only metadata tokens and leaves the payload stream intact for the subsequent computation. When both stream- and window-based kernels consume the metadata, we conservatively align the packet to $G = \text{lcm}(V_{\max}, W_{\max})$ and pad to $B_{\text{pkt}} = \lceil B_{\text{meta}}/G \rceil \cdot G$.

② The starting PL kernel forwards the metadata packet to any subsequent AIE. Thus, the AIE design should implement a directed graph that guarantees reachability: every AIE must be reachable at least once from another core or from a PL kernel capable of generating the metadata packet.

③ Each AIE kernel first waits for the metadata. After storing the required values, it forwards the metadata unchanged to the next core, ensuring that each one receives the runtime workload information. Then, the kernel starts processing the payload.

④ When an AIE kernel has multiple input streams, each stream carries an identical (replicated) metadata packet. Before consuming payload data, the kernel must therefore consume one metadata packet from *every* input: it parses the metadata from a designated stream and discards it from the others, ensuring that all inputs are aligned and only payload samples remain when computation starts.

In terms of applicability, both the examples considered in Listing 1 and Listing 2 already fulfill our main assumption, since the PL already streams payload data to the AIEs. Conversely, a design starting its computation in AIE needs adaptation, as it must follow ①, thus devising the PL kernel that starts the computation and creates the metadata packet, and ② - ④, to ensure the AIE computation starts upon receiving the runtime

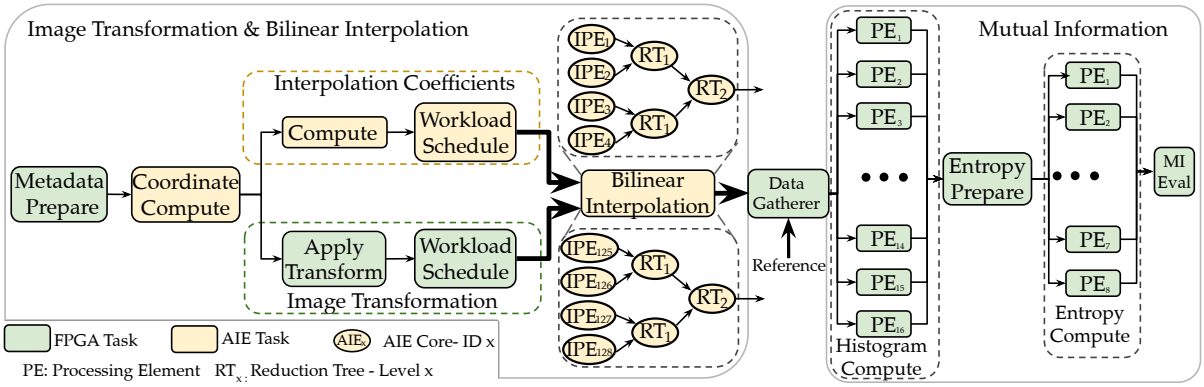


Fig. 3. *PeterPan* system design. A PL kernel streams metadata to the AIE graph; the PL computes transformed pixels, while the AIE computes interpolation weights. These are combined for bilinear interpolation across 128 AIE cores. The resulting stream is processed by a PL MI kernel to compute the similarity metric.

workload information. Since metadata is carried alongside the data stream, AIE kernels that do not require runtime flexibility can discard it. This enables heterogeneous AIE pipelines, in which only selected kernels consume metadata while others can benefit from static-scheduling optimizations.

B. PeterPan Runtime-Adaptable Accelerator

To validate the proposed methodology, we design *PeterPan*, an adaptive AIE-PL accelerator for 3D image registration.

To define *PeterPan*'s goal, we analyze the registration pipeline, which follows three steps: (i) an optimization pass that explores the transformation parameters space, (ii) a transformation that warps the floating volume, and (iii) a Mutual Information (MI) evaluation that measures alignment quality. These steps are iterated until convergence, both in single-pass registration and at each level of pyramidal registration. Prior work identifies transformation and MI as the most compute-intensive kernels [16], [17], accelerating the former with an AIE–PL design and implementing the latter with an existing PL MI kernel [24]. We follow the same strategy: we implement on AIEs and PL the transformation (TX) – including bilinear interpolation – and couple it with the open-source MI kernel. The combined TX+MI forms the *registration step*, which is executed repeatedly until convergence within a single-resolution registration. In pyramidal registration, each pyramid level is itself a single-resolution registration; hence, the registration step is repeated across iterations and across levels.

Figure 3 depicts *PeterPan* design. Each task (rectangles) is implemented by multiple PL or AIE kernels, with the second mapped onto dedicated cores (circles). The execution starts in the PL (following ①): a kernel fetches the runtime configuration from the host and prepares the metadata packet (Figure 2). Then it streams it into the AIE region. We devise the AIEs (yellow in Figure 3) as a directed graph to ensure that metadata can reach each core (according to ②). Each core first consumes the metadata from each input stream, stores the required information, forwards the packet to downstream cores, and then processes payload data (as indicated in ③ and ④).

The first AIE task computes the transformation coordinates. To this extent, we exploit a state-of-the-art algorithm that overlaps transformation and interpolation [6]. We use two AIE cores for the *Coordinate Compute* task, each producing transformed coordinates and forwarding them to the *interpolation coefficient*, and the *image transformation* cores. The former computes the interpolation coefficients, while the latter completes the image transformation by fetching pixels from memory.

