This is the authors' version of the accepted work, posted for personal use; not for distribution.
The definitive version is published in ACM ICPP 2022 Proceedings, DOI:10.1145/3547276.3548517

OpenMP Offloading in the Jetson Nano Platform

Ilias K. Kasmeridis
Department of Computer Science and Engineering,
University of Ioannina
Ioannina, Greece
ikasmeridis@cse.uoi.gr

Vassilios V. Dimakopoulos
Department of Computer Science and Engineering,
University of Ioannina
Ioannina, Greece
dimako@cse.uoi.gr

ABSTRACT

The NVIDIA Jetson Nano is a very popular system-on-module and developer kit which brings high-performance specs in a small and power-efficient embedded platform. Integrating a 128-core GPU and a quad-core CPU, it provides enough capabilities to support computationally demanding applications such as AI inference, deep learning and computer vision. While the Jetson Nano family supports a number of APIS and libraries out of the box, comprehensive support of OpenMP, one of the most popular APIS, is not readily available. In this work we present the implementation of an OpenMP infrastructure that is able to harness both the CPU and the GPU of a Jetson Nano board using the offload facilities of the recent versions of the OpenMP specifications. We discuss the compiler-side transformations of key constructs, the generation of CUDA-based code as well as how the runtime support is provided. We also provide experimental results for a number of applications, exhibiting performance comparable with their pure CUDA versions.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; **Multicore architectures**; • **Software and its engineering** → **Compilers**.

KEYWORDS

Jetson Nano, OpenMP, CUDA, compilers, runtime systems, offloading

ACM Reference Format:
Ilias K. Kasmeridis and Vassilios V. Dimakopoulos. 2022. OpenMP Offloading in the Jetson Nano Platform. In *51th International Conference on Parallel Processing Workshop (ICPP Workshops '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3547276.3548517

1 INTRODUCTION

Contemporary computing systems are characterized by heterogeneity, a feature found everywhere from low-budget PCs to high-performance supercomputers. A typical desktop computer utilizes a number of general-purpose CPU cores, along with a graphics processing unit (GPU) which accelerates computer graphics workloads. In addition to GPUs, other types of accelerators are also employed widely in the world of high-performance computing, including FPGAs or digital signal processors, so as to accelerate specific families of computations.

Striving mostly for low power, embedded systems are nowadays able to provide significant computing capabilities. As the performance requirements of applications tend to increase continuously, embedded systems have also embraced heterogeneity. Some indicative real-world examples are the TI Keystone II SoC [15], which combines a dual-core ARM CPU with TI C66x floating-point DSPs, the Parallella [1] with a ARM-based CPU and a 16-core Epiphany accelerator, as well as the NVIDIA Jetson family [19], which is based on a multicore ARM CPU and a manycore NVIDIA GPU.

Programming such systems can be a challenging task; it often entails the utilization of lower-level APIS, sometimes with steep learning curves, and this is particularly pronounced in the case of heterogeneous systems. Programming models such as OpenCL [9] and CUDA [18] are typical examples of platforms and APIS that are used to target a variety of accelerators and GPUs; they provide efficient albeit rather primitive mechanisms for an application to exploit the hardware capabilities. Their requirement for different code bases for the host CPU and the accelerator/GPU increases code complexity and decreases portability.

Many works propose OpenMP as a promising higher-level programming model, which promotes programmer productivity due to its ease-of-use and maintainability. With the recent device offloading features, OpenMP constitutes a suitable candidate for targeting heterogeneous systems under a unified programming interface. However, the devices for which OpenMP support is available still remains rather limited.

In this work, we present the implementation of OpenMP for the NVIDIA Jetson Nano 2GB embedded platform. The Jetson Nano modules and boards are the smallest and most affordable members of the Jetson family, but they are still powerful enough to support compute-intensive applications. While they support a number of APIS and libraries out of the box, comprehensive support of OpenMP is not readily available. We extend the lightweight OMPi source-to-source compiler [8] so as to target CUDA-based GPUs and exploit both the ARM CPU and the Maxwell GPU of the board. In particular,

- We present the first full-fledged OpenMP implementation for the Jetson Nano 2GB with complete offloading capabilities.
- We discuss the way the OMPi compiler was extended to translate OpenMP device constructs into equivalent CUDA code.
- We evaluate our implementation experimentally, using a number of benchmarks and applications and compare its performance to pure CUDA code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP Workshops '22, August 29-September 1, 2022, Bordeaux, France
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9445-1/22/08…$15.00
https://doi.org/10.1145/3547276.3548517

The remainder of our paper is organized as follows: Section 2 gives an overview of the OpenMP API and its device model. In Section 3, we present the actual implementation in the translator of OMPi compiler. Section 4 presents the Jetson Nano 2GB board in detail, including its hardware facilities as well as the available software tools; we also discuss the runtime library of OMPi which provides OpenMP support for the offloaded CUDA kernels. Section 5 includes experimental results and, finally, Section 6 summarizes and concludes this work.

1.1 Related Work

Using OpenMP for programming embedded systems or SoCs has been considered in many works. For example, Mitra et al. [15] discuss the implementation of the OpenMP 4.0 device model for the Texas Instruments Keystone II SoC, where the device is a C66x floating-point DSP; Agathos, Papadogiannakis and Dimakopoulos [2] implement OpenMP 4.0, targeting the Epiphany accelerator of the Parallella board, a CPU with 16 superscalar RISC cores; Kurth et al. [12] present OpenMP offloading for a custom version of the HERO [11] heterogeneous embedded platform.

