Ensuring Memory Consistency in Heterogeneous Systems Based on Access Mode Declarations

Ludovic Henrio, C Kessler, Lu Li

Ensuring Memory Consistency in Heterogeneous Systems Based on Access Mode Declarations

Ludovic Henrio
Université Côte d'Azur, CNRS, I3S, France.
ludovic.henrio@cnrs.fr

C. Kessler and Lu Li
University of Linköping, Sweden
firstname.lastname@liu.se

***Abstract*—Running a program on disjoint memory spaces requires to address memory consistency issues and to perform transfers so that the program always accesses the right data. Several approaches exist to ensure the consistency of the memory accessed, we are interested here in the verification of a declarative approach where each component of a computation is annotated with an access mode declaring which part of the memory is read or written by the component. The programming framework uses the component annotations to guarantee the validity of the memory accesses. This is the mechanism used in VectorPU, a C++ library for programming CPU-GPU heterogeneous systems and this article proves the correctness of the software cache-coherence mechanism used in the library. Beyond the scope of VectorPU, this article can be considered as a simple and effective formalisation of memory consistency mechanisms based on the explicit declaration of the effect of each component on each memory space.**

***Index Terms*—Memory consistency, CPU-GPU heterogeneous systems, data transfer, software caching, cache coherence**

I. Introduction

Heterogeneous computer systems, such as traditional CPU-GPU based systems, often expose disjoint memory spaces to the programmer, such as main memory and device memory, with the need to explicitly transfer data between these. The different memories usually require different memory access operations and different pointer types. Also, encoding memory transfers as message passing communications leads to low-level code that is more error-prone. A commonly used software technique to abstract away the distributed memory, the explicit message passing, and the asymmetric memory access mechanisms consists in providing the programmer with an object-based shared memory emulation. For CPU-GPU systems, this can be done in the form of special data-containers, which are generic, STL-like data abstractions such as `vector<...>` that wrap multi-element data structures such as arrays. These data-container objects internally perform transparent, coherent software caching of (subsets of) accessed elements in the different memories so they can be reused (as long as not invalidated) in order to avoid unnecessary data transfers. Such data-containers (sometimes also referred to as "smart" containers as they can transparently perform data transfer and memory allocation optimizations [6]) are provided in a number of programming frameworks for heterogeneous systems, such as StarPU [1] and SkePU [6], [7]. StarPU is a C-based library that provides API functions to define multi-variant tasks for dynamic scheduling where the data containers are used for modeling the operand data-flow among the dynamically scheduled tasks. SkePU defines device-independent multi-backend skeletons like map, reduce, scan, stencil etc. where operands are passed to skeleton calls within data containers.

VectorPU [10] is a recent C++-only open-source programming framework for CPU-GPU heterogeneous systems. VectorPU relies on the specification of *components*, which are functions that contain kernels for execution on either CPU or GPU. Programming in VectorPU is thus not restricted to using predefined skeletons like SkePU, but leads to more high-level and more concise code than StarPU. Like StarPU, VectorPU requires the programmer to annotate each operand of a component with the access mode (read, write, or both) including the accessing unit (CPU, GPU), and uses smart data containers for automatic transparent software caching based on this access mode information.

The implementation of VectorPU makes excessive use of static metaprogramming; this provides a light-weight realization of the access mode annotations and of the software caching, which only require a standard C++ compiler. Emulating these light-weight component and access mode constructs without additional language and compiler support (in contrast to, e.g., OpenACC or OpenMP), leads however to some compromises in static analyzability. In particular, VectorPU has no explicit type system for the access modes, as these are not known to the C++ compiler.

In this paper, we investigate how to formalize access modes and data transfers in CPU-GPU heterogeneous systems and prove the correctness of the software cache coherence mechanism used in VectorPU. The contributions of this paper are:

- We introduce a simple effect system modeling the semantics of memory accesses and communication in a CPU-GPU heterogeneous system, and define a small calculus expressing different memory accesses and their composition across program traces.
- We express VectorPU operations as higher-level statements that can be translated into the core calculus, and show that, if all memory accesses are performed through VectorPU operations, the memory cannot reach an inconsistent state and all memory accesses succeed.