Next, the design performs bilinear interpolation by combining transformed pixels with the corresponding interpolation coefficients. This stage is compute-intensive due to the high number of floating-point operations; therefore, we exploit spatial parallelism and distribute the work across 128 AIE cores, denoted as Interpolation Processing Elements (IPEs). Each one consumes a subset of the input pixels and weights and produces the corresponding interpolated output pixels.

The interpolated pixels must then be streamed to the PL MI kernel. This requires a PL *data gatherer* (Figure 3) that collects the outputs of the AIE cores and feeds the MI kernel, which expects two input streams (floating and reference). Designing this bridge is non-trivial and requires two considerations. First, AIE streaming links natively support only 32- and 128-bit widths; when the AIE output width does not match the PL input width, Vitis inserts Data Width Converters (DWCs) to adapt the interface. Second, the MI kernel scales by widening its input streams and, as reported by prior AIE–PL designs using the same kernel, it requires a 256-bit input to avoid becoming the system bottleneck in AIE–PL systems. Therefore, the *data gatherer* (Figure 3) should ensure a 256-bit output to the MI interface while handling 128-bit AIE input streams, posing a key **AIE–PL integration challenge** with direct implications for both routing complexity and performance.

Our *first solution* avoids DWCs by keeping the AIE–PL interface at 128 bits. The *data gatherer* then creates the 256-bit data for the MI by packing two consecutive 128-bit reads per stream. However, this approach does not scale: the **large number** of parallel input streams makes the gatherer increasingly complex, as it must concurrently serve all AIE outputs while also fetching the reference volume from device memory. This significantly increases routing pressure, and the **resulting design fails** in finding a valid routing during the

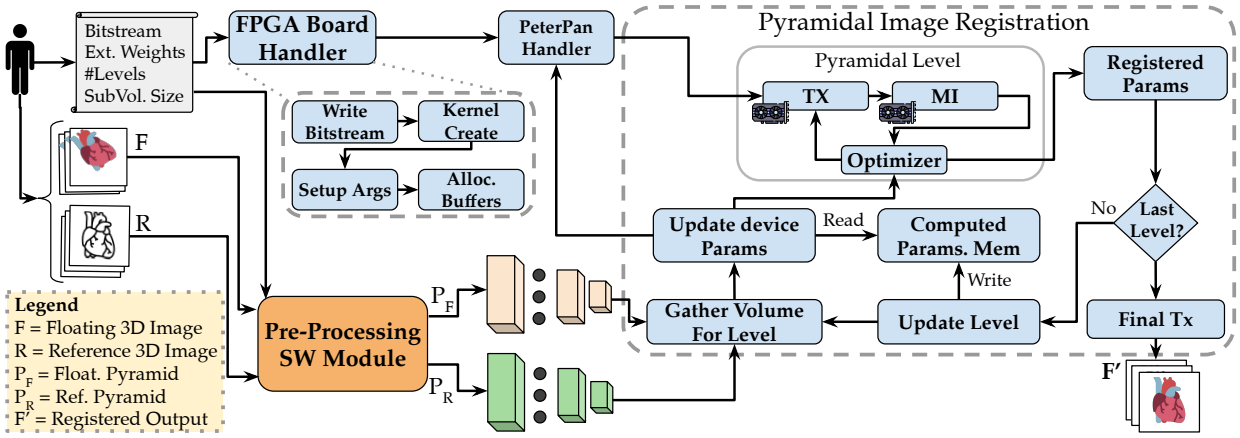


Fig. 4. Pyramidal 3D image registration application. A software module takes the input 3D images and user parameters, selects an informative sub-volume, and generates the floating/reference pairs for each pyramid level. In parallel, the accelerator is initialized and exposed through a C++ handler for runtime reconfiguration. Registration then proceeds from coarse to fine across the pyramid levels to estimate the final transform, which is finally applied to correct the full floating 3D image.

bitstream generation pipeline.

Since removing DWCs entirely causes routing issues, we instead reduce their number. A DWC is inserted whenever an AIE stream width does not match the corresponding PL interface; therefore, reducing the number of AIE–PL output stream connections directly reduces the number of DWCs. To this end, we introduce a merging stage in the AIE graph that combines two output streams into a single one, halving the number of AIE–PL connections and simplifying the *data gatherer*. Although merging two streams nominally halves throughput, the bilinear interpolation component produces an output that is 4× smaller than its input (four input pixels contribute to one output pixel in bilinear interpolation). Hence, the overall pipeline is bounded by the input bandwidth and tolerates this reduction without affecting end-to-end performance. We therefore apply a reduction-tree mechanism with up to two merging levels (RT_1 and RT_2 in Figure 3), reducing the number of output streams from 128 to 32, thereby substantially lowering the DWCs count and gatherer complexity. We denote by $PeterPan_1$ and $PeterPan_2$ the configuration with one and two merging levels, respectively.

C. End-To-End Application

PeterPan accelerates transformation and MI computation while supporting runtime-adaptable workloads, making it well suited for pyramidal registration. Image registration typically iterates three steps until convergence: an optimizer proposes new parameters, the floating image is transformed accordingly, and MI evaluates the alignment quality. In **single-resolution** registration, this loop runs on fixed-resolution inputs. In **pyramidal** registration, the same loop is executed across multiple pyramidal levels, from coarse to fine: each one performs a *single-resolution* registration, initialized with the parameters estimated at the previous (coarser) level. Figure 4 shows the application, which takes different hyperparameters as input and pairs *PeterPan* with a software module that prepares the input images for each level.

