Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

A GPU Optimization Workflow for Real-time Execution of Ultra-high Frame Rate Computer Vision Applications

Mohsen Nourazar, Brian G. Booth, and Bart Goossens

Received: date / Accepted: date

Abstract This work proposes a GPU optimization methodology for real-time execution of ultra high frame rate applications with small frame sizes. While the use of GPUs for offline processing is well-established, real-time execution remains challenging due to the lack of real-time execution guarantees, especially for embedded GPUs. Our methodology introduces guidelines and a workflow by focusing on: (a) controlling latency by means of minimization of CPU-GPU interactions; (b) computation pruning; and (c) inter/intra-kernel optimizations. Furthermore, our approach takes advantage of multi-frame processing to attain significantly higher throughput at the cost of increased latency when the application permits such trade-offs. To evaluate our optimization methodology, we applied it to the monitoring and controlling of laser powder bed fusion machines, a widely used metal additive manufacturing technique. Results show that in the considered application, the required performance could be obtained on a Jetson Xavier AGX platform, and by sacrificing latency, significantly higher throughput was achieved.

1 Introduction

In recent years, there has been a growing interest for video processing at ultra-high frame rates, exceeding 10 kfps, for different applications ranging from academic to industrial [28, 23, 16, 18]. Although Graphics Processing Units (GPUs) have been adopted in this domain, real-time execution of such applications remains challenging due to the lack of real-time execution guarantees of GPUs. With the advent of embedded GPUs, GPU accelerated real-time applications have attracted great attention. Particularly, coupling with technologies like NVIDIA GPUDirect that enables direct communication between GPUs and other PCIe devices [2], has made GPUs even more suitable for these applications.

Although running applications with both ultra-high frame rates and high resolution is beyond the capabilities of these embedded GPUs, they can still offer benefits for applications that deal with small frame sizes[1]. Even with small frame sizes, achieving real-time execution poses several challenges on embedded GPUs, because of significant run-time overheads and fluctuations in the latency (jitter).

To improve real-time performance, one commonly used GPU programming technique is kernel fusion. By combining multiple kernels into a single kernel, it is possible to reduce the run-time overheads and improve data locality, at least, if the resource contention can be controlled. Kernel fusion, referred to as operator fusion in the context of neural networks, has become a common technique to improve the performance of neural networks [27], and linear algebra [11]. Despite extensive research on this topic, [17, 4], effectiveness of kernel fusion highly depends on various prior and subsequent optimizations, which will be focused in our study.

Kernel fusion, when results in a single kernel, enables the use of kernels that remain resident on the GPU. Persistent kernels can be executed multiple times without launching overheads. Although persistent kernel has already been recognized as an effective approach to improve the real-time performance of GPU kernels, [14, 6, 6, 33, 32], its applica-

✉ Mohsen Nourazar
mohsen.nourazar@ugent.be

Brian G. Booth
brian.booth@ugent.be

Bart Goossens
bart.goossens@ugent.be

Department of Telecommunications and Information Processing, imec-IPI-Ghent University, 9000 Ghent, Belgium

[1] In our work, a frame is considered ”small”, if it fits within the shared memory of a streaming multiprocessor (SM) and the size of the work (e.g., the number of pixels or elements to be processed) falls within the range of thread-block size.

tion to practical applications remains a logistical challenge, as without care-full analysis incorporating this technique can result in high register/memory usage and resource contention.

In recent years, several approaches based on Domain-Specific Languages (DSL) have emerged for automatic parallelization and GPU acceleration of algorithms, e.g. OpenACC [30], HIPAcc[19], Rootbeer [20], Halide [22], TensorComprehensions [29]. Although these programming approaches all focus on providing a DSL with compiler optimizations for implementing computer vision algorithms on GPU, they generally do not allow to control optimization trade-offs, which makes them unsuitable under stringent (e.g. real-time) performance requirements, at least for the targeted application of this work. Alternatively, a lot of programmer intervention/expertise is required to adopt CUDA specific features, which are not always accessible from the DSL.

In this regards, this work aims to propose a set of guidelines and an optimization methodology that can be potentially automated by compilers for GPU acceleration of ultra-high frame rate applications that deal with small frame sizes. Our approach consists of computation pruning followed by iterative intra- and inter-kernel optimizations that reduce both driver and kernel latency. Optimizations are designed to minimize CPU-GPU interactions while making the codes more amenable to exploit data locality and data reuse opportunities. Then, data reuse techniques [8], that are adopted for ultra-high throughput, are used in combination with access patterns analysis (e.g. to avoid shared memory bank conflicts). Due to constraints, not every candidate will eventually be selected, and to select the final candidates, we consider adapted workload scheduling schemes in combination with parallel multi-frame processing to take advantage of underutilized GPU resources due to small frame sizes.

Although the proposed workflow focuses on small frame sizes, it is also applicable to applications with large frames. We focus on small frame sizes because: a) we aim to reduce run-time overheads, which are more significant when dealing with small frames, b) inter-kernel optimizations are particularly effective when the GPU resources are under-utilized, which is more likely when frame is small, and c) embedded GPUs, which are the intended devices for industrial applications, are more suitable for handling small frame sizes.

Many such applications exist in manufacturing and quality control systems [12, 24], where our approach can lead to significant improvements. One such application is laser powder bed fusion (LPBF), a widely used metal additive manufacturing (AM) technique that produces high quality parts for various industries [3, 25]. However, it suffers from printing defects, mainly keyhole and lack-of-fusion pores, resulting in sub-standard parts and increased scrap rates [7].

To evaluate our approach, we applied it to real-time control of an LPBF machine, which is an excellent example for our purpose. As the physical processes involved in pore formation occurs rapidly, the monitoring system needs to capture and analyze at very high frames rates ($\sim 20,000$ fps), and once pores are detected, the control system needs to act in real-time and adjust the parameters before the quality of the part is significantly reduced. In addition, monitoring needs to cover only the melt pool (i.e., the molten metal created by the laser) and a small area around it, which spans approximately 25×25 mm. Using a camera system capable of capturing at a resolution of 100×100 pixels with each pixel covering an area of roughly 200×200 microns in size, the pore formation phenomenon can be effectively monitored.

The acceptable latency levels depends on the required quality, which itself depends on the application. An LPBF printer's typical laser movement speed is around 1 m/s, meaning that a latency of 10 ms would result in 10 mm of non-optimized printing. For common metal AM applications, real-time control requires the latency to be in the range of microseconds to milliseconds [5]. This corresponds to, at most, a few millimeters of non-optimized printing.

The techniques used in this paper are intended to be applied directly in CUDA/OpenCL or, in a DSL that provides low-level access to some of the CUDA features (e.g., Quasar [13]) or similarly appropriate tiling/scheduling options. The proposed approach borrows from ideas of the DTSE methodology [8] (e.g. data reuse optimizations), but adapts these ideas to the setting of GPU compilation for ultra-high throughput, parallel multi-frame applications. The proposed workflow also provides a good starting point for anyone trying to accelerate such an application. Our contributions are as follows:

1. We propose a workflow for GPU acceleration of ultra-high frame rate applications, which specifically targets latency control in GPU processing by focusing on various optimization aspects, including both intra- and inter-kernel optimizations.
2. We design special kernel fusion schemes that makes the code more amenable to various optimization techniques, e.g. data reuse, while minimizes CPU-GPU interactions.
3. We enable a *latency-versus-throughput* trade-off in our methodology by adjusting the inter-kernel optimization decisions for dynamic multi-frame processing, in order to meet higher throughput requirements.
4. We utilize our approach for real-time high frame-rate LPBF monitoring, which results in the first real-time implementation of video-based monitoring systems for LPBF, to the best of our knowledge.