The NVIDIA Jetson ecosystem has been quite popular in a variety of applications. Focusing on the Jetson Nano family, [22] employs Jetson Nano boards to evaluate existing parallel programming models for automotive workloads. Mohebbanaaz, Sai and Rajani Kumari [16] use a single Jetson Nano board to implement and test a deep learning model that detects cardiac arrhythmia. Vishwani et al. [21] employ a Jetson Nano board to support face emotion recognition.

Supporting GPUs through OpenMP has been the subject of many works, even before the introduction of its device model. Early works include [13] and [17] which propose mechanisms for translating `parallel` regions and loops to equivalent CUDA code. After the OpenMP device model was added, Bertolli et al. [5], as well as Yang and Huiyang [24] proposed thread coordination schemes for kernels that contain multiple `parallel` regions. The integration of [5] into the CLANG/LLVM compiler is described in [6] and in [3] for Power systems containing NVIDIA GPUs.

2 OPENMP AND HETEROGENEITY

A key feature introduced in version 4.0 of OpenMP is its platform-independent device model, enabling seamless programming of heterogeneous systems. This model was designed to allow programmers to utilize all available compute devices, aiming to increase the performance and power efficiency of applications. One can simply accelerate specific areas of code (*kernels*), by advising the compiler to offload them to a device, which is connected to the host CPU. The kernel execution, along with the data mapping between the CPU and the device are orchestrated transparently by the compiler and the runtime library.

An application is executed on the host processor until a `target`-related construct is met, which can indicate that either a portion of code is to be offloaded to a device, or a device data environment must be created / modified. These constructs may contain a code region or act as stand-alone directives. By the end of the construct execution, control is returned to the CPU. Specifically, control flow can be transferred to a device with the use of the `target` directive, which contains a block of code. Data referenced inside the code

```
1  /* Host function that performs SAXPY on the device */
2  void saxpy_device(float a, float x[], float y[], int size)
3  {
4    #pragma omp target map(to: a,size,x[0:size]) \
5                       map(tofrom: y[0:size])
6    {
7      int i;
8      #pragma omp parallel for
9      for (i = 0; i < size; i++)
10       y[i] = a * x[i] + y[i];
11   }
12 }
```

Figure 1: Offloading code to a device.

block must be mapped to the device data environment, thus the `target` directive is usually parameterized with `map` clauses which specify the type of mapping that should be performed for each host variable (`alloc`, `to`, `from` and `tofrom` mappings).

Fig. 1 shows a simple example, where `saxpy_device` is a host function that performs SAXPY (Single-precision A*X plus Y) on the device, given two input arrays `x` and `y`; the result is stored in `y`. The device data environment includes variables `a`, `x`, `size` and `y`. The first three are mapped as `to`, thus a copy of them will transferred *to* the device. Variable `y` is mapped as `tofrom`, meaning that two actions will take place; upon entering the `target` construct, the compiler will transfer a copy of `y` *to* the device, and when the construct execution is completed, there will be a transfer *from* the device, back to the host. It should be noted that actual transfers may not be needed if the host and the device physically share memory. Finally, the `parallel for` directive triggers the creation of a device-side thread team, to which the `for` loop iterations will be distributed.

The device data environment creation is a process that can be initiated separately from the kernel execution. For variables which are used across multiple `target` regions, the programmer can avoid the repeated creations of data environments, by using the `target data` directive. The unique feature of `target data` is the ability to enclose multiple `target` constructs that can rely on a single data environment, substantially reducing unnecessary data movements.

Additionally, OpenMP offers the stand-alone directives `target enter / exit data` and `target update`. The former directives trigger a mapping (unmapping) of variables to (from) the device data environment, while the latter is used for keeping the device data environment consistent with the host data environment; these directives can appear at any point in the host code. Furthermore, `declare target` and `end declare target` directives can be used to mark global variables and function prototypes for access within the offloaded kernels.

Last but not least, as can be seen in Fig. 1, a `target` region may contain further OpenMP functionality. The support of OpenMP directives inside a device is specific to the device runtime being used during each kernel execution. Information about the structure of the device runtime will be discussed later.

3 GPU SUPPORT IN THE OMPI COMPILER

The OMPi compiler is an open-source, lightweight C infrastructure for OpenMP. Its main components are a source-to-source translator and a runtime system. The translator takes a C program that makes

use of OpenMP directives (pragmas) and produces a multi-threaded program designed for execution by the host; each directive gets replaced by equivalent C code, along with appropriate calls to the runtime system that implement the corresponding functionality. The generated source file can then be compiled using the system compiler and linked with the necessary libraries. If the application contains target directives, OMPi additionally generates kernel sources for all supported devices. These sources are compiled to obtain device-specific executables using the corresponding device tools. The full compilation chain is depicted in Fig. 2.

OMPi uses an abstract syntax tree (AST) to represent the user program; most of its transformations operate directly on the AST. Similarly to parallel and task directives, *outlining* is used when a target directive is encountered. The relevant portion of the AST, i.e. the body of the construct, is moved to a new function (*kernel function*) and its AST node is replaced by necessary data movements and code offloading runtime calls to/from the device in question.

Constructing the kernel function out of the body of the target construct requires complex preparatory work. It can be relatively straightforward if the device the kernel will be offloaded to contains general-purpose processing units. In such a case, assuming a standard C compiler is available for the device, the kernel function is more or less a replica of the original construct body; code generation mostly involves outputting its syntax subtree as C source code. OMPi currently supports two general-purpose OpenMP devices: the Epiphany accelerator [2] and a special one, which presents the nodes of a compute cluster as multiple OpenMP devices where code can be offloaded to using MPI as the communication substrate [10]. It is a completely different case, however, when targeting GPUs, because of their SIMD nature.