A mandatory input is *PeterPan*'s bitstream. Figure 4 shows our *FPGA Board Handler*, an abstraction that automatically manages buffer allocation/deallocation, optional padding, and returns a C++ object, namely *PeterPan Handler*. Such an object exposes a user-friendly API to customize the hardware parameters like the applied transformation – required multiple times in each *pyramidal level* step – and the workload sizes – adjusted before starting each pyramidal level.

Another key user input is the sub-volume depth D_z. Prior work showed that accurate rigid registration can be obtained by optimizing on a subset of slices, since a rigid transform applies consistently across the entire 3D volume [24]. Hence, we can estimate the registration parameters on an informative sub-volume and then apply the resulting transform to the full-resolution floating 3D image. However, selecting the most informative sub-volume **remains an open problem:** different choices can yield different outcomes, and state-of-the-art methods often rely on user-specified sub-volumes rather than automatic detection [24].

Another hyperparameter is the number of pyramid levels. Starting from the coarsest level, the algorithm computes an initial coarse alignment and outputs the estimated transformation parameters. Each subsequent level then uses the previous estimate as initialization, processes higher-resolution volumes, and refines the transformation. At the final level, a few additional iterations are performed on the original-resolution images. The resulting parameters are finally applied to the full floating volume through software transformation.

Given this workflow, software pre-processing can dominate runtime, as it must (i) extract the sub-volume and (ii) prepare the inputs for each pyramid level. We therefore introduce a **DPPU**, depicted in Figure 5. We design the DPPU to be efficient and, when applicable, to overlap with bitstream programming (which in our case takes roughly 2 seconds). It extracts and resizes either a manually specified sub-volume or an **automatically selected** sub-volume, chosen according to image features properly weighted by user-defined coefficients. In our implementation, we constrain D_z to be a multiple of 32 to align with the AIE vector access granularity. To remain general and reusable, the DPPU supports multiple feature maps, including

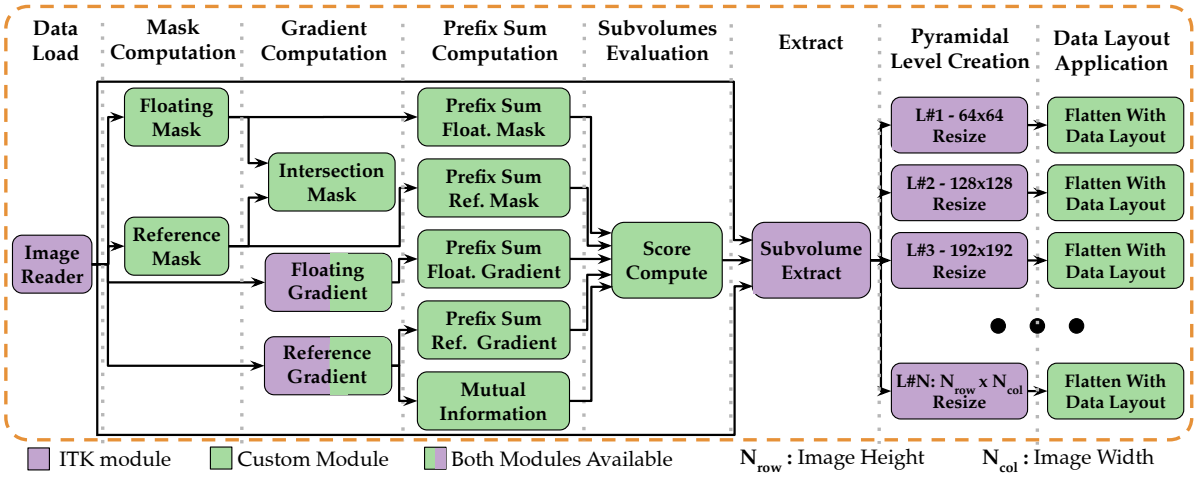


Fig. 5. Data pre-processing pipeline. Combining ITK modules with custom parallel operations, *PeterPan* application pre-processes the input volumes to find the most informative sub-volumes. Such a sub-volume is then resized to build the pyramidal levels requested by the end-user. Finally, each pair of reference and floating sub-volumes, once per level, is reshaped into a depth-wise data layout to favour the accelerator processing.

image masks (to focus on informative pixels) and gradients (to capture structural content). The scoring is modular: users can either disable any feature by setting its weight to zero or add more features whenever needed.

The DPPU implements a staged pipeline, parallelized with OpenMP [36], and built on ITK for format-agnostic I/O and resizing [37]. For automatic sub-volume selection, it first computes binary masks and gradients for each image in the reference (R) and floating (F) volumes. Then, it evaluates all depth-D_z candidates using a weighted combination of these two features, plus an additional metric: the MI. Including such a metric explicitly ties sub-volume selection to the registration objective, since MI is the similarity metric optimized during alignment. Equation (1) depicts our scoring function:

$$\begin{aligned}\text{Score}(R,F) = {} & W_0\,\text{Mask}_R + W_1\,\text{Mask}_F + W_2\,\text{Grad}_R \\ & + W_3\,\text{Grad}_F + W_4\,\text{MI}_{R-F} \end{aligned} \tag{1}$$

To evaluate the sub-volumes efficiently, we apply an axial prefix-sum for each feature, reducing the scan complexity from $O\big((Z - D_z + 1)\,D_z\,N_x\,N_y\big)$ to $O(Z\,N_x\,N_y) + O(Z)$. Indeed, by using the prefix-sum array, each feature value in a sub-volume $[z_0, z_0 + D_z)$ is obtained in constant time. For example, considering the floating mask in any sub-volume $[z_0, z_0 + D_z)$:

$$\text{Mask}_\text{F}(z_0) = \text{PS}_{maskF}[z_0 + D_z] - \text{PS}_{maskF}[z_0].$$

In practice, this yields a 30× speedup, selecting a sub-volume in less than 1 s. Finally, the selected sub-volume is resized to generate the pyramid levels and reshaped into a depth-wise layout to improve PL sequential accesses [12].