This paper is organized in five sections. The proposed workflow is detailed in section 2. Then, the application of the proposed workflow for the LPBF monitoring system is explained in section 3 along with a brief overview of the LPBF video analysis algorithms. Finally, sections 4 and 5 present the results, discussion, and conclusions.

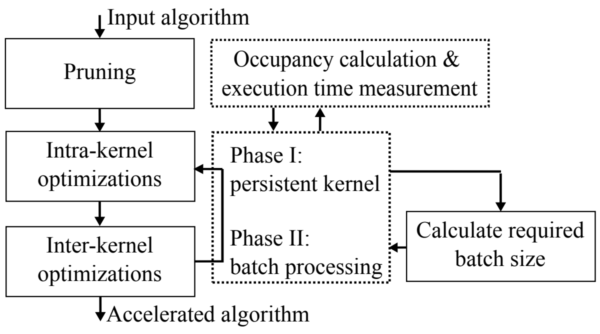

Input algorithm
Pruning
Occupancy calculation & execution time measurement
Intra-kernel optimizations
Phase I: persistent kernel
Inter-kernel optimizations
Phase II: batch processing
Calculate required batch size
Accelerated algorithm

Fig. 1: Overview of the proposed optimization workflow.

2 The Design of GPU-specific Optimizations

An overview of the proposed workflow is presented in Fig. 1. It takes a basic GPU implementation, e.g. in CUDA or Quasar, as its input and performs computation pruning in the initial step (section 2.3). Then, iterative intra- and inter-kernel optimizations are performed in two phases: 1) iterative optimizations to transform the entire algorithm into a persistent GPU kernel (sections 2.1 and 2.2), and 2) when throughput requirements are not satisfied, it takes advantage of multi-frame processing to trade latency with throughput (section 2.4). Thus, iterative optimizations are performed until the required number of frames can be processed in parallel. Both phases can be stopped early if requirements are satisfied.

The workflow includes a feedback loop that incorporates execution times and GPU occupancy calculations. Execution times are measured by NVIDIA Visual Profiler, or Quasar profiling tools, for both total and GPU-only execution times. As the process progresses, these values can even be estimated based on the applied fusion and time savings resulting from data reuse opportunities. GPU occupancy values can be obtained from the NSight compute occupancy calculator or Quasar APIs. The feedback loop influences mainly the inter-kernel optimization decisions. Subsequently, these decisions impact the intra-kernel optimizations as well.

2.1 Intra-Kernel Optimizations

To optimize the GPU kernels, we adopt a straightforward approach by focusing on two key optimization aspects:

1. *Memory throughput optimizations* to maximize the bandwidth and efficiency of data transfers by: 1) mapping data into the best suited layer of the GPU memory hierarchy (see Fig. 2), and 2) improving memory access patterns.
2. *Efficient workload scheduling* for achieving optimal GPU utilization and reducing latency by minimizing costly synchronization and reducing stalls. In addition, proper workload scheduling provides efficient communication between threads, as different communication techniques are available based on the thread mapping used.

Table 1: An overview of the available resources on GPUs.

GPU	Xavier AGX	Orin AGX	RTX A6000
SM count	8	16	84
Max. number of threads per SM	2048	1536	1536
Registers per block	65536	65536	65536
Shared mem. per SM (KB)	96	164	100
Global mem. bandwidth (GB/s)	136.5	204.8	768
Shared mem. bandwidth (GB/s)	1536	2662	19350

The intra-kernel optimization workflow consists of three steps, which starts by determining the opportunities for optimization (see below). Since there is a limit on the number of registers and amount of shared memory, generally only a limited subset of the data can be mapped into the fastest memory (see Table 1 for an overview of the available resources on GPUs). Hence, it is necessary to prioritize operations by conducting code analysis to obtain information for each identified optimization candidate. Finally, the decision variables concerning the selection of optimization techniques should be optimized. In the following subsection, we explain these steps and how optimization decisions are made.

2.1.1 Determining the opportunities for optimization

This involves identifying areas and patterns of the code where improvements can be made. Potential candidates include:

– *Out-of-place operations*, which refers to operations in which the output is stored in a new location in memory, rather than overwriting the input. Due to existence of data loading/storing, they are often suitable candidates for optimizations. Every load/store operation needs to be checked for access pattern optimization opportunities. In addition, these operations can be candidates for data reusing, sharing, and broadcasting optimizations.
– *In-place operations*, which refer to operations where threads are involved in performing tasks at the same memory location. Examples of such operations are histogram computations, calculation of statistics (e.g. sum, min, and max), cumulative operations (e.g. cumulative sum), and propagation algorithms (e.g. label propagation).
– *Expensive numerical computations* where each thread needs to perform significant amount of numerical computations. Mathematical computations (e.g., matrix multiply-accumulate (MMA), evaluation of transcendental functions, etc.) are examples of such operations.

We start with identifying expensive numerical computations. Depending on the mapping decision, these elements may qualify as candidates for either in-place or out-of-place

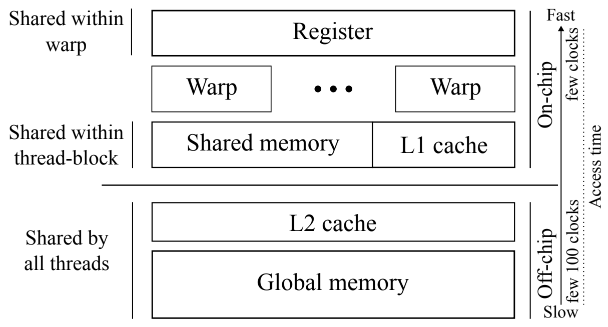


Fig. 2: Memory hierarchy of GPU.

optimizations. Candidates for in-place optimization are identified through analysis of both the algorithm and their memory access patterns, using prior information as a guiding factor. Code sections that involve modifying the input data directly, atomic operations, and successive assesses to the same memory address can be potential candidates for in-place operations. Finally, out-of-place operations are identified based on memory load and store operations.

2.1.2 Code Analysis

For the out-of-place operations, we analyze the memory efficiency, storage cost, and calculate the data reuse factors [8], and for the in-place operations, storage and workload size needs to be calculated. The *data reuse factor* of a given array is defined as the ratio of the number of (element) read operations from a copy of an array in a smaller memory, and the number of (element) read operations from the array in the larger memory on the higher hierarchy level [8]. The more often data is reused, the higher the performance benefits of shared memory storage. The *efficiency of memory operations* is determined by two factors: the access pattern (e.g. sequential/regular/random, coalesced/uncoalesced, 1D/2D access pattern) and the hierarchical level of memory where the data is located. The typical GPU memory hierarchy contains multiple modules as shown in Fig. 2, each with distinct bandwidth and access latency characteristics (see Table 1 for the global and shared memory bandwidth comparison).