This paper is organized as follows: Section II reviews VectorPU as far as required for this paper, for further information we refer to [10]. Section III provides our formalization of VectorPU programs and their semantics, and proves that the

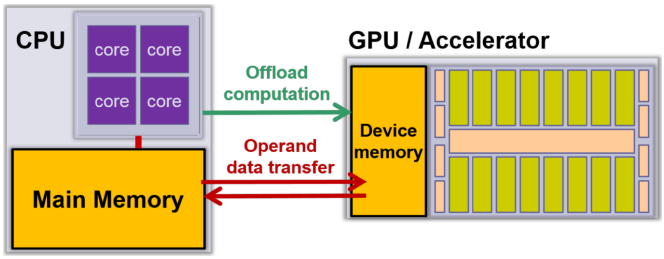

Fig. 1. A GPU-based system with distributed address space

coherence mechanism used in VectorPU is sound. Section IV discusses related work, and Section V concludes.

II. VectorPU

In heterogeneous systems with separate address spaces, for example in many GPU-based systems, a general-purpose processor (CPU) with direct access to main memory is connected by some network (e.g., PCIe bus) to one or several accelerators (e.g., GPUs) each having its own device memory, see Figure 1. Native programming models for such systems such as CUDA typically expose the distributed address spaces to the programmer, who has to write explicit code for data transfers and device memory management. Often, programs for such systems even must be organized in multiple source files as different programming models and different toolchains are to be used for different types of execution unit. This enforces a low-level programming style. Accordingly, a number of single-source programming approaches have been proposed that abstract away the distribution by providing a virtual shared address space. Examples include directive-based language extensions such as OpenACC and OpenMP4.5, and C++-only approaches such as the library-based skeleton programming framework SkePU [6] and the recent macro-based framework *VectorPU*, which we focus on in this paper as a case study.

VectorPU [10] is an open-source[1] lightweight C++-only high-level programming layer for writing single-source heterogeneous programs for Nvidia CUDA GPU-based systems.

Aggregate operand data structures passed into or out of function calls are to be wrapped by special data containers known to VectorPU. VectorPU currently provides one generic data container, called `vector<...>`, with multiple variants that eliminate the overhead of managing heterogeneity and distribution when not required (e.g., when no GPU is available). `vector<...>` inherits functionality from STL `vector` and from Nvidia Thrust `vector`, and wraps a C++ array allocated in main memory. VectorPU automatically creates on demand copies of to-be accessed elements in device memory and keeps all copies coherent using a simple coherence protocol, data transfers are only performed when needed.

VectorPU programs are organized as a set of C++ functions, some of which might internally use device-specific programming CUDA constructs[2] while others are expected to execute on the host, using one or possibly multiple cores. VectorPU *components* are functions that are supposed to contain (CPU or device) kernel functionality and for which operands are passed as VectorPU data container objects. Components and the types of execution units that access their operands are implicitly marked by annotating the operands of the function, either at a call of the function or for the formal parameters in the function's declaration, with VectorPU *access mode specifiers*.[3]

TABLE I
VectorPU access mode annotations for a parameter [10]

Access Mode	On Host	On Device
Read pointer	`R`	`GR`
Write pointer	`W`	`GW`
Read and Write pointer	`RW`	`GRW`
Read Iterator	`RI`	`GRI`
Read End Iterator	`REI`	`GREI`
Write Iterator	`WI`	`GWI`
Write End Iterator	`WEI`	`GWEI`
Read and Write Iterator	`RWI`	`GRWI`
Read and Write End Iterator	`RWEI`	`GRWEI`
Not Applicable	`NA`	`NA`

The access mode specifiers, such as `R` (read on CPU), `W` (write on CPU), `RW` (update, i.e., both read and write, on CPU), `GR` (read on GPU) and so forth, are available both as annotations of function signatures[4] and as C++ preprocessor macros that expand at compilation into (possibly, device-specific) C++ pointer expressions and side effects that allow to generate device specific access code and use device-specific pointer types for the chosen execution unit. For instance, `GW(x)` expands to a GPU pointer to the GPU device copy of `x`, which might be dereferenced for GPU writing accesses to `x`, such as the GPU code: `*( GW(x) + 2 ) = 3.14`. `GWI(x)` evaluates to an Thrust-compatible iterator onto the GPU device copy of `x`, and `WEI(x)` to an iterator-end reference to the last element of `x` on CPU side. It is also possible to specify partial access of a `vector` instead of the entire `vector` data structure[5]. Table I summarizes the access mode annotations currently defined for VectorPU.

The following example (adapted from [10]) of a CUDA kernel wrapped in an annotated function `bar` shows the use of VectorPU access mode annotations at function declaration:

```
// Example (annotations at function declaration):
__global__
void bar ( const float *x [[GR]], float *y [[GW]],
                  float *z [[GRW]], int size )
{ ... CUDA kernel code ... }
```

Here, the operand array pointed to by `x` may be read (only) by the GPU within `bar`, operand array `y` may be written (only)

[1] http://www.ida.liu.se/labs/pelab/vectorpu, https://github.com/lilu09/vectorpu

[2] VectorPU allows to directly annotate a CUDA kernel function, in addition to annotating its C++ wrapper function.