In order to support devices with non-general purpose processing units, a series of extensions were implemented in the translator part of OMPi. The transformation phase of OMPi was altered so as to support multiple transformations of a given OpenMP node, one for each different device. The resulting AST subtrees are then passed to different translator modules, according to the targeted device.

The code generation works in a modular fashion, mapping each OpenMP directive to a different internal transformation function. These functions form the *transformation set*. Internally, OMPi keeps a default set for general-purpose devices and another one for GPUs. For CUDA devices, we have implemented a CUDA C module that produces CUDA C code for each kernel function of the GPU subtree. Moreover, there also exists preliminary support for OpenCL devices, offered by a corresponding OpenCL module.

Focusing on GPU-type devices, upon encountering a target directive, OMPi first constructs the subtree for the outlined kernel function, as described previously. The compiler then derives the call graph of the subtree, by discovering all called functions inside the kernel. This step is required in order to inject all the necessary function prototypes and definitions and embed additional necessary wrapper functions. The final subtree is used to generate the kernel file, which contains pure CUDA C code.

3.1 Teams, Distribute and Combined Constructs

A typical use case for GPUs is the offloading of computationally intense loops. A target teams directive constructs a league of thread teams to execute the attached block of code in the device. While each team initially contains only one thread, the loop iterations can be executed concurrently by the teams using a distribute construct. A second level of parallelism is possible if within each team a parallel or parallel for construct is used to enable multiple threads within each team.

OpenMP offers the convenient combined target teams distribute parallel for construct, which is the recommended way to target loops to GPUs [7]. Given all the information in the combined construct, the compiler performs the following transformation steps:

- The teams directive and its clauses are used to generate a CUDA *grid* that consists of a number of CUDA *blocks*. In particular, the num_teams and num_threads clauses are used to specify the exact number of blocks and threads in each block that will execute the kernel, respectively. In addition, the programmer can make use of the thread_limit clause to specify an upper limit for the number of threads to be created.
- The iterations are initially distributed to the team masters. Each master thread receives its own iteration subspace, called *chunk* and additionally distributes it to the rest of the team members. Finally, all members execute their iterations, including the master threads.
  In practice, both distribution phases are executed by all GPU threads and there is no distinction between the team master and the team members. In the first distribution phase, each thread initially retrieves the chunk destined for the team master with a call to the runtime library function get_distribute_chunk. Then, all threads proceed to the next distribution phase and perform a call to get_static_chunk, get_dynamic_chunk or get_guided_chunk, according to the declared loop scheduling clause. The aforementioned functions belong to the device library and work in conjunction with the distributed chunk, returning the actual iterations subset meant for the calling thread.

3.2 Handling Standalone Parallel Regions

Having separate, non-combined parallel regions within the kernel code presents a major difficulty because the fork-join execution model of OpenMP does not match with the SIMD nature of GPUs computation units; this has been an important subject of research [5, 6, 24]. For CUDA version 3.5 and above, a possible solution is to utilize dynamic parallelism. This feature allows the creation of child kernels by threads already executing a kernel. Although dynamic parallelism is an elegant solution and may allow for cleaner code, it is generally accepted that it can introduce considerable runtime overheads.

Two solutions were proposed by Bertolli et al. [5]; *if-master*, which involves guarding the sequential portions of a kernel using an if-condition and *control-loop with inspector/executor model*, which is based on a state machine that consists of sequential states and parallel region states. Another proposal is the the *master-worker* scheme [24], which incorporates master threads to execute sequential regions and activates worker threads to handle parallel regions.

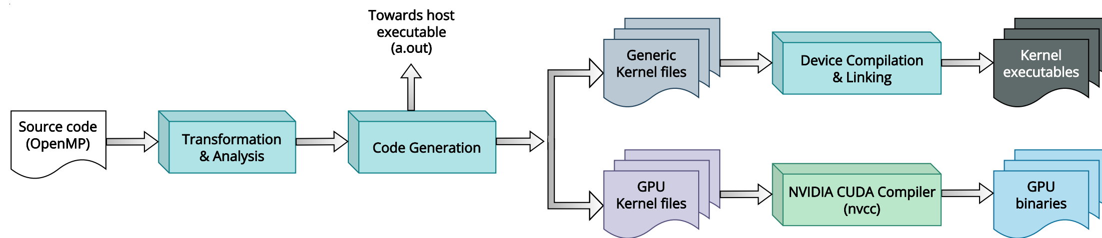


Figure 2: OMPi compilation chain

```
#pragma omp target map(tofrom: x[:96])
{
  int i = 2;
  #pragma omp parallel num_threads(96)
  {
    x[omp_get_thread_num()] = i+1;
  }
  printf(" x[0] = %d\n", x[0]);
  printf("x[95] = %d\n", x[95]);
}
```

(a) A target region with a `parallel` construct.

```
1  {
2    int _mw_thrid = omp_get_thread_num();
3    int _mw_nthr = omp_get_num_threads();
4    if (cudadev_in_masterwarp(_mw_thrid)) {  /* master warp */
5      if (!cudadev_is_masterthr(_mw_thrid))
6        return ((void *) 0);    /* 31 threads of master warp */
7
8      int i = 2;
9      /* #pragma omp parallel num_threads(96) */
10     {
11       __shared__ struct vars_st {
12         int (*i);
13         int (*x)[96];
14       } vars;
15
16       /* Shared memory */
17       vars.i = (int (*)) cudadev_push_shmem(&i, sizeof(i));
18       /* Device memory */
19       vars.x = (int (*)[96]) cudadev_getaddr(&(*x));
20
21       /* thrFunc0 contains the body of the parallel region */
22       cudadev_register_parallel(thrFunc0, vars, 96);
23       cudadev_pop_shmem(&i, sizeof(i));
24     }
25
26     printf(" x[0] = %d\n", (*x)[0]);
27     printf("x[95] = %d\n", (*x)[95]);
28
29     cudadev_exit_target(); /* End of target directive */
30   }
31   else {  /* worker warps */
32     cudadev_workerfunc(_mw_thrid);
33   }
34 }
```

(b) Produced kernel code.