2.1.3 Optimization decisions

Mapping onto the GPU memory hierarchy For out-of-place operations, the first step involves improving the efficiency of memory operations. If inefficient access patterns, such as unaligned access, uncoalesced access, 2D spatial or strided access, and random access are identified, the memory should be mapped to at least one level down in the hierarchy, so that the inefficient access is handled in the lower levels of the hierarchy. Warps[2] are traditionally not part of the memory hierarchy of the GPU. However, due to existence of warp-level primitives such as voting and shuffling, which are performed on the registers in a SIMD fashion, in our work, we view them as a *separate level of the memory hierarchy* (see in Fig. 2). However, this shared warp memory is limited by three factors: 1) number of threads that have access (limited to 32), 2) limited amount of GPU kernel registers, and 3) the data access pattern needs to be known at compile-time.

Mapping to the lower levels of hierarchy is more straightforward when the array sizes are small. Although the input data often exceeds the capacity of the lower level of hierarchy, it's worth noting that the intermediate results of parallel computations, like the final stages of a parallel reduction algorithm, are typically small enough to fit within these lower level memories. When the memory can not be mapped or is already mapped, careful access pattern analysis are performed to increase bandwidth, for example by avoiding shared memory bank conflicts or enabling vectorized access.

Prioritization of candidates Because of the limit on the amount of available registers and shared memory, it is important to prioritize candidates that can make the most use of the faster memory. Although optimal prioritization requires careful analysis which can easily lead to a complicated optimization problem, we adopt a pragmatic approach in which we assign a *fixed* level of the memory hierarchy to each array based on the data reuse factor. For the same data reuse factor, we prioritize them based on memory access pattern, with 2D spatial access being the highest priority, followed by random access, strided access, and finally coalesced access, in order of decreasing importance. The highest priority memory accesses are mapped to the warps and shared memory while, for the lowest priority, we do not perform any additional mapping. Later optimization can start from the initial solution we obtain. When the access pattern cannot be determined during compile-time, it is regarded as random access.

When out-of-place operations are identified due to the data reuse or sharing opportunities, they get higher priority for mapping even if they already access the memory efficiently. Short arrays, small enough to be process with in a thread-block, are mapped onto the warps when the access pattern is known at compile-time. While larger arrays are best communicated using shared memory. Both of these communication operations have their own synchronization primitives (thread and warp-based barriers), which in their turn add to the execution time (and latency) of the kernel. In this sense, it is useful to *re-order* operations such that the number of synchronization steps is minimized. When the operation can be performed in tiles, memory can be mapped to multiple levels of hierarchy.

Data reuse considerations under resource limitations If the data reuse candidate is not small enough to be mapped to the

[2] Warp refers to a group of threads (typically 32), which execute the same instruction simultaneously on a single SM.

warp memory or if there is not enough shared memory for all the data reuse candidates, then a combination of warp-level primitives and shared memory operations can be used. Hence, there are five choices: 1) mapping onto warp shuffling, 2) mapping onto shared memory, 3) mapping onto combination of both, 4) not exploiting the reuse opportunity, or 5) using re-computation rather than storage. The last choice is suited for simple computations that depend on values already stored in the registers. Lastly, if a combination is not possible, the candidate will need to be mapped (at least) one level higher in the hierarchy, depending on its prioritization.

In-place operations Computations of histograms, statistics, cumulative operations, and propagation algorithms are, perhaps surprisingly, not examples of data reuse [8, p. 188]. In fact, these operations are *in-place* operations that require a different treatment, consisting of 1) data layout optimization and 2) mapping onto the desired level of the memory hierarchy [8, p. 195]. Finding the desired levels is then done as follows: histogram computation can be performed on global memory (which requires atomic operations, but this generally causes performance issues [10]), in shared memory (again with atomic operations, but these are now considerably faster) or using warp voting techniques (e.g., for small histograms). Similarly, for computation of statistics, algorithms based on parallel reduction are common. These algorithms can be mapped onto a combination of shared memory with warp shuffling operations [9, p. 263]. By combining shared memory with warp shuffling, the amount of required shared memory for this operation, can be even reduced by a factor given by the warp size (typically, 32). Hence, for in-place operations we try to perform operation within warps when the workload and the memory size allows. If not, we use combination of shared memory and warp-shuffling operations.

Optimization of expensive numerical computations To optimize expensive numerical computations, faster versions of instructions can be adopted by means of specialized instructions or approximated computation. Two common examples are approximate sine/cosine functions which are available in the NVCC compiler via the `--use_fast_math` flag (or via special intrinsic functions), and the use of Tensor Cores for MMA operations. In many mathematical operations, it is also possible to pre-compute some parts of the calculations, which is especially useful when the same calculations need to be performed multiple times with different inputs.

2.2 Inter-Kernel Optimizations

A straightforward mapping of a modular implementation (e.g., in C) consists of mapping each function individually, which leads to multiple GPU kernels. Although this is useful from a code reuse and generality point of view, it is not ideal in terms of performance. Namely, executing multiple kernels involve launch overheads and costly synchronizations in addition to the fact that data locality is not well exploited. In applications with ultra-high frame rates and small frame sizes, these issues becomes critical as the kernel launch overheads can be comparable to kernel execution times.

In such scenarios, the use of kernel fusion and CUDA graphs can be effective [11, 21]. While CUDA graphs are a run-time solution that only reduces the kernel launch overheads, kernel fusion is a compile-time solution that addresses multiple speed issues by combining multiple kernels into a single large kernel. Applications with high frame rates and small frame sizes offer opportunities for both temporal and spatial parallelism, which can be efficiently mapped onto the GPU architecture through spatio-temporal mapping. This allows for a range of degrees of freedom that can be exploited by using a kernel fusion approach.

Spatio-temporal kernel fusion Based on the flexibility involved in the space-time mapping of the algorithm on a GPU, we consider two types of kernel fusion: *temporal* and *spatial*.

– *Spatial kernel fusion* is a static task scheduling technique that allows to exploit task-level parallelism by assigning parallel kernels to available SMs. A simple technique to implement this scheduling on a GPU consists in determining which task to perform based on the thread-block index. This is practically achieved by fusing the kernels and incorporating the corresponding control flow.
– *Temporal kernel fusion* fuses multiple kernels into a single kernel in a sequential way. If the fused kernels do not have any data dependency or at least not in some parts, then temporal parallelism can be exploited by leveraging instruction-level parallelism as well. But if the fused kernels have no operations that can be executed in parallel, synchronization barriers have to be used which will make the execution of kernels fully sequential. Regardless of whether temporal parallelism is exploited or not, temporal fusion generally improves the kernel launch overhead, data locality, shared memory utilization, and cache efficiency by maximizing data reuse and minimizing costly global memory operations.