IV. Experimental Setup and Results

For *PeterPan* design, we use Vitis 2023.1, relying on XRT 2024.2 runtime for deployment [38]. We target a Versal VCK5000 QDMA card, with a PL running at 210 MHz. CPU-based parts are compiled with `-O3` optimizations. Each measurement is averaged over 50 runs, using vendor tools for power measurement [38], [39]. As data, we use publicly available PET and CT scans sized $512 \times 512 \times 256$ [40], [41][2].

Firstly, §IV-A measures *PeterPan* **kernel execution time** to quantify the performance cost of runtime adaptability. We then compare *PeterPan* with solutions on CPU [37], [42], FPGA [24], GPU [43], [44], and the same VCK5000 device [6].

Then, §IV-B evaluates the DPPU, measuring the extracted sub-volume goodness in terms of resulting registration quality with respect to any candidate sub-volume.

Finally, §IV-C evaluates the full registration. First, we consider different *PeterPan* pyramidal configurations to find the highest-performance one. Then, we compare it against CPU-based libraries CPU [42], [45] and other accelerated solutions on GPU [44], FPGA [24], and VCK5000 [6]. We consider **end-to-end** latency – with PCIe data transfers – and energy (J). Table I reports codename, setup, and power cost for each solution.

A. PeterPan Fixed-Instance Evaluation

Here, we quantify the losses incurred by our method to achieve runtime flexibility. We consider *PeterPan*'s kernel execution time for the sole accelerator task (TX + MI) and compare it against other accelerated solutions.

[2]Patient: C3N-00704, Study: Dec 10, 2000 NM PET 18 FDG SKULL T, CT: WB STND, PET: WB 3D AC)

TABLE I
Machine configurations and codenames with associated power costs: TDP for CPU-only cases, accelerator power for the others

Code Name	CPU	Accelerator	PCIe	Power [W]
PeterPan$_1$ and *PeterPan*$_2$	AMD EPYC 7V13	VCK5000	8x4.0	44.65
TRILLI [6]	AMD EPYC 7V13	VCK5000	8x4.0	40.00
HEPHAESTUS [24]	AMD Ryzen 7 3700X	Alveo U280	8x4.0	32.00
ATHENA A5000 [44]	AMD Ryzen 7 5800X	RTX A5000	16x4.0	64.00
ATHENA A100 [44]	Arm Neoverse N1	RTX A100	16x4.0	72.00
ATHENA RTX4050 [44]	i7-13620H	RTX 4050	4x4.0	48.00
ITK_U7155H [37]	Intel Ultra 7 155H	-	-	28.00
Matlab_i7-13620H [42]	i7-13620H	-	-	45.00

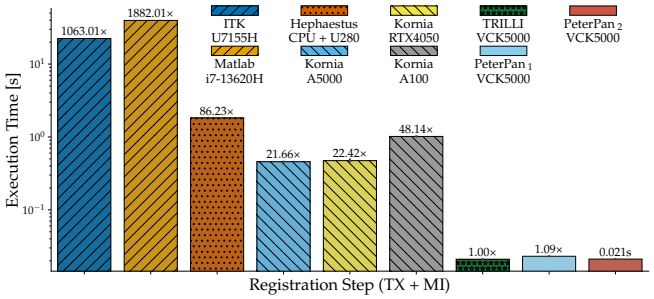


Fig. 6. Registration step (TX+MI) latency analysis. The non-optimized *PeterPan* version presents a 60% overhead, mitigated by the AIE-PL co-design strategy.

Figure 6 depicts the latency analysis, showcasing both version of *PeterPan*. The 2-level merging, *PeterPan*$_2$, hits slightly higher working frequencies (210MHz vs 200MHz), leading to better performance. Against competitors, *PeterPan*$_2$ consistently outperforms the FPGA- and GPU-accelerated solution, attaining 21.66× improvement against the fastest GPU-based implementation on NVIDIA A5000. Compared with AIE-PL competitors, *PeterPan*$_2$ achieves performance comparable to the current state-of-the-art implementation, TRILLI. Such a similarity is attributable to the similar partitioning strategy applied. Indeed, the design is currently compute-bounded on the interpolation task, and both solutions leverage 128 IPEs to parallelize it. Furthermore, as we use the same algorithm to overlap transformation and interpolation, both *PeterPan*$_2$ and TRILLI are bound to power-of-two IPE; consequently, further scaling IPEs would exceed the AIE budget. In terms of energy efficiency, instead, *PeterPan*$_2$ is less efficient than TRILLI (0.90×) as it has a higher power cost and equal latency.

Notably, the same analysis applied to the sole image transformation component shows a similar trend, with both AIE–PL solutions attaining strictly comparable results and outperforming any other GPU and FPGA-based implementation.

Remark #1: *PeterPan*$_2$ attains **performance comparable** to the state-of-the-art accelerator, but ensuring **adaptability** to different workload sizes.

Finally, Table II reports *PeterPan*$_2$ and TRILLI resource costs. TRILLI uses less DSP because it features a smaller spatial parallelism in the MI kernel – increasing it reduces the PL frequency and performance. The other resources are comparable: *PeterPan*$_2$ uses more AIEs cores, due to the reduction-tree mechanism used to reduce the DWCs count, but

TABLE II
PeterPan$_2$ AND TRILLI: RESOURCE UTILIZATION SUMMARY.