Figure 3: An example of the code generated by the master / worker transformation scheme

OMPi currently follows the master/worker scheme, which is based on the producer-consumer pattern. A simplified example of how it works is shown in Fig. 3. This pattern is based on the fact that the threads of a CUDA block are scheduled in batches of 32 threads, each batch called a *warp*. The compiler wraps the body of the target directive in Fig. 3a with code that divides the executing warps to one *master* warp and several *worker* warps, which contain worker threads. Only a single thread of the master warp is labeled as *master*; the remaining threads of the master warp are deactivated, i.e. return from the kernel function.

The responsibility of the master thread is to execute any sequential code and, additionally, assign the execution of parallel regions to worker threads. Worker threads operate within an infinite loop and are only terminated when reaching the end of a `target` region. While all kernels are launched with a fixed number of worker threads (96), a subset of them participate in a `parallel` region if there exists a `num_threads` clause; the rest of the threads remain inactive until the participating threads finish their execution.

This scheme is based on two named PTX ISA barriers, B1 and B2. B1 is used for synchronization of the master thread with all worker threads, before and after the execution of a parallel region, while B2 only involves threads that actually participate in the execution of the region. Initially, all worker threads block performing a synchronization call to B1. Whenever the master thread encounters a parallel region, it registers the thread function to be executed along with the necessary pointers to shared/global variables (registration phase) and then arrives at B1, waking up the blocked workers. Upon completion of the parallel region, a call to B2 takes place and afterwards, B1 is utilized again by all threads, implying the termination of the region.

The entire master thread functionality regarding parallel regions, is implemented in the `cudadev_register_parallel` runtime library function (line 22 in Fig. 3b). Worker threads execute only the `cudadev_workerfunc` call (line 32). Variables declared as `shared` for a specific parallel region, need to reside in the shared memory of each block being executed. For this purpose, we have implemented a shared memory stack, along with two basic functions: `cudadev_push_shmem` pushes a variable to the shared memory and returns a pointer to the newly allocated memory space (line 17), while `cudadev_pop_shmem` pops a variable from the stack, i.e. deallocates the previously allocated space (line 23). When reaching the end of the `target` region, a call to `cudadev_exit_target` by the master thread (line 29) terminates all the worker threads.

3.3 Kernel Binaries

A notable difference between OMPi and other compilers is that it does not embed the kernel files into the final executable. Instead, each kernel is output to an independent file. All kernel files are pure

CUDA C sources, which are ready for separate compilation with the available CUDA tools. The external tools are invoked by specific scripts, accompanied with the generated code. Moreover, through these tools, OMPi can instruct the NVCC compiler to produce any of two types of binaries, PTX or cubin, depending on the options used during OMPi configuration.

In PTX mode, kernel files are translated to the intermediate PTX (Parallel Thread Execution) format and the final step of their compilation is handled at runtime just before the actual offloading. This process is called JIT (just-in-time) compilation and tends to produce lighter kernel binaries. Moreover, it utilizes disk caching, a CUDA feature that aims to eliminate repetitive compilations of the same kernels. PTX files are GPU-architecture agnostic.

The *cubin* mode performs all the compilation steps and produces larger binaries, called cubins. Each cubin contains machine code for a single targeted CUDA architecture. While also slower, cubin mode does not depend on JIT compilation and thus reduces runtime overheads. For this reason OMPi uses the cubin mode by default. The actions of locating the binaries, loading them and offloading to the GPU are all carried by the runtime library of OMPi.

4 JETSON NANO 2GB AND OPENMP

The Jetson Nano family consists of a module and two development boards, and is a very popular embedded platform that has been employed for a wide range of computationally intensive applications (e.g. [4, 23, 14]). In what follows, we will ignore the module, since its features are identical to one of the development boards. Apart from its standard I/O connectivity, a major strength of Jetson Nano is its combination of low power consumption (5W) with significant processing capabilities. In particular, it consists of two computational devices: a quad-core ARM A57 host processor running at a frequency of 1.43 GHz and a 128-core NVIDIA Maxwell GPU supporting CUDA architecture version 5.3. The main difference between the original Jetson Nano and the Jetson Nano 2GB board, which is the one used in this work, lies in the amount of main memory available (4GB in the former, 2GB in the latter).

4.1 Available Offloading Options

Currently, CUDA C and related APIs constitute the main way of utilizing both the CPU and the GPU of a Jetson Nano board. Code written in CUDA C is translated by the NVCC driver to a host file and a device (kernel) file. The host file embeds CUDA runtime API calls to load and execute the kernel file on the device, and can be linked with the rest of the application files. Alternatively, a programmer may use plain C host code and utilize CUDA runtime API calls to manually orchestrate the kernel compilation and offloading process. Either way can be an efficient, albeit rather lower-level programming style for a heterogeneous system.

The standard compilers included in the JetPack SDK (GCC and NVCC) provide OpenMP support but only for the host ARM CPU; they cannot offload code to the GPU. Porting recent versions of GCC or other mainstream compilers, is almost impossible due to the limited resources of the board. We managed to build GCC version 11 from its sources but offloading could not be made to work. Although there has been some report that CLANG/LLVM can be built using specialized scripts for the 4GB Jetson Nano board, we did not have any success with our 2GB models.