Spatial kernel fusion combined with temporal kernel fusion can further improve throughput and latency of applications. The decision to fuse kernels spatially or temporally is typically independent of each other. However, resource limitations, execution times, data dependencies, available parallelism, and synchronization barrier requirements narrow down the feasible fusion schemes.

Our approach for kernel fusion starts by investigating kernels for which the work dimensions are around the same order of magnitude as the size of a thread-block. Luckily, in applications with a small frame size, this is common. For these kernels, block-stride loops can be used to schedule work items into a single thread-block by assigning more work items to each thread. This approach can even lead to an increase in data reuse factor due to the opportunity to copy data before the block-stride loop. Kernels with a single thread-block are a perfect match for temporal kernel fusion. Moreover, fusion of such kernels leaves other SMs unused. Thus, spatial kernel fusion can utilize the unused SMs for parallel task execution, provided there is no data dependency between the kernels.

Temporal fusion of single thread-block kernels is the next step, so that all such kernels are fused into a single kernel, while respecting the data dependency constraints. Spatial kernel fusion needs an estimation about the execution time of kernels that can be run in parallel. When execution times are in the same range, available SMs can be assigned to the parallel kernels. If the execution times are not in the same range, kernels can be fused temporally so that the execution time gets into the same range. If all the spatially fused kernels depend on a single kernel with a small work dimension, this kernel can be repeated for each parallel branches.

Kernel fusion often opens up opportunities for further intra-kernel optimizations by creating more data reuse and sharing opportunities. In addition, kernel fusion may require the implementation of more efficient intra-kernel optimizations that result in significant speed improvements at a higher level. Therefore, the proposed GPU acceleration workflow is an iterative approach and each inter-kernel optimization is always followed by an intra-kernel optimization.

Grid-level synchronization Kernels that are not fused due to grid-level synchronization requirements, whether caused by data dependencies or large work dimensions that cannot be handled within a single thread-block, can still benefit through kernel fusion by utilizing barrier-based inter-block communication for grid-level synchronizations [31]. Therefore, after applying the temporal and spatial kernel fusions, we fuses all the remaining kernels into a single mega-kernel using the barrier-based grid-level synchronization. Barrier based grid-level synchronization, combined with the use of grid-stride loops, also allows for the fusion of kernels that have very large work dimensions, larger than the number of threads that can be run in parallel by the GPU. This is particularly useful for video processing algorithms.

Kernel-IO interactions Using pinned host memory for data transfers between host and device memory at the input and output stages reduces CPU interactions, which is especially advantageous for embedded GPUs as the system memory on these devices is physically shared between CPU and GPU. Pinned host memory is not swapped out to disk by the operating system, which allows fast data transfers between CPU and GPU [1]. However, it must be taken into account that L2 caching is not available for this type of memory and therefore, memory access on this region gets higher priority for intra-kernel memory-based optimizations. Furthermore, if both the GPU and the IO device support the GPUDirect feature, CPU interactions can be eliminated by moving data directly from IO device to the GPU memory.

Avoiding conditional kernel launches It is important to avoid conditional kernel launching, particularly when it requires GPU-CPU synchronization. One common example of conditional launching is the use of termination condition checks. In such cases, it is recommended to launch the kernel in a loop with a fixed number of iterations and check the end condition in an outer loop.

Persistent GPU kernels Finally, fusing the entire algorithm into a single GPU kernel, by using all the aforementioned techniques, offers a significant advantage of utilizing persistent or semi-persistent GPU kernels. With only a single kernel launch, the persistent kernel allows iterative execution of the entire algorithm within the kernel [14].

2.3 Dynamic Pruning of Redundant Computations

Algorithms are often designed with generality in mind. When specialized to a certain problem domain, they may end up performing redundant computations. For example, a connected component labeling (CCL) algorithm based on equivalence labeling [15] may allocate a large number of new labels that need to be merged during region merging, even if it is known that there are only two objects in the image. A trivial optimization consists in eliminating such redundant calculations without introducing inaccuracies in the final result. Computation pruning can be achieved by exploiting prior and domain information through various methods such as static and dynamic video frame cropping, dropping worthless frames, and performing low-cost pre-processing to provide estimations in favor of pruning costly computations.

In this work, our consideration of prior knowledge is focused on a specific type: the selection of a region of interest (ROI) within the image that can completely determine the resulting processing outcomes. A trivial pruning technique then consists in cropping the images to select only the considered ROI. In *static* cropping, a fixed ROI is selected based on the prior experiments and data analysis. While in *dynamic* cropping, the center point and region size are adjusted based on both the content of each frame and the interest of each computation stage. Cropping techniques provide a significant advantage in terms of mapping onto the memory hierarchy of a GPU as the cropped image may fit entirely within a fast level of memory. For both the static and dynamic cropping, memory alignment should be taken into account for selecting the starting address of the cropped region to increase the

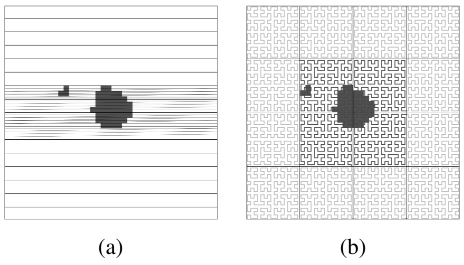


Fig. 3: Warp-based spatial region assignment in dynamic cropping: a) row-based, b) block-based.

memory bandwidth. Besides alignment, region size adjustment is best to be performed based on the available shared memory. For loading data, vectorized memory access should also be considered, particularly when the number of threads are less than the number of pixels in the cropped region. Dynamic cropping with a variable region size can be implemented efficiently by warp-based spatial region assignment (see Fig. 3). In this way, uninteresting regions can be ignored by not participating in the assigned warp in further computations. Note that warps can be efficiently excluded from computations by conditional statements while thread-based conditional execution results in thread divergence.

Similarly, a temporal selection of frames can be pruned by excluding frames that are not useful. This technique is most effective if the frames to be skipped are identified at an early stage of computation or even before loading into the desired memory. For example, by exploiting extra sources of (smaller sized or lower dimensional) auxiliary data if available. If the use of auxiliary data affects the control flow, it may be necessary to read this data from within a kernel to avoid the overhead and increased CPU-GPU interactions associated with conditional launching of kernel functions. Alternatively, in cases where the control flow is frequently impacted, dynamic load balancing techniques should be considered (e.g., run-time warp/block level based techniques [26]).

By exploiting prior information or performing low-cost pre-processing, it might be possible to provide estimations in favor of pruning computations. For example, a target object's position can be estimated based on its prior exact position or prior probability distribution. Then, in addition to simplifying subsequent image processing tasks, embedding these estimations into the image can further simplify later analysis.

2.4 Trading Latency for Throughput

Multi-frame processing The small frame size and the optimizations that are applied to schedule work items into a single thread-block, may result in under-utilized GPU resources. For applications that require higher throughput, required throughput can be obtained by utilizing the unused resources to process multiple frames in parallel, at the cost of increased buffering latency.

