Programming and Mapping for Mixed Heterogeneous Devices: The Case of Optical Flow

Sergio Alonso Pascual
Departamento de Informática
Universidad de Valladolid. ETSII
Valladolid, Spain
sergio.alonso.pascual@uva.es

Arturo Gonzalez-Escribano
Departamento de Informática
Universidad de Valladolid. ETSII
Valladolid, Spain
arturo@infor.uva.es

***Abstract*—Current parallel systems are increasingly heterogeneous, mixing devices of different types and computing capabilities. Exploiting multiple different devices for the same application continues to be a challenge that ranges from technical problems related to synchronizing and communicating diverse devices to problems of load distribution and flexibility to adjust the computation to the platform resources. In this work, we study the problem of using and extending a heterogeneous portability layer to program and adapt HSOpticalFlow to heterogeneous platforms. HSOpticalFlow is a streaming application to estimate the apparent movement of objects in a sequence of images. It is a simple but characteristic example of the structure of applications based on multilevel ILS (Iterative Loop Stencil), also known as multi-grid methods, applied to a sequence of inputs. Starting from the original CUDA reference code, we present a methodology and programming techniques based on the Controller programming model to implement it as a pipeline among multiple devices. We discuss a technique to determine a proper work partition and mapping for a set of devices. This allows for building very efficient parallel solutions, using similar devices or taking advantage of devices with lower computing power, to reduce the load and increase the productivity of more powerful ones. We present the results of an experimental study using several GPUs of different vendors, architectures, and generations, showing that this solution allows combinations of devices to be efficiently exploited to improve performance. Specifically, the results include speedups of 1.91x using two NVIDIA A100 GPUs and 1.21x using one NVIDIA V100 GPU and one AMD WX9100 GPU, which is about 3x slower than the NVIDIA GPU for this application.**

***Index Terms*—Heterogeneous programming, Streaming, Parallel pipeline, Optical Flow.**

I. Introduction

It is increasingly common to find parallel systems with high heterogeneity. In both large computing platforms and integrated on-chip systems. Although we have an increasing number of tools and proposals to program and manage heterogeneous devices from code with a high level of abstraction (for example SYCL [1], Controller [2], or OpenH [3]), in many cases they are not yet mature enough to easily exploit the full potential of these systems. There are still issues related to the efficient and transparent movement of data between devices of different architectures, synchronization, overlapping computation with data transfers, and an important lack of load balancing or resource allocation techniques to adjust the computation to the execution platform. In general, current models provide us with general-purpose programming tools, which must be used by the programmer to build their solutions taking into account the structure of the application and the target platform.

Streaming or data-flow programs are an interesting and demanding type of application for exploiting highly heterogeneous systems. These applications are characterized by receiving as input a *stream*, a channel through which data is received, generally as multiple instances of a structure of the same type. For example, the frames of a video stream. These applications perform a series of tasks on each data instance, which often have different granularity or computing needs. Adapting these applications to heterogeneous platforms while taking advantage of their resources can be complex.

We choose the 2D optical flow method known as Hierarchical Horn and Schunck (HSOpticalFlow) [4] as motivation and case study. It is a streaming application to estimate the apparent movement of objects in a sequence of images. It minimizes an energy function using a finite difference approximation of the Euler-Lagrange equation corresponding to this problem. HSOpticalFlow presents a simple but characteristic example of the structure of applications based on multi-level ILS (Iterative Loop Stencil), also known as *multi-grid* methods [5]–[7]. This class encompasses a whole set of scientific applications based on finite elements to solve partial differential equations with iterative methods at various levels of granularity. These applications implement techniques known as V or W cycles, where the flow of the program goes up and down through the granularity levels to achieve greater precision with faster convergence. At each level, the program works with larger and larger data structures at a finer level of detail. Therefore, the workload of the tasks depends on the granularity level they are working on.

In this work, we present the following contributions:

- A programming methodology to build pipeline solutions for multi-grid streaming applications for heterogeneous systems. It exploits the potential overlapping of computations and memory movements that are a distinctive feature of parallel pipelines.
- An implementation of HSOpticalFlow applying this methodology. We use the Controller model as a portability layer to exploit multiple GPUs of the same or different vendors and architectures. We modify the portability layer

to efficiently support specific texture features of this application.
- A technique to systematically determine a proper work partition and mapping for a set of devices.
- An experimental study to show the efficiency of the proposed solution compared to a reference CUDA version and its port to SYCL (both from the official CUDA and oneAPI samples repositories), and our port to OpenCL. The experiments include several scenarios covering combinations of different types of GPU architectures, in three machines with five different types of NVIDIA and AMD GPUs.

The results show speedups of, for example, 1.91x using two NVIDIA A100 GPUs and 1.21x using one NVIDIA V100 GPU and one AMD WX9100 GPU, which is about 3x slower than the NVIDIA GPU for this application. All the codes and result data are available at https://www.dropbox.com/scl/fi/zv5u8d1put3s1kxo6bt6q/controller.zip?rlkey=erymxfdq6ytift15avf6h9ybh&st=zfnctse3&dl=0.

The rest of the work is organized as follows. Section 2 discusses related work. Section 3 presents the proposal and the details of an implementation using HSOpticalFlow as a case study. Section 4 describes the experimental study and discusses its results. Section 5 presents the conclusion and possible future work.