Recently, the NVC/NVC++ compiler bundled in the NVIDIA HPC toolkit implements most OpenMP offloading facilities. However, these features are only available for devices with CUDA architecture version greater than 7.0, effectively excluding the less powerful Jetson families of boards (the CUDA architecture is 5.3 for the Nano's GPU and 6.2 for the TX2). Conclusively, OpenMP offloading is currently unavailable.

Our work presents the first full-fledged OpenMP implementation which supports the host CPU, while also offering offloading capabilities targeting the Maxwell GPU. Moreover, what we consider important is the lightweight nature of OMPi, which bases its compilation chain only on the standard toolkit of the Jetson Nano platform.

4.2 OpenMP Offloading in OMPi

Regarding devices, the runtime system of OMPi is organized as a collection of *modules*, each one implementing support for a particular device class; multiple devices (of the same type) may be served by one module. Modules consist of two parts: the host part and the device part. The former enables the host CPU to access any of the available module's devices through a fixed interface and is loaded on demand as a plugin (shared library). The device part provides OpenMP and other runtime support within the device, for the offloaded code.

To support a new device, one has to create a module that implements OMPi's interface for communication with devices. For the Jetson Nano board, we created a new *cudadev* module; although we target the Maxwell GPU of the board, the module has been designed to be quite general so that it can be adapted to support other CUDA-based GPUs as well.

4.2.1 The host part of the cudadev module. The host part is responsible for discovering and establishing a communication path with the device so as to be able control it, transfer data and offload code. While all devices are discovered during the application startup phase, the *cudadev* module adheres to OMPi's lazy approach to actual device initialization: a device is fully initialized only when the first kernel is about to be offloaded to this particular device. During initialization, all the hardware characteristics of the GPU are obtained and stored in relevant data structures. In addition, a primary CUDA context is created. Once the device has been initialized, the host part of the module offers functions to allocate/deallocate memory in the device as well as transfer data to/from the device. For all the above functionality, the module employs lower-level CUDA driver API calls.

A central operation of the *cudadev* module is kernel launch, which consists of three phases:

(1) The loading phase, which locates the kernel function in the corresponding kernel file and creates a CUDA kernel function. If the kernel binary is in PTX format, there exists a preliminary step of just-in-time PTX compilation and linking with the *cudadev* device library. If OMPi operates in cubin mode, there is neither JIT compilation nor linking with the device library needed; everything was handled at compile time.

(2) The parameter preparation phase, responsible for passing the correct parameters to the kernel that is to be launched, maintaining a mapping of these parameters to their corresponding host addresses.
(3) The launch phase, which performs the actual kernel launch. In this phase the CUDA grid and block dimensions of the launching kernel are set. Then, it performs a call to `cuLaunchKernel` function, using the parameter set, the loaded CUDA kernel function and the required dimensions.

4.2.2 The device part of the cudadev module. The device part (device runtime library) contains the implementation for the OpenMP functionality employed inside the offloaded kernels; it gets linked with a compiled kernel file and is offloaded along with it. The operations of the most important OpenMP facilities, currently supported by the device part, are outlined below:

- *Parallel regions*; As already mentioned, *cudadev* supports both standalone `parallel` constructs, as well as combined constructs, where `parallel` is combined with a `target` directive, such as `target teams distribute parallel for`. In the non-combined case, OMPi initiates kernels with a fixed number of 128 threads, since the Jetson Nano GPU has a total of 128 cores in its streaming multiprocessor. All warps but one (the master warp) are masked out in the sequential regions of the kernel. A subset of the remaining 3 worker warps operate in a parallel region; the subset size is determined by the number of threads requested by the application, otherwise all 96 cores are employed. Threads are synchronized before and after the execution of a parallel region. Combined parallel directives do not utilize the master/worker scheme at all; the number of threads requested by the programmer equals the number of launched threads, all of which execute the enclosed parallel region.
- *Worksharing*; all schedules are supported (static, dynamic, and guided) for `for` loops; the static schedule is also supported for `distribute` directives. `sections` directives are implemented using locks; the library keeps track of the remaining sections using a counter initialized to the number of sections. The thread that reaches a section first acquires a lock and reduces the counter until the latter becomes 0. To avoid warp divergence, each section is assigned to threads from different warps. All `single` regions are executed by the master thread, using a logic similar to the if-master scheme. All threads are synchronized after the execution of worksharing regions, unless a `nowait` clause was used.
- *Synchronization*; In CUDA, threads within a warp operate in lockstep. Providing an efficient locking mechanism is non-trivial mostly because of the warp *divergence* that takes place when threads belonging to the same warp take different execution paths. We implement locks through busy-spinning with atomic compare and swap (CAS) instructions on a global control variable; it gets the value of 1 by the thread that acquires the lock, while the rest of the threads wait until the variable becomes 0 and the lock is released. OpenMP `critical` regions utilize the implemented locking mechanism; the compiler generates the lock/unlock calls before/after the region.
- *Barriers*; Our implementation is based on named barriers. An encountered barrier construct is translated to a `bar.sync` PTX instruction, allowing a total of 16 barriers to be utilized by a single block. A restriction of the `bar.sync` instruction is that it can only accept, as an argument, a number of threads that is a multiple of the warp size ($W = 32$). If a subset of threads participating in a parallel region contains N threads, and N does not satisfy this restriction, *cudadev* performs a barrier synchronization for $X = W\lceil \frac{N}{W} \rceil$ threads. Eventually, CUDA will skip threads that did not call the function containing the barrier, i.e. the inactive threads that do not participate in the parallel region. In this way, any subset of N threads participating in a parallel region can be synchronized independently of the remaining $X - N$ inactive threads.