	BRAM	URAM	DSP	LUT	REG	AIE
PeterPan$_2$	177 (18.30%)	128 (27.65%)	93 (4.73%)	485,646 (53.97%)	457,113 (25.40%)	237 (59.25%)
TRILLI	182 (18.87%)	128 (27.65%)	55 (2.79%)	484,024 (53.79%)	471,336 (26.19%)	205 (51.25%)

TABLE III
REGISTRATION-PARAMETER ERRORS IN BEST, WORST, AND AVERAGE CASES.

Case	$Error_{Tx}$ [px]	$Error_{Ty}$ [px]	$Error_{Ang}$ [°]
Best	0.0176	0.0084	0.1266
Worst	2.7333	0.4758	0.4635
Average	0.4177	0.3668	0.2192

saves BRAM and registers.

B. Data Pre-Processing Unit Analysis

The DPPU supports both manual and automatic modes. Here, we focus on the latter, which is more complex because it selects the informative sub-volume and prepares the pyramid levels' inputs. For the analysis, we select a sub-volume size $D_z = 32$.

For the DPPU input weights – used to balance the scoring function – we adopt a simple *algorithm-driven* choice with *no prior knowledge* about the specific input: 0 weight for gradients, as we do not use this information; 1 weight for masks, and 0.3 for MI, to balance its contribution with respect to the masks.

To assess the quality of the extracted sub-volume, we use it as input to the registration task and measure its accuracy as the error between estimated and ground-truth (ideal) registration parameters. Then, we rank the extracted sub-volume among all candidates by accuracy and report its percentile (i.e., whether it lies within the top x% lowest-error sub-volumes).

For this analysis, we consider 20 PET and CT volumes, each deformed with randomly sampled deformations drawn from a uniform distribution. For such deformation, we keep literature deformation ranges of $[-128, 128]$ pixels in translation along X and Y axis and $[-45°, 45°]$ in rotation around Z [44].

Remark #2: By leveraging the **proposed heuristic**, our DPPU **automatically** extracts a sub-volume that ranks, by accuracy, in the **top** 14% among all the possible sub-volumes.

Quantitatively, Table III depicts the errors in the best, worst, and on average. Notably, *PeterPan* achieves the same average accuracy as literature solutions, with an average error of less than one pixel per transformation parameter. Overall, the end-to-end pipeline latency is 0.81 s, meeting our goal of staying below the bitstream reconfiguration time (≈ 2s on PCIe devices).

C. Rigid Image Registration Analysis

To compare *PeterPan*$_2$ pyramidal registration against competitors, we must find the best configuration in terms of number of levels and sub-volume depth D_z extracted by the DPPU.

Figure 7 reports end-to-end latency while scaling both parameters. Horizontal lines correspond to a 1-level pyramid, i.e., *single-resolution* registration over a sub-volume of depth D_z. We evaluate all configurations from 64×64 up to the original image size (512×512) with a step of 64. Each additional level introduces extra data transfers to the accelerator and an additional registration stage. Consequently, latency initially decreases until a minimum is reached; after that, the overhead of extra levels dominates and latency increases. Regarding D_z, smaller sub-volumes improve performance because fewer

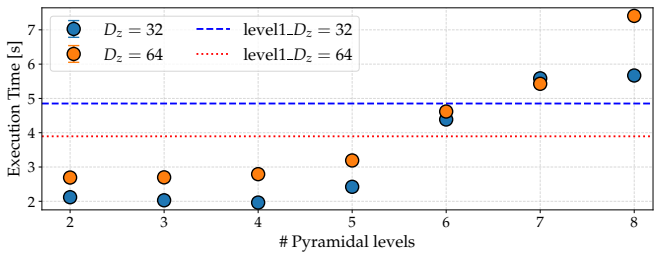


Fig. 7. Pyramidal level scaling with $D_z \in [32, 64]$. After a certain minimum, extra-level overhead exceeds the pyramidal benefits, worsening performance.

pixels are processed while still achieving sub-pixel accuracy. We therefore select four pyramid levels and $D_z = 32$.

We compare *PeterPan*$_2$ configuration against hardware-accelerated solutions from the literature. In addition, we include two variants to isolate the contribution of each optimization: (i) $TRILLI_PYR$, that integrates TRILLI into our pyramidal registration pipeline (requiring a bitstream per level); (ii) *PeterPan_onlyPyr*, performing pyramidal registration on entire 3D images (no sub-volume extraction). Figure 8 reports end-to-end latency, including data transfers for each solution. For energy (J), we use the power costs in Table I. Overall, AIE–PL designs outperform GPU- and FPGA-based alternatives. Quantitatively, *PeterPan*$_2$ achieves a 6.93× speedup over the fastest GPU solution (NVIDIA A100) and a 11.17× energy improvement over the most energy-efficient GPU alternative (still A100-accelerated). Then, by comparing the two AIE-PL alternatives, *PeterPan*$_2$ and TRILLI, we prove the improvements achieved thanks to software programmability. First, we consider $TRILLI_PYR$. This solution is 3.31× **slower** than the *single-resolution* TRILLI version. In other words, the reconfigurability overhead completely cancels any pyramidal benefit, leading to even worse performance than non-optimized applications. In contrast, the runtime-flexible *PeterPan_onlyPyr* removes such overheads and becomes 1.51× faster than *single-resolution* TRILLI and 5.01× faster than $TRILLI_PYR$, directly quantifying the impact of avoiding reconfiguration. Finally, the complete *PeterPan*$_2$ pipeline achieves a 3.06× speedup over *single-resolution* TRILLI and a 2.74× improvement in energy efficiency, despite a +4W higher power consumption (Table I).