[3] In contrast to e.g. SkePU [7] which overloads element access and iterator operations so that monitored accesses are also possible on demand in non-componentized (i.e., ordinary C++) CPU code, VectorPU only relies on access mode annotations to perform lazy data transfer, not knowing when data is going to be accessed inside a component.

[4] The current VectorPU prototype implementation does not (yet) type-check access-mode annotations in signatures of externally defined functions.

[5] The current VectorPU implementation does not (yet) support coherence for *overlapping* intervals of elements resulting from multiple (partial) accesses some of which (may) access the same element. A solution for this problem has been described for SkePU smart containers by Dastgeer [6].

by the GPU, and operand array `z` may be read or written (or both) by the GPU. When calling `bar`, the first three operands need to be passed as VectorPU `vector` container objects. The `size` formal parameter is a scalar (not a data container), so it will be available on GPU on a copy-in basis but no coherence will be provided for it by VectorPU.

It is also possible to put the annotations into a call, and hence characterize a function as a VectorPU component:

```
// declare a CPU function:
void foo ( const float *x, float *y,
            float *z, int size );

// declare three vectors:
vectorpu::vector<float> vx(100), vy(100), vz(100);

// call to VectorPU annotated function foo:
foo ( R( vx ), W( vy ), RW( vz ), size ) ;
```

Here, the access mode specifiers and the resulting coherence policy only apply to that particular invocation of `foo`, while other invocations of `foo` might use different access mode specifiers. The following example shows how to use iterators:

```
vectorpu::vector<My_Type> vx(N);
std::generate( WI(vx), WEI(vx), RandomNumber );
thrust::sort( GRWI(vx), GRWEI(vx));
std::copy( RI(vx), REI(vx),
           ostream_iterator<My_Type>(cout, ""));
```

where `std::generate` is a CPU function filling a section between two addresses with values (here, random numbers), and `thrust::sort` denotes the GPU sorting functionality. provided by the Nvidia Thrust library.

Using only available C++(11) language features, VectorPU provides a flexible unified memory view where all data transfer and device memory management is abstracted away from the programmer. Nevertheless, its efficiency is on par with that of handwritten CUDA code containing explicit data movement and memory management code [10]. In particular, the VectorPU prototype was shown to achieve 1.4x to 13.29x speedup over good quality code using Nvidia's *Unified Memory* API on several machines ranging from laptops to supercomputer nodes, with Kepler and Maxwell GPUs. For a further discussion of VectorPU features, such as specialized versions of `vector`, for descriptions of how to use VectorPU together with lambda expressions e.g. to express skeleton computations, and for further experimental results we refer to [10].

III. Formalization

In this section we provide a minimal calculus to reason on the memory operations that can exist in a framework that deals with memory consistency like VectorPU. We first define a set of effects operations can have on the consistency of the memory. Then we define a small calculus expressing different memory accesses and their composition into complex procedures. Finally, we express VectorPU operations as higher-level statements that can be translated into the core calculus presented before, and show that if all memory accesses are annotated correctly through VectorPU annotations the program cannot try to access an invalid data and the memory spaces are put in coherence when needed. We also show that VectorPU tracks the validity status of the memory adequately. In this section we abstract away the values stored in memory and we do not deal with any form of aliasing. A more precise analysis of effects and aliasing is out of the scope of this paper, it could be for example inspired from [11]. We place ourself in a simplified setting where each variable is hosted in exactly two memory locations, e.g. a CPU (main) memory and a GPU memory location, but the work could be extended to multiple memory locations without any major difficulty.

A. An effect system for consistency between memory locations

We start from a simple effect system, it expresses the effect of writing or reading a memory location on the consistency status of the memory. Each location is either in *valid* state when it holds a usable data or *invalid* state when the value at the location is not valid anymore.

We express five operations: reading, writing, *Push* for uploading the local memory location into the other one, and *Pull* for the contrary. *Noop* is an operation that does nothing.

$$E ::= \mathit{Push} \mid \mathit{Pull} \mid r \mid w \mid \mathit{Noop}$$

The effect of these operations express their requirements and effects on a single memory location. We express below the semantics of each of the operations on the consistency status of the concerned memory location. The *memory status of a variable* is a pair of the status of its locations, where each status is either V for valid or I for invalid. The first element is the status of the local memory, and the second one is the status of the remote memory. For example, for a program running on a CPU while the remote memory is a GPU, a status (V, I) means that the memory is valid and can be read on the CPU, but is invalid on the GPU and should be transferred before being usable there.