Multi-frame processing introduces an additional dimension to the input data, which can impact both intra- and inter-kernel optimizations. Kernels can be influenced differently by the additional dimension based on the assigned work-group and the available resources. Decisions related to the fusion require more attention to ensure that (a) enough resources are available for multi-frame processing, and (b) a balanced number of thread-blocks/SMs are used for the kernels separated with grid-level synchronization. When resources are limited, decisions need to be adjusted to prioritize temporal fusion over spatial fusion, particularly when data reuse opportunities exists. Assuming tasks A and B, with respective execution times A_{time} and B_{time}, temporal fusion is prioritized over spatial fusion when $(A_{time} + B_{time} - DR_{time}) < 2\max(A_{time}, B_{time})$, where DR_{time} represents the time saved by exploiting data reuse through temporal fusion. Accordingly, if a kernel, that runs on multiple thread-blocks, can benefit from data reuse, it needs to be implemented within a single thread-block using block-stride loops.

Variable batch-size Due to the dynamic computation pruning, it is beneficial to use a variable batch size. In this case, GPU kernels process all the available frames without waiting for a fixed number of frames. Variable batch size may require variable work-group size. However, variable kernel configurations should be avoided, as this results in more run-time overheads. Rather, a fixed kernel configuration with grid/block stride loops should be used, so that unneeded warps and thread-blocks can be unloaded at run-time, depending on the batch size. While dynamic optimization decisions are typically required for variable batch sizes, we base our decisions on the most likely number of frames. The optimization decisions and results obtained from processing a single frame serve as the initial solution and basis for determining the most probable number of frames for multi-frame processing.

3 Accelerating the LPBF Monitoring Application

To evaluate our optimization workflow, we start from an existing LPBF video analysis algorithm [7]. An overview of the algorithms and the hardware-software platform utilized for this evaluation can be found in sections 3.1 and 3.2, respectively. Section 3.3 explains how the proposed workflow, explained in section 2, is applied to this application, and finally, utilization of the multi-frame processing for the LPBF monitoring is explained briefly in section 3.4.

From an implementation perspective, different versions of the code have been made for each optimization iteration, each of which add incremental optimizations upon the previous ones. Although these different versions are very useful for benchmarking (e.g., on future GPU architectures), this

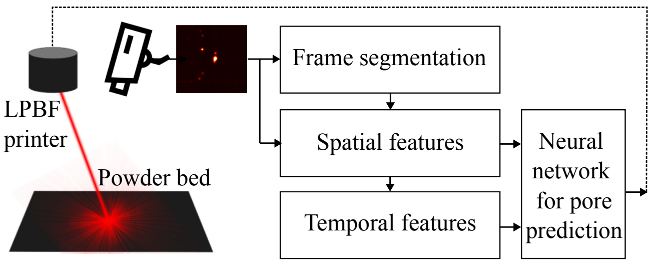


Fig. 4: Building blocks of the LPBF monitoring system.

adds extra work in case the initial requirements of the algorithm (to which the optimizations are applied) changes. However, since this is based on a systematic approach and all these versions follow the same workflow, it is possible to have these steps integrated into an automatic domain-specific language compiler, possibly with special user directives to steer the optimizations, to avoid the required excessive manual work.

3.1 LPBF Video Analysis

Figure 4 presents an overview of the LPBF monitoring system [7]. The analysis starts by reading in a video frame and segmenting it into the melt pool (the molten metal spot created by the laser), spatters (molten metal droplet expelled from the melt pool), and background image regions. The segmentation is performed by static thresholding and CCL to label the melt pool as being separate from the spatters. The segmented regions are then used to extract physical features, including melt pool area, spatter amount and number, melt pool width-length ratio, and average image intensity of the melt pool. Additionally, six polar angle features are computed to capture the direction of spatters leaving the melt pool, and the Histogram of Oriented Gradients (HOG) is used to capture image texture using edge orientations. Temporal variance features are also computed from the spatial features over a chosen time period to estimate melt pool stability, resulting in a total of 40 features. These features are then fed into a neural network for the prediction of lack-of-fusion pores and keyhole pores. While this LPBF monitoring system has shown effectiveness in predicting pores in 3D printed samples, its speed needs to be improved for real-time use.

3.2 The Hardware-Software Platform

The hardware setup includes an LPBF machine, a high-speed camera, a GPU board, and an interfacing board that connects these three components together. The printer is a 3D Systems ProX DMP320 LPBF machine that is controlled by the Materialise Control Platform (MCP). The camera is Luxima LUX2810 which can operate at high frame rates by reading out a subset of the 2.8 megapixels on the sensor. In this application, the sensor produces 8-bit monochrome images at a resolution of 96×96 pixels at 20,000 frames per second. The interfacing board synchronizes the video frames and machine signals received, before sending them to the GPU over a 10 Gb Ethernet connection. The machine signals are a vector of 32-bit integers which include: the state of the laser (on/off), the frame counter, the (x,y) position of laser. The GPU board is a Jetson Xavier AGX which is equipped with an ASUS 10 Gb Ethernet card through its PCIe interface. The prediction results are then sent back to the MCP via the interfacing board over the the same Ethernet connection.

The NVIDIA Jetson Xavier AGX development kit has an 8-core CPU and a GV10B GPU with 512 CUDA cores. These CUDA cores are distributed among 8 SMs for 64 CUDA cores/SM. The CPU and GPU share the same physical memory but have their own caches. Besides the CUDA cores, the SMs are equipped with 8 Tensor Cores. Tensor Cores are capable of performing 4×4 matrix-multiply-and-accumulate operations in one GPU clock cycle. In addition to the Jetson Xavier AGX, we also evaluated the proposed approach on NVIDIA RTX A6000 GPU. The RTX A6000 has 84 SMs equipped with 10752 CUDA cores and 336 Tensor Cores in total. While the Jetson board has an integrated low power 8-core CPU, a high performance Intel Core i7-9800X CPU is used for the RTX A6000. The Jetson board consumes 30W at maximum power configuration while the system that includes the A6000 consumes more than 500W at maximum performance. Finally, both GPUs support GPUDirect RDMA feature which allows direct transferring of input frames to the GPU memory space.

The software platform consists of a main program written in Quasar that performs the video analysis, and a C++ library that performs low-level, driver-related tasks.

3.3 Utilizing the Workflow

3.3.1 Exploiting Signal Data to Prune Computations

Dropping Empty Frames Because the camera captures frames regardless of whether the laser is on or off, there will be empty frames in the frame group. By using machine signals, it is possible to identify and ignore these empty frames to avoid unnecessary computations. The occurrence and frequency of empty frames depend on the object's geometries during manufacturing and can be quite frequent. From an implementation perspective, a new kernel is proposed prior to the segmentation stage to identify them before loading frames.

Exploiting the Melt Pool Position By having synchronized video frames and printer signals, we are able to further exploit signal data to prune computations. We can use the x-y position of the laser to calibrate the laser position in the machine to the laser position in the pixel array of the camera [7]. This allows us to perform dynamic cropping by estimating the melt pool position *early on* using the laser x-y coordinates.

Based on [7], a region of 40×40 pixels around melt pool has been shown to have sufficient information for predicting pores. Therefore, a dynamic ROI cropping, with the melt pool at the center, is performed to make the frames smaller. To improve memory bandwidth, we adjust the estimated center point so that the start address of cropping region has the correct alignment for a vectorized memory load. To implement non-fixed window size cropping, threads of each warp communicate to determine if the warp has any non-zero data. Empty warps do not participate in further computations.