II. Background and related work

The multigrid version of the HSOpticalFlow method [6] improves convergence, especially for large displacements. An example implementation is developed in CUDA and included in the domain-specific examples provided with the development toolkit [8]. There is also a port using SYCL in the examples included in the current Intel oneAPI release [9]. There exist also variants with improvements for situations with object occlusions [10], or for using full-color images [11]. However, these variants do not change the multigrid structure of the application.

Various proposals exist for high-level heterogeneous parallel programming models. These models attempt to simplify the programming of portable applications for different types of devices or combine several of them. Some proposals use a single source code compiled and adapted to different platforms. For example, proposals such as OpenMP [12], Kokkos [13], or the SYCL standard [1], which is becoming consolidated thanks to the evolution of the compilers that implement it, such as AdaptiveCpp [14] or the DPC++ language integrated into Intel oneAPI [15]. Execution mechanisms for heterogeneous tasks that trace dependencies to hide data communications across devices introduce non-negligible overheads (see e.g. [16]). Other less well-known or academic proposals for fast synchronization across different types of devices include, for example, the Controller model [2].

These models provide the necessary tools for general-purpose programming and support multiple heterogeneous devices. However, it is the programmer's responsibility to analyze the structure of the applications and generate the appropriate code to distribute and balance the load, adapting it to the requirements of a system with different types of devices, especially if they have different computing capabilities.

Some models, such as FastFlow [17] or SkePU [18], focus on generating code for applications with different parallel patterns from high-level expressions. Some include patterns based on the pipeline or the streaming structure. However, load distribution and balancing for multilevel pipeline structures on heterogeneous devices of a diverse nature remain problematic.

Some heterogeneous programming models, such as StarPU [19], Sigmoid [20], OpenH [3], or OpenMP extensions [12], include solutions for load distribution and balancing integrated into the model or its execution mechanisms. These generic solutions balance the load by dividing the computation into subtasks and distributing them dynamically on demand with a master-worker or task-farm scheme. These solutions introduce scheduling overheads that are not negligible for small tasks. For example, it is not recommended to use tasks smaller than 500 μ-seconds in StarPU [19]. Applications such as HSOpticalFlow, execute for each pair of frames thousands of kernels at each level, with kernel loads around 4 mu-seconds or 12 μ-seconds in the smaller levels (measured in an NVIDIA A100 GPU). Data migration across devices forced by dynamic scheduling may also introduce unneeded costly data movement overheads for small kernels. Multi-grid streaming applications present pre-established and known execution structures, dependencies, and locality properties that can be better exploited with more static solutions.

III. Proposal

This section describes our proposal of a programming methodology and techniques to exploit multiple heterogeneous devices for streaming applications where the tasks associated with the data-flow instances are the stages of an iterative method, potentially using multiple kernels and iteration levels with different resolutions. The proposed solution can be extrapolated to other multi-grid methods and stream-based programs.

A. Case study

We have chosen the HSOpticalFlow application as a case study and motivation. We use as a reference the implementation included in the domain-specific samples provided with the CUDA development toolkit. The listing 1 shows the pseudocode of the HSOpticalFlow algorithm.

It employs a multilevel strategy, from coarse to fine grain. It first solves the problem for lower resolution versions of the images, and it successively upscales the solution to be used as the starting point in the calculation of the next resolution level. At each level, the algorithm executes several *warp iterations*. Each warp iteration starts with an initial estimate of the solution. It first applies a warp operator on the target image and computes the derivatives. Then, it computes a fixed number of iterations of a Jacobi method, implemented as an ILS (Iterative Loop Stencil) that computes in parallel the new

Listing 1. Pseudocode of the HSOpticalFlow algorithm. Function calls represent kernel launches (except swap). Input parameters are colored in blue, while output ones are red.

```
// Create lower resolution versions of images (src y tgt)
for(lvl = nlvls - 1; lvl > 0; lvl--){
  Downscale(src[lvl], src[lvl-1]);
  Downscale(tgt[lvl], tgt[lvl-1]);
}
// Initial estimate (u, v) starts at 0
for(lvl = 0; lvl < nlvls; lvl++){
  for(warp = 0, warp < nwarps; warp++){
    // Warp target image according to estimate (u, v)
    WarpImage(tgt[lvl], u[lvl], v[lvl], dist[lvl]);
    // Compute matrices of the equation
    ComputeDerivatives(src[lvl], dist[lvl], Ix[lvl], Iy[lvl], Iz[lvl]);
    // Solve equation for du, dv
    for(i = 0; i < nsolves; i++){
      JacobiSolve(du0[lvl], dv0[lvl], Ix[lvl], Iy[lvl], Iz[lvl], du1[lvl], dv1[lvl]);
      swap(du0[lvl], du1[lvl])
      swap(dv0[lvl], dv1[lvl])
    }
    // Update current estimate
    add(u[lvl], du0[lvl], u[lvl]);
    add(v[lvl], dv0[lvl], v[lvl]);
  }
  if(lvl < nlvls - 1){
    // Prolongate solution (u, v) for the next level
    Upscale(u[lvl], u[lvl+1]);
    Upscale(v[lvl], v[lvl+1]);
  }
}
```

value of each matrix element with the previous values of the neighboring elements.

A representation of the task flow for comparing a pair of frames can be seen in figure 1. The original application uses five resolution levels, three warp iterations at each level and 500 Jacobi iterations per warp iteration. The computation begins at the lowest resolution level (level 0), performing the selected number of chained warp iterations. A *prolongation* or *upscale* function is used to generate a higher-resolution result from the output of the last warp iteration of the previous level. This new estimation is the input for the first warp iteration of the next level.