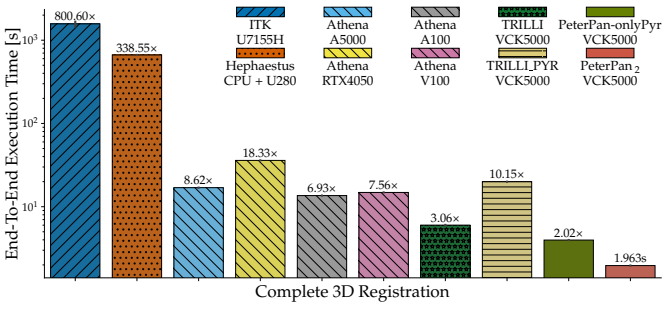


Fig. 8. Latency comparison against literature. *PeterPan_onlyPyr* outperforms any other solution, with further gains attained using the DPPU (*PeterPan*$_2$).

Remark #3: Our approach for software programmability permits *PeterPan* to support a more efficient software pipeline that better leverages the accelerator. This delivers a 3.06× speedup and a 2.74× energy efficiency improvement over the state-of-the-art AIE–PL solution in **the end-to-end application**.

V. Related Work

Dynamic Versus Static Scheduling – To address the inflexibility of static scheduling, prior work explored dynamic scheduling for VLIW processors (including AIEs) [46]–[50]. However, its performance depends on the compiler's ability to extract instruction-level parallelism [51]. As a result, VLIWs are mainly used for regular workloads where fine-grain parallelism can be reliably exposed [52]. Nevertheless, static scheduling remains actively studied, offering the best performance-to-cost ratio for regular embedded workloads, while irregular and control-dominated tasks benefit from runtime flexibility [52]. Hybrid approaches combine both paradigms, enabling heterogeneous pipelines that retain their advantages [53]. Following this approach, we keep VLIW static scheduling where effective and introduce runtime flexibility via FPGA orchestration.

Image Registration Accelerators – Given its compute-intensive nature, image registration is often offloaded to hardware accelerators. CPU-based frameworks offer flexibility but limited performance [37], [42]. FPGA solutions can deliver high performance and energy efficiency at the cost of design complexity [16], [24], while GPU-based approaches provide high throughput with higher energy consumption [43], [44]. Deep-learning alternatives have also been explored, but are more effective for non-rigid than affine/rigid registration and still typically require heuristic-based preprocessing [54], [55].

Pyramidal Image Registration - To improve robustness and efficiency of image registration, research efforts explore coarse-to-fine pipelines [22], [56], [57], theoretically relating low- and high- resolution objectives for rigid transformations [58]. Such pyramidal pipelines are widely used in medical multimodal rigid registration in both 2D/2D and 3D/3D scenarios [59], [60]. The improvements and wide adoption of such an algorithm further motivate this research.

VI. Conclusions

This research proposes a method for AIE–PL designs adaptable to different input workloads. We rely on FPGA orchestration to forward the runtime workload information, collected and used by each AIE kernel. To validate our method, we devise *PeterPan*, an accelerator for image registration **adaptable** to different input workloads. Then, we pair *PeterPan* with an ad-hoc DPPU, that relies on a **novel heuristic** to extract sub-volumes and enhance pyramidal registration. Overall, we exploit *PeterPan* flexibility attaining an **end-to-end** 3.06× speedup and 2.74× energy efficient improvement over literature AIE–PL solutions.

VII. Acknowledgements

This work was supported by the European Commission (HORIZON EIC Cancerscan, GA 101186829) and partially by the AMD University Program. The authors thank the anonymous reviewers for their feedback.

References

[1] M. Langhammer and G. Constantinides, "Soft gpgpu versus ip cores: Quantifying and reducing the performance gap," in *Proceedings of the 14th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, ser. HEART '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 44–52. [Online]. Available: https://doi.org/10.1145/3665283.3665288

[2] M. Langhammer, G. Baeckler, and K. Bozman, "A 950 mhz simt soft processor," 2025. [Online]. Available: https://arxiv.org/abs/2504.07538

[3] S. Gribok, M. Langhammer, and B. Pasca, "Fpga modular multipliers using hybrid reduction techniques," in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, 2024, pp. 288–296.

[4] G. R. Goslin, "Guide to using field programmable gate arrays (FPGAs) for application-specific digital signal processing performance," in *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, J. Schewel, P. M. Athanas, V. M. B. Jr., and J. Watson, Eds., vol. 2914, International Society for Optics and Photonics. SPIE, 1996, pp. 321 – 331. [Online]. Available: https://doi.org/10.1117/12.255830

[5] D. Kumarathunga, Q. Hu, and Z. Fang, "Autontt: Automatic architecture design and exploration for number theoretic transform acceleration on fpgas," in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2025, pp. 1–9.

[6] G. Sorrentino, P. S. Galfano, E. D'Arnese, and D. Conficconi, "Soaring with trilli: An hw/sw heterogeneous accelerator for multi-modal image registration," in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2025, pp. 56–65.

[7] R. J. Weber, B. J. LaMeres, and J. A. Hogan, "Real-time, dynamic hardware accelerators for blas computation," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 5, no. 1, pp. 227–236, 2017.