Prior information of melt pool position can also simplify the spatial features extraction as well. A pixel search for the melt pool (by means of the `find_index_max` function) is no longer needed. The melt pool position can be embedded into the label image by assigning a specific label to the melt pool during segmentation. By doing so, the index of the bin in CCL histogram corresponding to the melt pool is known. This prior information simplifies all the melt pool related computations: computations for average melt pool intensity and the melt pool width-length ratio do not depend on histogram computations and can be performed along with the histogram computations. Once the histogram is computed, the first bin gives the melt pool area and the other non-zero bins indicates the number of spatter and their amount.

3.3.2 *Intra-Kernel optimizations*

While computation pruning needs to be adapted to the application, the intra-kernel optimizations are straightforward to apply. Each task depicted in Fig. 5, is checked to identify the most suited candidates for the optimizations. Presented in Fig. 5, are the tasks required for the LPBF monitoring, separated based on the functionality. Below, we will discuss a few of these candidates and corresponding decisions:

- Identifying the index of non-empty frames can be performed by a cumulative sum over the laser status signal array. Assuming the number of available frames to be less than 8 for each kernel launch, this is mapped to warp-level operations within a single warp.
- Multi-scan label-equivalence based connected component labeling (MSLE-CCL) algorithms use two stages [15]: 1) initial unique label assignment, 2) minimum label propagation. Assigning a unique label for the CCL algorithm which can be performed by a cumulative sum over the non-zero pixels, is mapped to a combination of warp shuffling and shared memory operations. Every warp performs a cumulative sum using warp-level primitives for few rows of the frame. Once done, the last thread of each warp stores the partial sum to the shared memory, and then a single warp performs cumulative sum over the partial sums. Finally, every warp use these values to assign a unique label to each non-zero pixel.
- The second stage of MSLE-CCL algorithm is the propagation of the minimum pixel labels to the 4-connected neighbouring pixels. For this purpose, each thread checks the neighbouring pixels' value and label. This stage is also mapped to a combination of warp-level and shared memory operations. As presented in 6, we first perform warp-level operations in two directions, horizontally and vertically, in a small block around the melt pool and then shared memory is used to cover the whole frame. Since only a few spatters, which are usually much smaller than the melt pool, can be in the region that was not covered by warp-level operations only a few iterations of this shared memory based approach need to be applied. Furthermore, only the non-empty warps are involved in this operation thanks to the dynamic ROI cropping.
- To sum intensity within polar angle zones, a block-stride loop is used to cover the entire zone. The sum operation

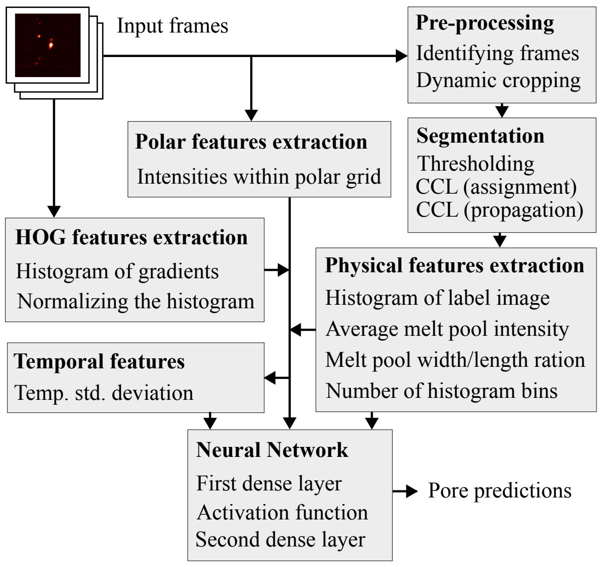


Fig. 5: Flow chart of the LPBF video analysis.

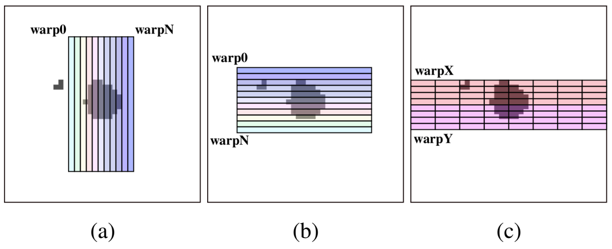


Fig. 6: Different steps of minimum label propagation: a) warp-shuffling in vertical direction; a) warp-shuffling in horizontal direction; and c) based on shared memory.

is mapped to warp-level primitives, and shared memory is used to sum up the result of each warp.

- While Tensor Cores have the potential to enhance performance for MMA operations, none of the MMA operations in this case are currently assigned to the Tensor Cores. The sizes of the matrices being multiplied, $[1,40] \times [40,164]$ and $[1,164] \times [164,2]$, are too small for the available fragment sizes.
- The segmentation and physical features extraction kernels have a data reuse factor of one for the frame data, which means that the matrix is not mapped to higher levels of memory. However, when these two kernels are fused during later inter-kernel optimizations, the data reuse factor for the region of the melt pool will increase, which justifies the need to map to the shared memory.

3.3.3 Inter-Kernel optimizations

Kernel fusion starts by fusing all the kernels within in each block of Fig. 5. Temporal fusion involves re-implementing kernels with the same kernel configuration and merging them into a single kernel. When a data dependency exists, synchronizations are required in between.

Initial fusion results in seven kernels (Pr = Pre-processing, Se = Segmentation, Ph = Physical, Po = Polar, Ho = HOG, Te = Temporal, and Ne = Neural net.), where all operate using a single thread-block, except the *Po kernel* which requires six thread-blocks. After initial fusion, the intra-kernel optimization is applied once again to identify and optimize the newly introduced opportunities.

For the second round of kernel fusion, we evaluate whether it is feasible to combine these seven kernels. Based on data dependency, *Se* and *Ph* can be fused into *Se-Ph*, and *Te* and *Ne* can be fused into *Te-Ne*. Although *Se-Ph*, *Po*, and *Ho* can run in parallel, execution times of these kernels justifies temporal fusion of *Po* with *Ho*, and then spatial fusion of *Se-Ph* with *Po-Ho*. As there is no dependency between *Po* and *Ho* kernels, no synchronization is required in between which allows temporal parallelism to be leveraged as well. Finally, the *Pr* kernel is repeated for both the *Se-Ph* and *Po-Ho* to reduce the number of kernels to two. Fig. 7 illustrated these steps. Further optimization is available by use of barrier-based grid-level synchronization that fuses these two remaining kernels into a single kernel.

IO-Kernel Interaction To feed the input frames, circular buffers are considered in a pinned memory region. While driver fills the buffers with all the available data it receives, the kernels can access the buffers. A similar approach is considered for the output. The utilization of the kernel fusion technique along with intra-kernel optimizations diminish the impact of L2 cache unavailability as the input data is loaded and mapped to the shared memory just once.