In this application, and for UHD image resolutions or higher, the occupation and use of the resources of a GPU are practically complete. We have experimentally tested that using the same GPU to compute multiple pairs of frames simultaneously, with several processes executing the original reference application, does not increase the throughput. The work is serialized. Thus, we focus on the use of more devices to improve the throughput of the application. In this work, we study the use of multiple devices in a single node.

B. Pipeline structure

The iterative and multi-level structure of HSOpticalFlow, when working on a sequence of images, provides opportunities to distribute the workload in different ways among several devices. The image sequence creates a continuous data flow or *pipeline*. This pipeline structure can be deployed across several devices, distributing the load for each pair of frames. When a device finishes its part, it sends the result to the next device in the pipeline, and it can begin the computation of its part of the pipeline for the next pair of frames. Tasks mapped to

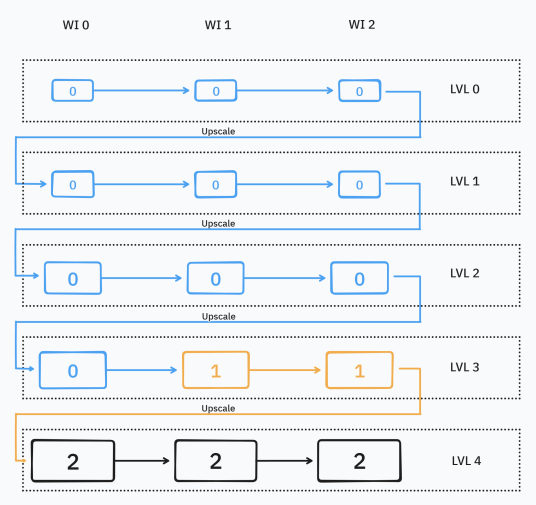


Fig. 1. HSOpticalFlow's flow diagram. Each box represents a warp iteration. The number inside each box represents the identifier of the device executing the warp iteration, in a hypothetical heterogeneous mapping.

different devices can be executed in parallel. The computation and data movements between devices can be overlapped using asynchronous communications. Once the pipeline is full, the latencies of data movements can be partially or completely hidden.

An important decision for this approach is choosing the cut-off points at which the computation of the same pair of images is continued on another device. There are three main options. Each one implies increasing implementation complexity to obtain finer granularity for controlling load balancing between devices.

1) Partition only between levels.
2) Partition between warp iterations at any level.
3) Partition between Jacobi iterations inside a warp iteration.

Our preliminary experimental tests indicate that distributing entire levels between devices of different computing capacities makes the granularity too coarse, preventing the possibility of creating a good load balance in many situations. On the other hand, the complexity of programming a device change between arbitrary Jacobi iterations is high. Thus, for this work, we decided to study the distribution of complete warp iterations (the second option), which proves to be a good balance in terms of implementation complexity and reasonable control of the load on each device. Figure 1 shows an example of load distribution by assigning different warp iterations to each device. Higher-level indexes imply a computation with higher-resolution images, and thus, a higher workload.

We choose a solution to specify the cut points where the computation of a pair of frames continues in another device. In this work, we specify these cut points manually using a function call. An example is shown in figure 2.

Listing 2. Code to specify the warp iterations to be executed on each device. The parameters of the AddCutPoint function indicate the last level and warp iteration executed by a device identifier. The example corresponds to the assignment in figure 1.

```
InitWorkPartition();
// Device-id, Last Level and Warp
AddCutPoint(0, 3, 0);
AddCutPoint(1, 3, 2);
AddCutPoint(2, 4, 2);
```

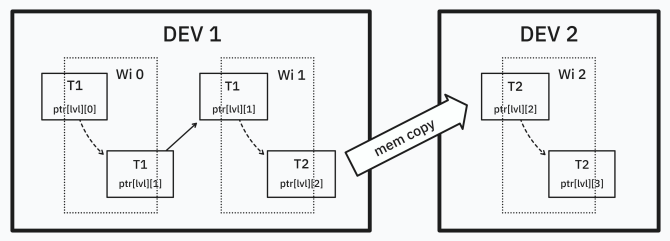


Fig. 2. Diagram of the use of data structures in warp iterations mapped to the same device, or when the next iteration is mapped to another.

C. Data transfers

Figure 2 shows an example of the data structures and movements used to compute several warp iterations across devices. The figure shows three warp iterations of the same level. W_i indicates the computation of the i-th warp iteration. T_j indicates the data structure j. The T_j structures on the top-left side of the box representing each computation are the input, and the ones in the bottom-right are the output. We use the same data structure as input and output for warp iterations of the same level mapped to the same device. When two consecutive warp iterations are mapped to different devices, the output data of the first warp iteration should be moved to another data structure to be used as input for the next warp iteration in the next device. For a level change within the same device, the output structure should be different because the size used on the higher resolution level is different. An upscale operation is also needed to generate the input of the next level.