[8] M. Huang, V. K. Narayana, H. Simmler, O. Serres, and T. El-Ghazawi, "Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, Nov. 2010. [Online]. Available: https://doi.org/10.1145/1862648.1862650

[9] J. Olivito, F. Serrano, J. A. Clemente, H. Mecha, and J. Resano, "Analysis of the reconfiguration latency and energy overheads for a xilinx virtex-5 field-programmable gate array," *IET Computers & Digital Techniques*, vol. 12, pp. 150–157, 2018. [Online]. Available: https://digital-library.theiet.org/doi/abs/10.1049/iet-cdt.2016.0095

[10] J. Zhuang, S. Xiang, H. Chen, N. Zhang, Z. Yang, T. Mao, Z. Zhang, and P. Zhou, "Aries: An agile mlir-based compilation flow for reconfigurable devices with ai engines," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 92–102. [Online]. Available: https://doi.org/10.1145/3706628.3708870

[11] G. Sorrentino, P. S. Galfano, E. D'Arnese, and D. Conficconi, "Voted – versal optimization toolkit for education and heterogeneous systems development," in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025, pp. 1–5.

[12] P. S. Galfano, G. Sorrentino, E. D'Arnese, and D. Conficconi, "Co-designing a 3d transformation accelerator for versal-based image registration," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 219–222.

[13] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "Chimps: a high-level compilation flow for hybrid cpu-fpga architectures," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 261. [Online]. Available: https://doi.org/10.1145/1344671.1344720

[14] L. G. Brown, "A survey of image registration techniques," *ACM computing surveys (CSUR)*, vol. 24, no. 4, pp. 325–376, 1992.

[15] B. Zitova and J. Flusser, "Image registration methods: a survey," *Image and vision computing*, vol. 21, no. 11, pp. 977–1000, 2003.

[16] E. D'Arnese, D. Conficconi, E. D. Sozzo, L. Fusco, D. Sciuto, and M. D. Santambrogio, "Faber: A hardware/software toolchain for image registration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 291–303, 2023.

[17] D. Conficconi, E. D'Arnese, E. Del Sozzo, D. Sciuto, and M. D. Santambrogio, "A framework for customizable fpga-based image registration accelerators," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 251–261.

[18] S. E. Salcudean, H. Moradi, D. G. Black, and N. Navab, "Robot-assisted medical imaging: A review," *Proceedings of the IEEE*, 2022.

[19] M. Sinclair, A. Schuh, K. Hahn, K. Petersen, Y. Bai, J. Batten, M. Schaap, and B. Glocker, "Atlas-istn: Joint segmentation, registration and atlas construction with image-and-spatial transformer networks," *Medical Image Analysis*, vol. 78, p. 102383, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361841522000354

[20] Y.-J. Wang, P.-J. Chen, X. Liang, and Y.-H. Lin, "Augmented reality with image registration, vision correction and sunlight readability via liquid crystal devices," *Scientific Reports*, vol. 7, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:4130641

[21] G. M. K. Kumar, J. Mendola, and A. Shmuel, "Nestedmorph: Enhancing deformable medical image registration with nested attention mechanisms," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2025.

[22] P. Thevenaz and M. A. Unser, "Spline pyramids for intermodal image registration using mutual information," in *Wavelet Applications in Signal and Image Processing V*, A. Aldroubi, A. F. Laine, and M. A. Unser, Eds., vol. 3169, International Society for Optics and Photonics. SPIE, 1997, pp. 236 – 247. [Online]. Available: https://doi.org/10.1117/12.292794

[23] W. Yin, J.-J. Sonke, and E. Gavves, "Pc-reg: A pyramidal prediction–correction approach for large deformation image registration," *Medical Image Analysis*, vol. 90, p. 102978, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361841523002384

[24] G. Sorrentino, M. Venere, D. Conficconi, E. D'Arnese, and M. D. Santambrogio, "Hephaestus : Codesigning and automating 3d image registration on reconfigurable architectures," in *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 22, no. 5s, 2023. [Online]. Available: https://doi.org/10.1145/3607928

[25] "Versal adaptive soc technical reference manual (am011)," https://docs.xilinx.com/r/en-US/am011-versal-acap-trm, accessed: 2023-11-17.

[26] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel, "Network-on-chip programmable platform in versaltm acap architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 212–221.

[27] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Jayadev, J. Cuppett, A. Morshed, B. Gaide, and Y. Arbel, "Versal network-on-chip (noc)," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2019, pp. 13–17.

[28] M. A. Unser, A. Aldroubi, and C. R. Gerfen, "Multiresolution image registration procedure using spline pyramids," in *Mathematical Imaging: Wavelet Applications in Signal and Image Processing*, A. F. Laine, Ed., vol. 2034, International Society for Optics and Photonics. SPIE, 1993, pp. 160 – 170. [Online]. Available: https://doi.org/10.1117/12.162061

[29] *Software Pipelining of Loops*, 2025th ed., AMD, 2025, release Date: 2025-11-26. [Online]. Available: https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/Software-Pipelining-of-Loops

[30] E. Taka, A. Arora, K. C. Wu, and D. Marculescu, "MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine," in *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2023, pp. 96–105.