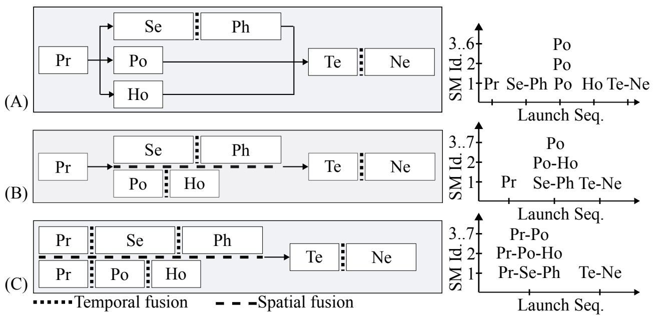


Fig. 7: The proposed kernel fusion steps.

Persistent GPU kernel Finally, having the whole algorithm fused into a single GPU kernel along with the zero-copy implementation for the input-output data, allows implementation of a persistent GPU kernel. The resulting persistent kernel is launched once and iterates the algorithm using a loop inside the kernel to continuously process the input data.

3.4 Multi-frame processing

The kernels preceding grid-level synchronization employ a larger number of thread-blocks to process each frame, which results in resource imbalance between the two sides of grid-level synchronization. To address this, the *Po* kernel, which runs with six thread-blocks, is implemented using a block-stride loop within a single thread-block. Since execution times of *Pr-Po-Ho* and *Pr-Se-Ph* are in the same, we do not prioritize temporal fusion over spatial fusion. Finally, grid-stride loops are used for spatial fusion of parallel instantiation of kernels for processing frames in parallel.

4 Results and discussion

This section presents the evaluation results for the LPBF application. The evaluation was carried out by measuring the performance during 6 layers of the printing process, where each layer consisting of an average of 8,500 frames, with approximately 10% of frames being empty. All experiments were conducted on two GPUs, the Jetson Xavier AGX and RTX A6000. The results are measured using timers via the tic/toc API in Quasar. To evaluate the approach for throughput higher than the capabilities of camera, pre-captured frames are transferred to the input buffer beforehand.

Table 2 presets the average measured execution time per frame for the various steps of the optimization workflow. Dynamic computation pruning is the first optimization that is applied, which resulted in a gain of ∼3 times in performance of Jetson board. Then, initial intra-kernel optimizations led to more than 2 times speed-up in execution time of Jetson Xavier. Since the initial implementation is already a GPU

Table 2: Measured execution time per frame.

Optimization	Jetson Xavier	RTX A6000
Initial implementation [7]	7 ms	3.5 ms
+Computation pruning	2.5 ms	600 µs
+Intra-kernel optimizations	900 us	310 µs
+Initial kernel fusion	350 µs	130 µs
+Unconditional kernel launch	280 µs	120 µs
+Kernel fusion (Fig. 7(a))	200 µs	88 µs
+Kernel fusion (Fig. 7(b))	126 µs	57 µs
+Kernel fusion (Fig. 7(c))	95 µs	42 µs
+Grid-level synchronization	63 µs	31 µs
+Persistent kernel	45 µs	18 µs

accelerated implementation (namely, by means of the Quasar compiler), the speed-up achieved at this point can be considered as a significant accomplishment.

The following optimizations are inter-kernel optimizations, each followed by an intra-kernel optimization. Initial kernel fusion, which results in seven kernels as illustrated in Fig. 5, provides $\sim$3 times speed-up on both GPUs.

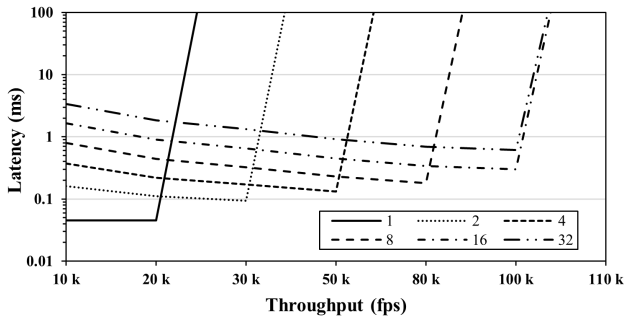
To demonstrate how conditional kernel launch can affect the performance, a termination condition was initially employed based on the number of frames processed by the GPU. In the following step, this condition was removed, and the kernel was launched in a loop with a fixed number of iterations, and condition was then checked in an outer loop. Since launching kernels is a task executed by the CPU, the resulting overhead varies in proportion to the CPU's performance. Thus, the speed-up achieved by this optimization for the Jetson board, which has a low-power/low-performance CPU, is higher than the RTX A6000 which uses a high-performance CPU.

Subsequently, three extra kernel fusions, according to the steps presented in Fig. 7, are applied, which leads to more than a 3x speed-up. Final optimizations are based on the grid-level synchronization. Fusing the two remaining kernels into a single kernel results in 63 µs for processing each frame, and finally, the implementing the entire algorithm as a single persistent kernel allows processing each frame in 45 µs on the Jetson Xavier and 18 µs on RTX A6000. Overall, these optimization could speed-up the initial implementation by up to 155 times on the Jetson board and by up to 194 times on the RTX A6000 GPU.

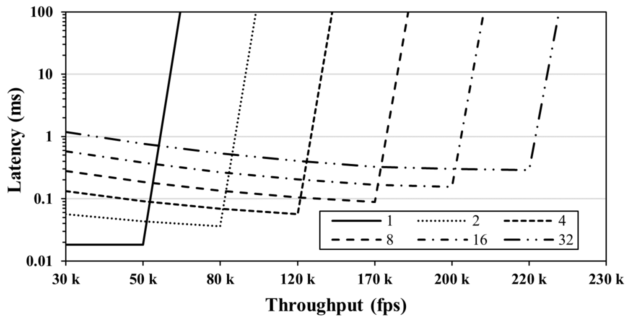
Trading latency The next experiments focus on processing multiple frames in parallel. Table 3 shows the maximum throughput achieved by multi frames processing. To eliminate the buffering latency, we assume that all frames are already present in the input buffer. Based on the results, doubling the number of frames lead to $\sim$ 1.5 times improvement in throughput when there are sufficient resources available. On the Jetson, throughput improvements become saturated when processing more than 16 frames in parallel, while for the RTX A6000, the saturation occurs in higher number of frames because of having more SMs (see Table 1).

Table 3: Maximum achievable throughput (fps) by multi-frame processing.

Parallel frames	**Jetson Xavier**	**RTX A6000**
1	22k	55k
2	33k (1.5x)	85k (1.5x)
4	56k (2.5x)	125k (2.3x)
8	87k (4.0x)	170k (3.1x)
16	106k (4.8x)	200k (3.6x)
32	106k (4.8x)	222k (4.0x)



(a)



(b)

Fig. 8: Latency-throughput curves for different number of frames processed in parallel; a) Xavier AGX; b) RTX A6000.

Finally, Fig. 8 presents the latency-throughput curves for the different number of frames processed in parallel, where both the computation and the buffering latency are considered. Each curve begins with a low-load latency and ends in saturation throughput, demonstrating the trade-off between latency and throughput. The inflection points indicate the configuration where the latency is minimized while achieving the maximum possible throughput. As an example, for a throughput of 50 $kfps$, the optimal configuration is to process 4 frames in parallel when using the Jetson Xavier.