In this application, the computations on each level use the same logical data structures with different resolutions (see listing 1). The structures for each level can be allocated separately, or the implementation can reuse the higher resolution ones on each level, using only part of it. To simplify the programming of the pipeline, we define an array of pointers for each data structure. Each one points to a specifically allocated data structure for that level, or to the highest resolution one to reuse it. It is a program design decision with no impact on the rest of the discussion. Common data structures are privately allocated on all devices, storing the pointer associated with each device in a two-dimensional array indexed by levels and device identifiers. Warp iterations are executed on specific devices determined by the pipeline structure. The results of a warp iteration could need to be moved across devices if the data structures between warps or levels are allocated on different devices. We also extend the idea to arrays with pointers indexed by level and warp iteration. The data structure for each level and warp is allocated on the proper device, storing the pointer in as many array positions as needed to reuse it in as many warp iterations as possible. Using these pointer arrays when coding the pipeline makes the device mapping transparent. In figure 2 we include in the computation boxes the reference of the pointer arrays ptr, for the corresponding level and warp iteration.

Listing 3. Pseudocode for processing several frames. In blue are the input arguments and in red are the output ones.

```
tgt = loadFrame( 0 );
for(i = 1; i < nFrames; i++){
    src = tgt;
    tgt = loadFrame( i );
    ComputeFlow(src, tgt, u, v, ...);
    Norm(u, v);
}
```

D. Controller implementation

We have chosen the Controller programming model to build an experimental prototype that can easily work with multiple heterogeneous devices. This model offers a mechanism to directly integrate kernels written in the lowest-level programming models supplied by the device vendor, such as CUDA, OpenCL, Hip for GPUs, and OpenMP for multicore CPUs. It includes a system to detect dependencies between tasks at runtime and transparently perform the memory transfers between devices needed to keep consistency. Its execution system includes a very efficient mechanism for managing synchronizations and memory transfers between devices of different natures. In this section, we describe the improvements needed and the advantages of using it to simplify the pipeline implementation.

1) Preparation: We use the original HSOpticalFlow code provided by NVIDIA in the CUDA development toolkit samples as a starting point. The reference version extensively uses single-channel textures due to its benefits related to (1) The mirror addressing mode that solves problems in the calculation of derivatives and accesses in the contours; (2) The bilinear interpolation used in constraint, extension, and warping operations; and (3) to improve data locality thanks to texture caches [21]. We have incorporated functionalities into the Controller programming model to work with this type of memory.

The original code is prepared to run with only a pair of input frames. To have a complete streaming application that works on a sequence of images, we add an external loop to read and process frames (see the listing 3). To more accurately measure the working times of the main algorithm, instead of writing the results to a file, we calculate the norm of the result of each pair of frames. This also allows checking the correctness of any version derivated from this code with a good level of confidence.

2) Porting the kernels: In Controller, a *HitTile* is a data type that represents data structures with descriptive metadata,

Listing 4. Snippet of Controller code for the core part of the algorithm. Controller functions and types are highlighted in brown color.

```
// Create lower resolution versions of images (src y tgt)
...
// Initial estimate (u, v) starts at 0
for (lvl = 0; lvl < n_lvls; ++lvl) {
  for (int wi = 0; wi < n_warps; ++wi) {
    int   cid  = Get_Cid(lvl, wi); // ctrl id
    PCtrl ctrl = Ctrl_Get(cid);
    ...
    // Warp target image according to current estimate
    Ctrl_Launch(ctrl, Warp, thr_space[lvl], CTRL_BLK_DEF,
      pp_tgt[lvl][cid], pp_u[lvl][wi], pp_v[lvl][wi],
      p_tmp[cid]);

    // Compute matrices of the equation to solve
    ...
    // Solve equation for du, dv (stencil)
    for (int iter = 0; iter < n_solves; ++iter) {
      Ctrl_Launch(ctrl, Solve, thr_space_aug[lvl],
        CTRL_BLK_DEF, p_du0[cid], p_dv0[cid],
        p_Ix[cid], p_Iy[cid], p_Iz[cid],
        alpha, p_du1[cid], p_dv1[cid]);
      hit_tileSwap(p_du0[cid], p_du1[cid]);
      hit_tileSwap(p_dv0[cid], p_dv1[cid]);
    }

    // Update current estimate
    Ctrl_Launch(ctrl, Add, thr_space[lvl], CTRL_BLK_DEF,
      pp_u[lvl][wi], p_du0[cid], pp_u[lvl][wi + 1]);
    Ctrl_Launch(ctrl, Add, thr_space[lvl], CTRL_BLK_DEF,
      pp_v[lvl][wi], p_dv0[cid], pp_v[lvl][wi + 1]);
  }

  // Prolongate solution (u, v) for use in the next level
  ...
}
```

pointers to memory allocations, etc. A HitTile can be allocated on the host and several devices, with a memory image on each of them. The metadata also includes information about whether the memory images on the host and devices have been read or written. This allows the Controller model to detect the need for data movements to maintain consistency among different copies of a HitTile's memory on the host or across various devices.

To port a CUDA kernel to the Controller programming model, we change the array parameters to HitTile types. In the Controller model, the prototypes of the kernel functions also include information about the input or output role of a HitTile parameter to automatically detect the need for data movements across the host and devices. The array accesses in the original code are replaced with an equivalent Controller function for accessing data in HitTile structures. The native CUDA thread indexes used in the array accesses are replaced with their Controller counterpart. The use of shared memory, like in the Jacobi solver kernel, does not need any adjustments. In general, this results in a slight simplification of the kernel code, with array sizes being built into HitTiles and global thread indexes being calculated implicitly by Controller.