[31] AMD-Xilinx, "Petalinux aie examples," GitHub repository, 2022, accessed: 2025-12-12. [Online]. Available: https://github.com/Xilinx/plnx-aie-examples

[32] ——, "Vitis tutorials," GitHub repository, 2019. [Online]. Available: https://github.com/Xilinx/Vitis-Tutorials

[33] J. Zhuang, J. Lau, H. Ye, Z. Yang, S. Ji, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, Y. Shi, D. Chen, J. Cong, and P. Zhou, "Charm 2.0: Composing heterogeneous accelerators for deep learning on versal acap architecture," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 3, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3686163

[34] NECSTLab / Politecnico di Milano, "Aie-kmeans: K-means clustering on amd versal ai engine," GitHub repository, 2024, accessed: 2025-12-12. [Online]. Available: https://github.com/necst/AIE-kmeans

[35] X. AMD. (2022) Window vs stream communication. [Online]. Available: https://docs.xilinx.com/r/2020.2-English/ug1079-ai-engine-kernel-coding/Window-vs.-Stream-in-Data-Communication

[36] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[37] M. M. McCormick, X. Liu, L. Ibanez, J. Jomier, and C. Marion, "Itk: enabling reproducible research and open science," *Frontiers in Neuroinformatics*, vol. 8, 2014.

[38] AMD/Xilinx, *Xilinx Runtime (XRT) Tools and Utilities Guide*, 2023, accessed: 2023-12-29. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1393-xrt-tools

[39] N. Corporation, *NVIDIA System Management Interface (nvidia-smi)*, 2023, accessed: 2023-12-29. [Online]. Available: https://docs.nvidia.com/deploy/nvidia-smi-user-guide/

[40] K. Clark, B. Vendt, K. Smith, J. Freymann, J. Kirby, P. Koppel, S. Moore, S. Phillips, D. Maffitt, M. Pringle *et al.*, "The cancer imaging archive (tcia): maintaining and operating a public information repository," *Journal of digital imaging*, vol. 26, no. 6, pp. 1045–1057, 2013.

[41] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC), "Radiology data from the clinical proteomic tumor analysis consortium lung adenocarcinoma [cptac-luad] collection [data set]," *The Cancer Imaging Archive*, 2018. [Online]. Available: https://doi.org/10.7937/k9/tcia.2018.pat12tbs

[42] I. MathWorks, "Image processing toolbox," 1994-2020. [Online]. Available: https://mathworks.com/products/image.html

[43] E. Riba, D. Mishkin, D. Ponsa, E. Rublee, and G. Bradski, "Kornia: an open source differentiable computer vision library for pytorch," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2020, pp. 3674–3683.

[44] G. Sorrentino, M. Venere, E. D'Arnese, D. Conficconi, I. Poles, M. Santambrogio *et al.*, "Athena: a gpu-based framework for biomedical 3d rigid image registration," in *2023 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2023, pp. 1–5.

[45] B. C. Lowekamp, D. T. Chen, L. Ibáñez, and D. Blezek, "The design of simpleitk," *Frontiers in neuroinformatics*, vol. 7, p. 45, 2013.

[46] C. E. Love and H. F. Jordan, "An investigation of static versus dynamic scheduling," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 192–201, 1990.

[47] A. F. de Souza and P. Rounce, "Dynamically scheduling vliw instructions," *Journal of Parallel and Distributed Computing*, vol. 60, no. 12, pp. 1480–1511, 2000.

[48] S. Jee and K. Palaniappan, "Dynamically scheduling vliw instructions with dependency information," in *Proceedings Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*. IEEE, 2002, pp. 15–23.

[49] A. F. De Souza and P. Rounce, "On the scheduling algorithm of the dynamically trace scheduled vliw architecture," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. IEEE, 2000, pp. 565–572.

[50] B. R. Rau, *Dynamic scheduling techniques for VLIW processors*. Hewlett Packard Laboratories, 1993.

[51] J. C. Dvorak, "How the itanium killed the computer industry," *PCMag*, Jan. 2009. [Online]. Available: https://www.pcmag.com/archive/how-the-itanium-killed-the-computer-industry-236394

[52] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 26th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 127–136. [Online]. Available: https://doi.org/10.1145/3174243.3174264

[53] J. Cheng, L. Josipovic, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 288–298. [Online]. Available: https://doi.org/10.1145/3373087.3375297

[54] J. M. Sloan, K. A. Goatman, and J. P. Siebert, "Learning rigid image registration - utilizing convolutional neural networks for medical image registration," in *Bioimaging (Bristol. Print)*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:4343225

[55] S. Potluri, A. Fasih, L. K. Vutukuru, F. A. Machot, and K. Kyamakya, "Cnn based high performance computing for real time image processing on gpu," in *Proceedings of the Joint INDS'11 & ISTET'11*, 2011, pp. 1–7.

[56] P. Thevenaz and M. Unser, "Optimization of mutual information for multiresolution image registration," *IEEE Transactions on Image Processing*, vol. 9, no. 12, pp. 2083–2099, 2000.

[57] A. A. Cole-Rhodes, K. L. Johnson, J. LeMoigne, and I. Zavorin, "Multiresolution registration of remote sensing imagery by optimization of mutual information using a stochastic gradient," *Trans. Img. Proc.*, vol. 12, no. 12, p. 1495–1511, Dec. 2003. [Online]. Available: https://doi.org/10.1109/TIP.2003.819237

[58] B. Nasihatkon and F. Kahl, "Multiresolution search of the rigid motion space for intensity-based registration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 1, pp. 179–191, 2018.

[59] J. Jia, S. Huang, F. Liu, Q. Ren, G. Li, M. Cheng, C. Jin, and Y. Wu, "Automatic 2d/2d registration using multiresolution pyramid based mutual information in image guided radiation therapy," *World Academy of Science, Engineering and Technology*, vol. 80, pp. 964–968, 08 2011.

[60] J. Wu and M. Murphy, "Assessing the intrinsic precision of 3d/3d rigid image registration results for patient setup in the absence of a ground truth," *Medical physics*, vol. 37, pp. 2501–8, 06 2010.