5 Conclusion

This work proposed an optimization workflow for real-time execution of applications with ultra-high frame rates and small frame sizes on GPUs by focusing on computation pruning, and intra- and inter-kernel optimizations. The proposed approach is evaluated for a real-time LPBF machine using the NVIDIA Jetson Xavier AGX and RTX A6000, which led to throughput of 22 $kfps$ with computation latency of 45 µs using single-frame processing on the Jetson board, and 55 $kfps$ with the latency of 18 µs on the RTX A6000.

By allowing parallel multi-frame processing, low occupancy issues of GPU multiprocessors can be mitigated at the cost of increased latency. The joint analysis of throughput and latency in parallel multi-frame processing interestingly leads to curves containing inflection points which highlight suitable local trade-offs that minimize latency for a given maximal frame rate.

Acknowledgements

This work is financially supported by the VLAIO ICON project 'Vision in the Loop' (HBC.2019.2808), a collaboration between imec, Flanders Make, Materialise, Dekimo, ESMA and AdditiveLab. The RTX A6000 GPU used for this research was donated by the NVIDIA Corporation.

References

1. (2023) CUDA C++ Programming Guide. Accessed: 2023 June 13
2. (2023) GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, accessed: 2023 May 28
3. Abe F, Osakada K, Shiomi M, Uematsu K, Matsumoto M (2001) The manufacturing of hard tools from metallic powders by selective laser melting. Journal of materials processing technology 111(1-3):210–213
4. Adnan AM, Radhakrishnan S, Karabuk S (2015) Efficient Kernel Fusion Techniques for Massive Video Data Analysis on GPGPUs. arXiv preprint arXiv:150904394
5. Adnan M, Lu Y, Jones A, Cheng FT (2021) Application of the fog computing paradigm to additive manufacturing process monitoring and control. IEEE Transactions on Multimedia 21(6)
6. Allen T (2018) Improving real-time performance with CUDA persistent threads (CuPer) on the Jetson TX2. Concurrent Real-Time
7. Booth B, Heylen R, Nourazar M, Verhees D, Philips W, Bey-Temsamani A (2022) Encoding stability into laser powder bed fusion monitoring using temporal features and pore density modeling. Sensors 22(10):3740
8. Catthoor F, Danckaert K, Brockmeyer E, Kulkarni K, Kjeldsberg PG, Van Achteren T, Omnes T (2002) Data access and storage management for embedded programmable processors. Springer Science & Business Media
9. Cheng J, Grossman M, McKercher T (2014) Professional CUDA c programming. John Wiley & Sons
10. Farber R (2011) CUDA application design and development. Elsevier
11. Filipovič J, Madzin M, Fousek J, Matyska L (2015) Optimizing CUDA code by kernel fusion: application on BLAS. The Journal of Supercomputing 71(10):3934–3957
12. Fürtler J, Bodenstorfer E, Mayer KJ, Brodersen J, Heiss D, Penz H, Eckel C, Gravogl K, Nachtnebel H (2007) High-performance camera module for fast quality inspection in industrial printing applications. In: Machine Vision Applications in Industrial Inspection XV, SPIE, vol 6503, pp 155–166
13. Goossens B, De Vylder J, Philips W (2014) Quasar—A new heterogeneous programming framework for image and video processing algorithms on CPU and GPU. In: 2014 IEEE International Conference on Image Processing (ICIP), IEEE, pp 2183–2185
14. Gupta K, Stuart JA, Owens JD (2012) A study of persistent threads style GPU programming for GPGPU workloads. IEEE
15. He L, Ren X, Gao Q, Zhao X, Yao B, Chao Y (2017) The connected-component labeling problem: A review of state-of-the-art algorithms. Pattern Recognition 70:25–43
16. Kubík P, Šebek F, Krejčí P, Brabec M, Tippner J, Dvořáček O, Lechowicz D, Frybort S (2023) Linear woodcutting of European beech: experiments and computations. Wood Science and Technology 57(1):51–74
17. Li A, Zheng B, Pekhimenko G, Long F (2022) Automatic horizontal fusion for GPU kernels. In: 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, pp 14–27
18. Liu X, Guo Y, Zhang W, Wu D, Huang R, Yang M, Lu B (2022) Dynamic formation characteristics and mechanism of hybrid laser arc welding surface layer by Ni-based filler metal based on rotating laser induction. Journal of Materials Research and Technology 20:3600–3615
19. Membarth R, Reiche O, Hannig F, Teich J, Körner M, Eckert W (2015) Hipa cc: A domain-specific language and compiler for image processing. IEEE Transactions on Parallel and Distributed Systems 27(1):210–224
20. Pratt-Szeliga PC, Fawcett JW, Welch RD (2012) Rootbeer: Seamlessly using gpus from java. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Sys-

tems, IEEE, pp 375–380

21. Qiao B, Özkan MA, Teich J, Hannig F (2020) The best of both worlds: Combining CUDA graph with an image processing DSL. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), IEEE, pp 1–6
22. Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices 48(6):519–530
23. Reinke P, Beckmann T, Ahlers C, Ahlrichs J, Hammou L, Schmidt M (2023) High-Speed Digital Photography of Vapor Cavitation in a Narrow Gap Flow. Fluids 8(2):44
24. Scime L, Fisher B, Beuth J (2018) Using coordinate transforms to improve the utility of a fixed field of view high speed camera for additive manufacturing applications. Manufacturing Letters 15:104–106
25. Sepasgozar SM, Shi A, Yang L, Shirowzhan S, Edwards DJ (2020) Additive manufacturing applications for industry 4.0: A systematic critical review. Buildings 10(12):231
26. Steinberger M, Kenzel M, Boechat P, Kerbl B, Dokter M, Schmalstieg D (2014) Whippletree: Task-based scheduling of dynamic workloads on the GPU. ACM Transactions on Graphics (TOG) 33(6):1–11
27. Truong L, Barik R, Totoni E, Liu H, Markley C, Fox A, Shpeisman T (2016) Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 209–223
28. Varga M, Ventura Cervellón A, Leroch S, Eder S, Rojacz H, Rodríguez Ripoll M (2023) Fundamental abrasive contact at high speeds: Scratch testing in experiment and simulation. Wear 522:204696, 24th International Conference on Wear of Materials
29. Vasilache N, Zinenko O, Theodoridis T, Goyal P, DeVito Z, Moses WS, Verdoolaege S, Adams A, Cohen A (2018) Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. arXiv preprint arXiv:180204730
30. Wienke S, Springer P, Terboven C, an Mey D (2012) OpenACC—first experiences with real-world applications. In: Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18, Springer, pp 859–870
31. Xiao S, Feng Wc (2010) Inter-block GPU communication via fast barrier synchronization. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, pp 1–12
32. Zhang L, Wahib M, Chen P, Meng J, Wang X, Matsuoka S (2022) Persistent Kernels for Iterative Memory-bound GPU Applications. arXiv preprint arXiv:220402064
33. Zou A, Li J, Gill CD, Zhang X (2023) RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization. IEEE Transactions on Parallel and Distributed Systems