3) Porting the host code: A snippet of the Controller host code for the core part of the algorithm is shown in listing 4. It shows how kernels are launched in Controller and how the pointer arrays are used to create the pipeline structure implicitly.

Controller programs read a configuration file at runtime to associate devices to device identifiers. The Controller runtime synchronizes all kernel and host-task execution according to the data dependencies derived from the input/output roles of the HitTile parameters. The implicit data movement mechanism of the Controller model automatically performs the needed data moves between devices asynchronously, to allow overlapping computation and communications efficiently [2], [22].

We initialize each frame by loading the data from the file system to the host memory as the first step of the pipeline. At the end of the pipeline, the norm is computed on the host for each frame pair. To ensure the frame loading and the norm operations are independent of each other, and the pipeline operations can overlap properly, we launch the frame loading as an asynchronous *host task*, and the norm calculation as an asynchronous kernel in a CPU device with a single thread.

IV. Experimental study

This section describes an experimental study carried out to verify the efficiency of the proposed solution.

A. Experimental environment and design

We design experiments to compare the performance results of our Controller implementation with reference versions of the application in CUDA and SYCL for NVIDIA GPUs, and OpenCL for AMD GPUs. We use as CUDA reference the original CUDA implementation, adding a loop to work on a sequence of frames. We have also modified the memory allocations to use pinned memory for the data structures in the host. This slightly improves the performance results, leading to a fair comparison with the Controller version where this decision is implicitly managed. We have ported this reference CUDA version to OpenCL to work with AMD GPUs. The SYCL reference is a port of the original CUDA program included with the Intel oneAPI samples as a guide to port CUDA programs to SYCL.

The experiments have been carried out on two heterogeneous machines of the Trasgo group research cluster, named *Manticore* and *Gorgon*, and the *Leonardo* supercomputer at CINECA. They have the characteristics listed in table I.

The inputs of the programs are sequences of video frames of different lengths. The video images have 8K resolution (7680×4320). We measure the execution time of the program section that computes, for the whole input sequence, the optical flow between pairs of consecutive frames. We skip allocation and initialization of data structures.

For each scenario, 30 repetitions are run, to obtain a large enough sample for the Central Limit Theorem to be considered applicable. The results presented are the average execution times for each scenario after eliminating outliers, that is, results below or above the mean $\pm 1.5 \times IQR$ (interquartile range).

We consider the following platform selection scenarios. Reference programs only work on a single device. The scenarios using multiple devices are designed to test the Controller program:

TABLE I
SPECIFICATIONS FOR EACH EXPERIMENTAL PLATFORM

System	Processor	RAM	GPUs	Software
Manticore	2x Intel Xeon Platinum 8160 @ 2,10 GHz	512 GB DDR4	2x NVIDIA Tesla V100 32 GB HBM2 2x AMD Vega 10 XT Radeon PRO WX 9100	Rocky Linux 9.3 CUDA 12.6, ROCm 6.1.0 GCC 11.4 oneAPI 2024.2
Gorgon	2x AMD EPYC 7713 @ 2,0 GHz	512 GB DDR4	1x NVIDIA A100 80 GB HBM2 2x NVIDIA RTX 4500 Ada Gen 24 GB GDDR6 1x AMD NAVI31 Radeon PRO W7800	Rocky Linux 9.3 CUDA 12.6, ROCm 6.1.0 GCC 11.4 oneAPI 2024.2
Leonardo	1x Intel Xeon Platinum 8358 CPU @ 2.60GHz	512 GB DDR4	4x NVIDIA A100-SXM-64GB	RedHat 8.7 CUDA 12.3 GCC 12.2

- Manticore. 1 device: V100, WX9100. 2 devices: 2 V100, 2 WX9100, WX9100 + V100
- Gorgon. 1 device: A100, RTX4500, W7800. 2 devices: 2 RTX4500, RTX4500 + A100, W7800 + A100, RTX4500 + W7800. 3 devices: W7800 + RTX4500 + A100
- Leonardo. 1 device: A100. 2 devices: 2 A100. 4 devices: 4 A100

B. Work partition and mapping decisions

In the experiments with multiple devices, we decide the number of levels and iterations to be executed on each device with the following guidelines. We execute the program with the same pair of frames on each device to measure the execution times of a warp iteration of each level on each device. Let $t_{l,d}$ be the measured execution time of a warp iteration $w_{l,i} : i \in [0, n)$ of level $l \in [0, m)$, on the device $d \in [0, k)$. Let $m : (l, i) \rightarrow [0, k)$ be the mapping function that associates a warp iteration $w_{l,i}$ with the device identifier d where it is executed. Valid mapping functions m divide the space of level-warp indexes in consecutive sets that are assigned to the devices: $m(l, i) \leq m(l', i') : l < l' \vee l = l', i < i'$. When the pipeline is full and the operations executed by each device are fully overlapped, the maximum execution time of a pipeline stage is p:

$$p_d = \sum t_{l,m(l,i)} : l \in [0, m), i \in [0, n), m(l, i) = d \quad (1)$$

$$p = max(\ p_d : d \in [0, k)\ ) \quad (2)$$

We obtain the potentially optimal work partition by finding the mapping function m that minimizes the execution time of the longest pipeline stage p. Finding the work partitions is currently done offline with a dynamic programming method. For our proposed multi-device scenarios, the best work partitions found using this approach are shown in the top part of table II.

C. Results

In this section, we discuss the results of the experimentation and our observations.