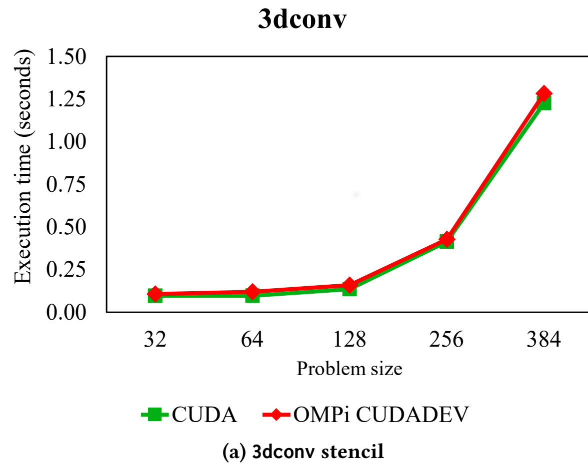
5 EXPERIMENTAL RESULTS

In this section we report performance tests, comparing existing CUDA applications to equivalent OpenMP ones, compiled with our compiler. The board we experimented with comes with JetPack version 4.6, based on the CUDA Toolkit version 10.2. The available compilers for the C language in Jetson Linux are GCC, which produces only host executables and NVCC, provided by the CUDA toolkit.

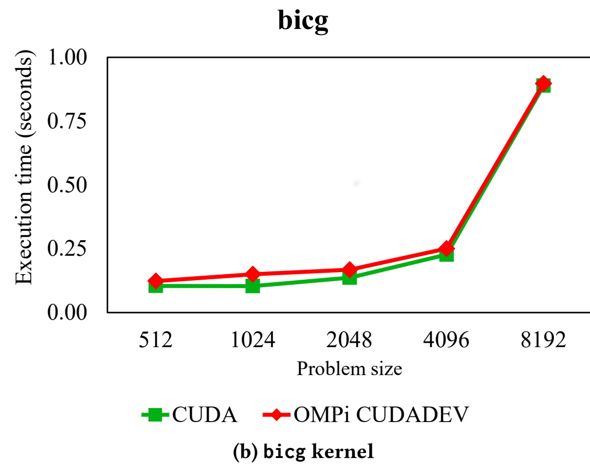
For our study we used the Unibench suite [20]. Unibench contains a remake of the Polybench-ACC benchmark suite, with most of the benchmarks modified so as to use OpenMP `target`-related constructs to offload compute kernels, instead of pure CUDA. The suite consists of a large collection of *kernel*, *stencil* and *solver* applications. Kernels are mini applications that perform basic calculations, while solvers are based on more complex code. Stencils are methods that calculate the value of an array cell according to its neighbourhood. For each application, the suite provides a sequential implementation along with its CUDA and OpenMP equivalents. We report here performance results from a selected set of one stencil application, four kernel applications, as well as one solver application: *3dconv*, *bicg*, *atax*, *mvt*, *gemm* and *gramschmidt*, which perform a variety of operations on matrices and vectors. They were obtained from the `linear-algebra` and `stencils` categories of the Polybench suite and represent typical GPU workloads. We get similar results with the rest of the applications in the suite.

The matrix sizes of all applications were parametrizable and for our purposes here we used three basic configurations: Applications `gemm` and `gramschmidt` use 128, 256, 1024 and 2048, as sizes for all dimensions of used matrices and vectors, while `bicg`, `atax` and `mvt` work on larger array sizes, 1024, 2048, 4096 and 8192. Triple-nested for-loops in `3dconv` use the third configuration which represents smaller problem sizes, of the order of 32, 64, 128, 256 and 384. Moreover, all applications use 32×8 threads, except for the `gramschmidt` application which is fixed to use 256×1 threads, and the `3dconv` application which uses $2 \times 4 \times 32$ threads. All OpenMP application equivalents make use of the recommended `target teams distribute parallel for` directive. The values we used for the `num_teams` and `num_threads` clauses matched the problem size. Internally, OMPi maps these values to two dimensions, so as to match the block and grid dimensions of the equivalent CUDA applications.

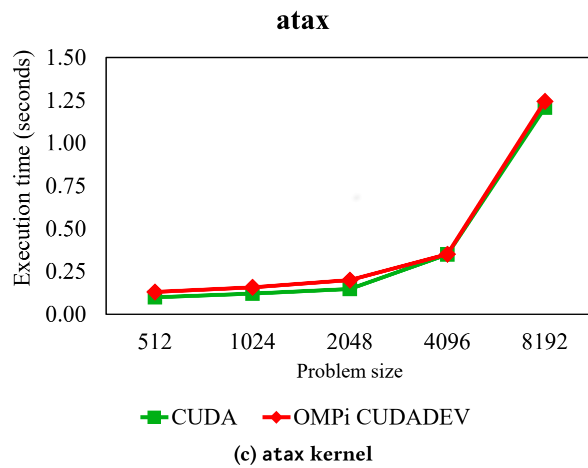
Some modifications were necessary for the OpenMP version of the applications, to better optimize the execution of the kernels. In