Figure 3 shows a comparison, using one device, of the execution times of the reference and the Controller versions for a single iteration of the algorithm, using only two frames, in several machines and devices. This experiment shows the basic efficiency of the coordination and memory management mechanisms of each version before exploiting multiple devices with Controller. The results show that the Controller program is nearly as efficient as the best CUDA reference version using pinned memory. They also show that the SYCL version is slower than the references. The main problem is that SYCL only supports a 4-channel image format, and the code is adjusted to mimic the use of the original single-channel textures with 4-channels textures using padding. For a fair comparison, the rest of the experimental study uses as a baseline only the CUDA reference program with pinned memory.

Figure 4 shows plots with execution times of the reference and Controller programs on each of the devices of the corresponding machine and speedups with different scenarios of device combinations. All the plots represent the number of frames on the x-axis. Let t_d be the execution time of the application for the chosen number of frames when it is fully mapped to device d. Let $t_f = min(t_d : d \in [0, k))$ be the execution time of the fastest device in the scenario. Let t be the execution time of the application mapped to several devices. We calculate the speedup in a given scenario with a heterogeneous environment as $S = t_f/t$, the ratio between the execution time when using the fastest device alone for the whole computation and the execution time when exploiting several devices. The speedup plots show S values relative to the fastest device, whose name is shown in the plot title.

We define the *Heterogeneous Expected Speed-up* as $S_e = \sum_{d=0}^{k-1} t_f/t_d$. This metric estimates the best speedup that could be obtained, with no overheads and a perfect load balance adapted to the relative computing power of the devices for this application. We define the *Heterogeneous Efficiency* as $\text{HE} = (S/S_e) \times 100$, the ratio between the measured speedup and the heterogeneous expected speedup in percentage. The measured and expected speedups and heterogeneous efficiency of each scenario are also shown in table II.

The execution time plots on all machines show that the Controller program in one device is slightly faster than the CUDA reference in NVIDIA platforms, and nearly as efficient as the OpenCL reference in the AMD platforms. The slight improvement over the CUDA reference version comes from overlapping the main computation with the host tasks, which are loading the frames and computing the norm.

For some scenarios (e.g. three different devices in Gorgon), the speedup for a few frames is lower than one because the pipeline is not filled yet and the time of the slower devices

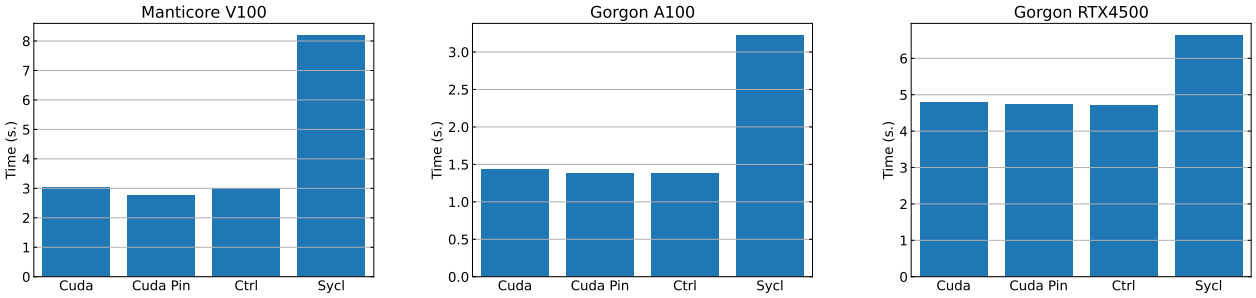
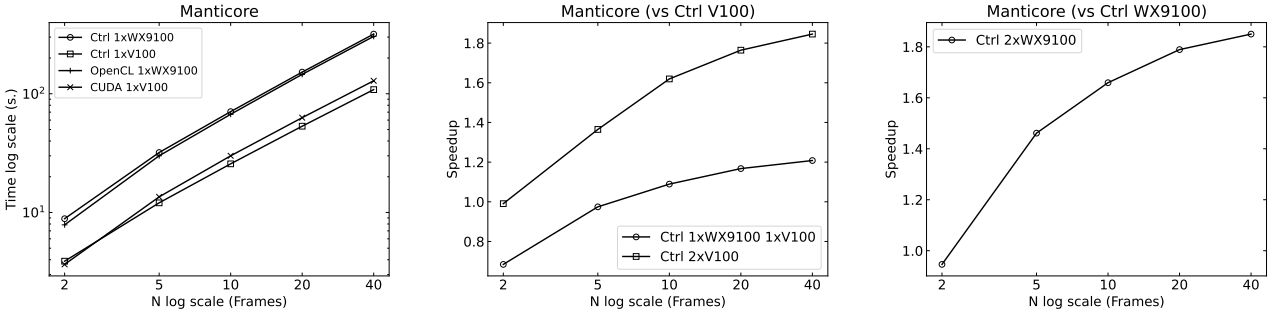
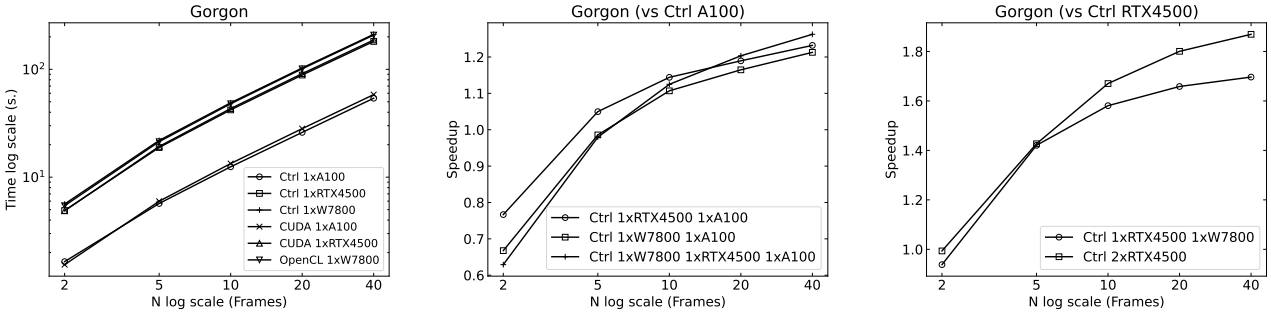


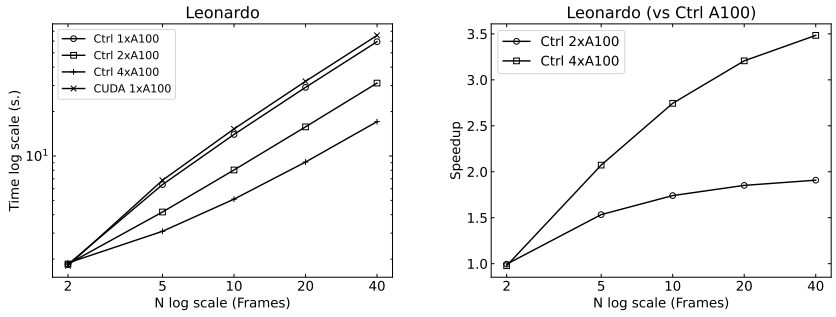
Fig. 3. Execution times of the baseline programs for a single pair of frames.



(a) Manticore



(b) Gorgon



(c) Leonardo

Fig. 4. Execution times and speedups obtained.

TABLE II

TOP: DEVICES AND PARTITIONS SELECTED FOR THE MULTI-DEVICE SCENARIOS. EACH COLUMN PRESENTS THE LIST OF GPUS USED. EACH DEVICE NAME IS FOLLOWED BY THE INDEXES OF ITS LAST LEVEL AND WARP ITERATION. BOTTOM: MEASURED AND EXPECTED SPEEDUP, AND EFFICIENCY, FOR 40 FRAMES.

Metric	Manticore			Gorgon					Leonardo	
	V100 4,0 V100 4,2	WX9100 4,0 WX9100 4,2	WX9100 3,2 V100 4,2	RTX4500 4,0 RTX4500 4,2	RTX4500 3,1 A100 4,2	W7800 3,1 A100 4,2	RTX4500 4,0 W7800 4,2	W7800 3,0 RTX4500 3,2 A100 4,2	A100 4,0 A100 4,2	A100 3,2 A100 4,0 A100 4,1 A100 4,2
S	1.85	1.85	1.21	1.87	1.23	1.21	1.70	1.26	1.91	3.48
S_e	2.00	2.00	1.37	2.00	1.29	1.25	1.87	1.55	2.00	4.00
%HE	92.5	92.5	88.3	93.5	95.3	96.8	90.9	81.3	95.5	87.0

hinders the performance. Nevertheless, in all the plots we observe how the speedup grows with the number of frames, as the pipeline stays full longer.

In scenarios that use two instances of the same type of device (see e.g. 2x WX9100 devices on Manticore) the speedups for 40 frames grow to at least 1.85x, with more than 92% HE. Some scenarios with mixed devices show speedups that may seem low for the number of devices but have relatively high HE (more than 90%) as this metric accounts for the differences in computing power among devices. Nevertheless, there are three specific scenarios with lower heterogeneous efficiencies (less than 90%). The first two are the scenario in Manticore with WX9100 and V100, and the one in Gorgon with the three devices. The differences in computing power do not allow proper mapping of warp iterations to create a better balance. The design of the pipeline structure with a finer granularity would be better for this scenario, allowing the selection of the cut points in the Jacobi iterations within the warp iterations. The third scenario appears in Leonardo using four identical A100 GPUs. There is some load unbalancing, but the longer time to fill the pipeline with four devices also affects the results with up to 40 frames. More frames are needed to continue improving the speedup. The overhead of synchronization and communication is less than 8.7% in all the scenarios considered.

V. CONCLUSION

This work presents a programming methodology to build pipeline solutions for multi-grid streaming applications, exploiting multiple heterogeneous devices of different vendors, architectures, or computing capacities. We use as a case study the HSOpticalFlow application, which estimates the apparent movement of objects in a sequence of images. We describe how to use the Controller heterogeneous programming and portability model to build a practical implementation. The Controller execution layer transparently handles data transfers across devices, deriving them from task dependencies and creating an efficient overlap of computation and communications. The resulting application runs as a parallel pipeline across devices. We introduce a technique to systematically determine a proper work partition and mapping for a set of devices. We present an experimental study designed to show the efficiency of the proposed solution compared to a reference CUDA version, and its ports to SYCL and OpenCL. The results show that the proposed solution is close in efficiency to the best reference versions when executing on a single GPU. When using combinations of several devices, our implementation obtains good speedups, with more than 90% of efficiency in most cases. Situations where efficiency is between 80% and 90% could be improved by using a finer granularity for the work partition.