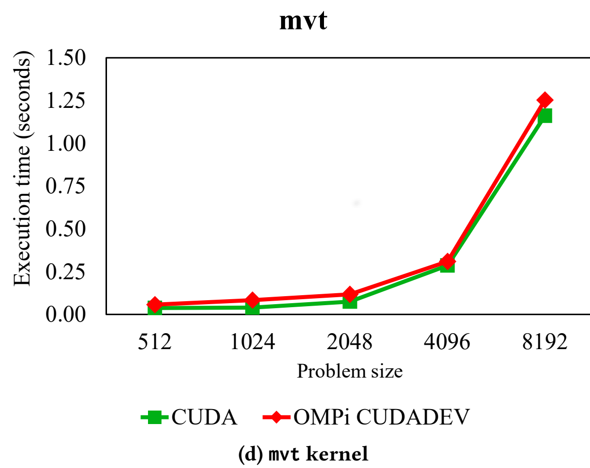
(a) `3dconv` stencil



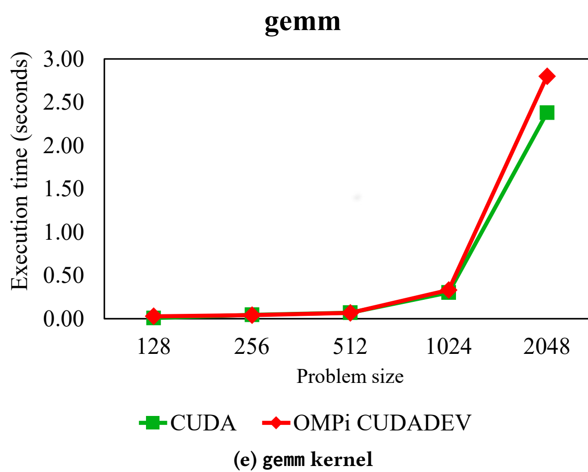
(b) `bicg` kernel



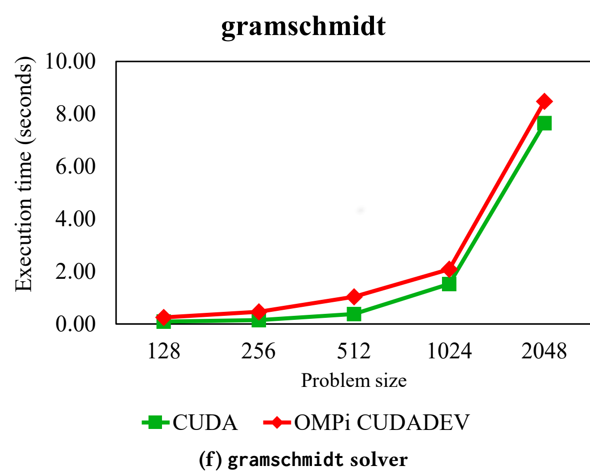
(c) `atax` kernel



(d) `mvt` kernel



(e) `gemm` kernel



(f) `gramschmidt` solver

Figure 4: Application execution times using the CUDA runtime API and the OMPi *cudadev* module

particular, we used the `collapse` clause to merge perfectly nested loops and we inserted proper `num_teams` and `num_threads` clauses to all `target`-related constructs.

The results are given in Fig. 4. Each plot contains timing results for the pure CUDA version and the OpenMP equivalent, which was compiled using OMPi. The x-axis is the problem size (the dimension of the involved matrices/vectors), while in the y-axis execution time is given in seconds. There was negligible variation among runs, so the execution time is reported as a simple average of 10 runs. We measure kernel execution time, plus any required memory operations.

Overall, the performance attained by our implementation proved quite satisfactory. For all applications, OMPi follows closely the performance of pure CUDA, as is evident in the plots. We only found one notable discrepancy that we have not managed yet to explain; it occurs in the *gemm* application and only for the largest problem size (2048), where the OpenMP executable is about 18% slower. We are currently investigating this phenomenon.

6 CONCLUSION

We present the first OpenMP implementation on the Jetson Nano 2GB developer kit that offers offloading capabilities for the Maxwell GPU of the board. The implementation is based on the OMPi OpenMP compiler which has been extended to target CUDA-based devices. The transformation and code generation phases have been adapted to produce CUDA C kernel sources out of OpenMP `target` regions; these kernels then get compiled and linked using only the base NVIDIA tool chain. The runtime module utilizes the CUDA driver API to handle memory allocations, data transfers and kernel launches, while the device library part provides OpenMP support for each offloaded kernel. We have experimented extensively with various benchmark and application suites that showcase the efficiency of our implementation, which matches closely the performance of pure CUDA code. We are currently generalizing our runtime system to support other NVIDIA GPUs. In addition, we work on further extending OMPi to target OpenCL devices.

This work is open-source software and is available as part of the latest release of the OMPi OpenMP compiler, which can be found in https://paragroup.cse.uoi.gr/wpsite/software/ompi/.

ACKNOWLEDGMENTS

We acknowledge support of this work by the project "Dioni: Computing Infrastructure for Big-Data Processing and Analysis." (MIS No. 5047222) which is implemented under the Action "Reinforcement of the Research and Innovation Infrastructure", funded by the Operational Programme "Competitiveness, Entrepreneurship and Innovation" (NSRF 2014-2020) and co-financed by Greece and the European Union (European Regional Development Fund).

The Jetson Nano 2GB developer kits used for this research were donated by the NVIDIA Corporation.

REFERENCES

[1] Adapteva. 2014. Parallella Reference Manual. (Sept. 2014).

[2] Spiros N. Agathos, Alexandros Papadogiannakis, and Vassilios V. Dimakopoulos. 2015. Targeting the Parallella. In *Proc. Euro-Par 2015, 21st Int'l European Conf. on Parallel and Distributed Computing*. Vienna, Austria, (Aug. 2015), 662–674.