In future work, we plan to study the use of the proposed mechanisms in other applications and contexts, explore their exploitation with new classes and combinations of heterogeneous devices including multi-node scenarios, devise a tuning algorithm to automatically find the optimal work partition, and consider the transparent introduction of new levels of granularity in the mapping to allow higher precision for load balancing.

ACKNOWLEDGMENT

This work is part of the action PID2022-142292NB-I00 (Knowledge Generation Project 2022), funded by MICIU/AEI /10.13039/501100011033 and by FEDER, EU. Support has also been received from the Investigo Program of the State Public Employment Service, Call for the Hiring of Research Personnel, financed by the European Union-NextGenerationEU. This research was partially supported by grants from NVIDIA and utilized an NVIDIA A100 GPU. It was also supported by EuroHPC Joint Undertaking for awarding us access to Leonardo at CINECA, Italy (project EHPC-DEV-2024D07-079).

REFERENCES

[1] The Khronos SYCL Working Group, "Sycl 2020 specification (revision 8)," The Khronos Group, Tech. Rep., 2023.

[2] Y. Torres, F. J. Andújar, A. Gonzalez-Escribano, and D. R. Llanos, "Supporting efficient overlapping of host-device operations for heterogeneous programming with ctrlevents," *Journal of Parallel and Distributed Computing*, vol. 179, p. 104708, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731523000722

[3] S. Farrelly, R. R. Manumachu, and A. Lastovetsky, "Openh: A novel programming model and api for developing portable parallel programs on heterogeneous hybrid servers," *IEEE Access*, vol. 12, pp. 23 666–23 694, 2024.

[4] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1, pp. 185–203, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370281900242

[5] M. Heath, *Scientific Computing: An Introductory Survey*. McGraw Hill, 1997, ch. 11.5.7.

[6] E. Meinhardt-Llopis, J. Sánchez Pérez, and D. Kondermann, "Horn-Schunck Optical Flow with a Multi-Scale Strategy," *Image Processing On Line*, vol. 3, pp. 151–172, 2013, https://doi.org/10.5201/ipol.2013.20.

[7] G. Araujo, D. Griebler, D. A. Rockenbach, M. Danelutto, and L. G. Fernandes, "Nas parallel benchmarks with cuda and beyond," *Software: Practice and Experience*, vol. 53, no. 1, pp. 53–80, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3056

[8] NVIDIA, "Cuda samples: Hsopticalflow," GitHub, on https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/HSOpticalFlow.

[9] Intel, "oneapi samples: Hsopticalflow," GitHub, on https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/StructuredGrids/guided_HSOpticalFlow_SYCLMigration.

[10] T. Gopal and J. Watada, "Gpu based horn-schunck method to estimate optical flow and occlusion," in *Theory and Applications of Models of Computation*. Springer, 2019, pp. 424–437. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-14812-6_26

[11] Y. Mileva, A. Bruhn, and J. Weickert, "Illumination-robust variational optical flow with photometric invariants," in *Proceedings of the 29th DAGM Conference on Pattern Recognition*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 152–162.

[12] M. Gonzàlez-Tallada and E. Morancho, "Compute units in openmp: Extensions for heterogeneous parallel programming," *Concurrency and Computation: Practice and Experience*, vol. 36, no. 1, p. e7885, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.7885

[13] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, and N. Ellingwood, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[14] AdaptiveCPP contributors, "Adaptivecpp," GitHub, on https://github.com/AdaptiveCpp/AdaptiveCpp.

[15] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Programming Accelerated Systems Using C++ and SYCL*. Apress, 01 2023.

[16] H. Carter Edwards and Daniel A. Ibanez, "Kokkos' task dag capabilities," Sandia National Laboratories, Tech. Rep. SAND2017-10464, 2017.

[17] M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, M. Torquati, M. Aldinucci, and P. Kilpatrick, "The rephrase extended pattern set for data intensive parallel computing," *International Journal of Parallel Programming*, vol. 47, no. 1, pp. 74–93, Feb. 2019.

[18] A. Ernstsson, L. Li, and C. Kessler, "Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 62–80, feb 2018. [Online]. Available: https://doi.org/10.1007/s10766-017-0490-5

[19] G. Tzanos, V. Soni, C. Prouveur, M. Haefele, S. Zouzoula, L. Papadopoulos, S. Thibault, N. Vandenbergen, D. Pleiter, and D. Soudris, "Applying StarPU runtime system to scientific applications: Experiences and lessons learned," in *POMCO 2020 - 2nd International Workshop on Parallel Optimization using/for Multi- and Many-core High Performance Computing*, Barcelona / Virtual, Spain, Dec. 2020. [Online]. Available: https://inria.hal.science/hal-02985721

[20] B. Pérez, E. Stafford, J. Bosque, and R. Beivide, "Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems," *Journal of Parallel and Distributed Computing*, vol. 157, no. C, pp. 30–42, nov 2021. [Online]. Available: https://doi.org/10.1016/j.jpdc.2021.06.003

[21] M. Smirnov, "Optical flow estimation with cuda," NVIDIA, White Paper, 2013.

[22] S. Alonso Pascual, "Modelo de ejecución y sincronización en múltiples dispositivos heterogéneos," Departamento de Informática, Facultad de Informática de A Coruña, Trabajo Fin de Máster (Master thesis), Marzo 2023, https://trasgo.infor.uva.es/sdm_downloads/tfm-sergio-alonso-pascual/.