[3] Samuel F. Antao et al. 2016. Offloading Support for OpenMP in Clang and LLVM. In *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC* (LLVM-HPC '16). Salt Lake City, Utah, 1–11.

[4] Luis Barba-Guaman, José Eugenio Naranjo, and Anthony Ortiz. 2020. Deep learning framework for vehicle and pedestrian detection in rural roads on an embedded GPU. *Electronics*, 9, 4.

[5] Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin OBrien Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *LLVM-HPC '14: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*. New Orleans, LA, (Nov. 2014), 12–21.

[6] Carlo Bertolli et al. 2015. Integrating GPU support for OpenMP offloading directives into Clang. In *LLVM-HPC '15: Proc. of the Second Workshop on LLVM Compiler Infrastructure in HPC*. Austin, TX, (Nov. 2015), 1–11.

[7] Joshua Hoke Davis, Christopher Daley, Swaroop Pophale, Thomas Huber, Sunita Chandrasekaran, and Nicholas J. Wright. 2021. Performance assessment of OpenMP compilers targeting NVIDIA V100 GPUs. In *Proc. WACCPD 2020, Int'l Workshop on Accelerator Programming Using Directives*. Springer International Publishing, (Nov. 2021), 25–44.

[8] Vassilios V. Dimakopoulos, Elias Leontiadis, and George Tzoumas. 2003. A portable C compiler for OpenMP v.2.0. In *Proc. EWOMP 2003, 5th European Workshop on OpenMP*. Aachen, Germany, (Sept. 2003), 5–11.

[9] Khronos OpenCL Working Group. 2020. The OpenCL 3.0 Specification. (Sept. 2020).

[10] Ilias Kleftakis and Vassilios V. Dimakopoulos. 2021. Experiences with task-based programming using cluster nodes as OpenMP devices. In *HPCS 2020/2021, 18th Int'l Conference on High Performance Computing and Simulation*. Barcelona, Spain, (Mar. 2021).

[11] Andreas Kurth, Alessandro Capotondi, Pirmin Vogel, Luca Benini, and Andrea Marongiu. 2018. HERO: an open-source research platform for HW/SW exploration of heterogeneous manycore systems. In *Proc. ANDARE 2018, 2nd Workshop on Autotuning and Adaptivity Approaches for Energy Efficient HPC Systems*. Limassol, Cyprus, (Nov. 2018).

[12] Andreas Kurth, Koen Wolters, Björn Forsberg, Alessandro Capotondi, Andrea Marongiu, Tobias Grosser, and Luca Benini. 2020. Mixed-Data-Model Heterogeneous Compilation and OpenMP Offloading. In *Proc. CC 2020, 29th Int'l Conference on Compiler Construction*. San Diego, CA, USA, 119–131.

[13] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44, 4, (Feb. 2009), 101–110.

[14] Martina Lofqvist and José Cano. 2020. Accelerating deep learning applications in space. In *Proc. Small Satellite Conference, Pre-Conference Workshop Session IV: Advanced Concepts II* Article 21. (July 2020).

[15] Gaurav Mitra, Eric Stotzer, Ajay Jayaraj, and Alistair P. Rendell. 2014. Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture. In *Proc. IWOMP 2014, 10th Int'l Workshop on OpenMP*. Salvador, Brazil, (Sept. 2014), 202–214.

[16] Mohebbanaaz, Y. Padma Sai, and L.V. Rajani Kumari. 2022. Cognitive assistant DeepNet model for detection of cardiac arrhythmia. *Biomedical Signal Processing and Control*, 71, 103221.

[17] Gabriel Noaje, Christophe Jaillet, and Michaël Krajecki. 2011. Source-to-source code translator: OpenMP C to CUDA. In *Proc. HPCC 2011, IEEE Int'l Conference on High Performance Computing and Communications*. Banff, AB, Canada, (Sept. 2011), 512–519.

[18] NVIDIA. 2022. *CUDA C++ Programming Guide*. NVIDIA Corporation, (Mar. 2022).

[19] NVIDIA. 2022. NVIDIA Jetson Linux Developer Guide, 32.7.1 Release. https://docs.nvidia.com/jetson/l4t/index.html. (Apr. 2022).

[20] Marcio M. Pereira, Rafael C. F. Sousa, and Guido Araujo. 2017. Compiling and optimizing OpenMP 4.x programs to OpenCL and SPIR. In *Proc. IWOMP 2017, 13th International Workshop on OpenMP*. Stony Brook, NY, USA, (Sept. 2017).

[21] Vishwani Sati, Sergio Márquez Sánchez, Niloufar Shoeibi, Ashish Arora, and Juan M. Corchado. 2020. Face detection and recognition, face emotion recognition through NVIDIA Jetson Nano. In *Proc. ISAMI 2020, 11th Int'l Symposium on Ambient Intelligence*. Springer International Publishing, Cham, (Sept. 2020), 177–185.

[22] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. 2020. Using parallel programming models for automotive workloads on heterogeneous systems – a case study. In *Proc. PDP 2020, 28th Euromicro Int'l Conference on Parallel, Distributed and Network-Based Processing*. Västerås, Sweden, (Mar. 2020), 17–21.

[23] Sebastián Valladares, 5erly Toscano, Rodrigo Tufiño, Paulina Morillo, and Diego Vallejo-Huanga. 2021. Performance evaluation of the NVIDIA Jetson Nano through a real-time machine learning application. In *Proc. IHSI 2021, 4th Int'l Conference on Intelligent Human Systems Integration*. (Jan. 2021), 343–349.

[24] Yi Yang and Huiyang Zhou. 2014. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '14). Orlando, Florida, USA, (Feb. 2014), 93–106.