Each operation has a signature in the sense that it may require a certain memory status and will produce another memory status. The signature of each operation is expressed below. We use variables X, Y, Z, T that are considered as universally quantified in each rule. They can be instantiated with either V or I.

$$\mathit{Push} : (V, X) \mapsto (V, V) \qquad \mathit{Pull} : (X, V) \mapsto (V, V)$$

$$r : (V, X) \mapsto (V, X) \qquad w : (X, Y) \mapsto (V, I)$$

$$\mathit{Noop} : (X, Y) \mapsto (X, Y)$$

These signatures are effects expressing that r is a reading operation requiring validity of data and ensuring not to modify it, the distant status is unchanged; w is a writing operation that modifies data locally but do not require validity, they invalidate the remote memory. *Push* uploads the local memory and thus makes valid the distant memory; it requires that the data is locally valid, and *Pull* is the symmetrical operation.

An additional operation could be defined: an rw operation would represent a read and/or write access, it would both require data validity and invalidate remote status: $(V, X) \mapsto (V, I)$. This operation is however not needed here.

B. A language for modelling consistency and effects

We now create a core calculus to be able to reason on programs involving sequences of effects on different memory locations. x, y range over variables and we introduce statements manipulating variables. We use sequence and simple loops and conditionals. Operations with effects now apply to a variable; the *rem*$(E\ x)$ is a remote operation on the remote memory. For example, a GPU procedure writing x and reading y would correspond to the pseudo-code: *rem*$(w\ x)$; *rem*$(r\ y)$. Statements S are defined as:

$$S ::= E\ x \mid \textit{rem}(E\ x) \mid S; S' \mid \textit{While}(\textit{cond})S \mid \textit{if}(\textit{cond})\ S\ \textit{else}\ S'$$

where $E\ x$ denotes some effect E on variable x, with $E \in \{r, w, \textit{Push}, \textit{Pull}, \textit{Noop}\}$.

We are interested in conditionals dealing with the validity status of the variables. Other conditionals are expressed as a generic binary operator $\oplus$ but other operators with different arities could be added as well:

$$\textit{cond} ::= \textit{isValid}\ x \mid \textit{remIsValid}\ x \mid x \oplus y$$

where *isValid* x and *remIsValid* x denote checks of the validity status flag of the local and remote location of x, respectively.

We now define a small step operational semantics for our core calculus. It relies on the validity status of variables, recorded in a store σ mapping variable names to validity pairs. Semantics is written as a transition relation between pairs consisting of a statement and a store: (S, σ). The sequencing operator ; is associative with *Noop* as a neutral element. Consequently each non-empty sequence of instruction can be rewritten as $S; S'$ where S is neither a sequence nor *Noop*. $\sigma[x \mapsto (X, Y)]$ is the update operation on maps.

The semantics is presented in Figure 2. Like in the previous section, we use validity variables X, Y, Z, T that are universally quantified in each rule. The first four rules present the evaluation of conditional statements, we suppose additional rules exist for evaluating $\oplus$[6]. The next rule applies an effect on a variable x updating the validity store, and the REMOTE EFFECT rule applies an effect occurring on the distant memory, it applies the symmetric of the effect to the variable. Note that *Push* is the symmetric of *Pull* and we could have removed one of those two operations without loss of generality. The last rules are standard ones for *if* and *while* statements.

Initial state: To evaluate a sequence of statements S using the variables $\mathrm{vars}(S)$, we put it in a configuration with an initial store where data is hosted on the CPU and all variables are initially mapped to (V, I): $\sigma_0 = (x \mapsto (V, I))^{x \in \mathrm{vars}(S)}$.

A configuration is *reachable* if it is possible to obtain this configuration starting from the initial configuration and applying any number of reductions: (S, σ) is reachable if $(S, \sigma_0) \to^* (S', \sigma)$ where $\to^*$ is the reflexive transitive closure of $\to$. We write $(S, \sigma) \not\to$ and say that the configuration is *stuck* if no reduction rule can be applied on (S, σ).

[6]We are only interested in cache coherence properties, we thus suppose that evaluation of $\oplus$ always succeed, and in particular variables accessed by the operation are specified as a r operation preceding the condition.

Property 1 (Progress)**.** *A configuration is stuck if the validity status of the accessed variable is incompatible with the effect to be applied*[7]*:*

$$\begin{aligned}&(S, \sigma) \not\to \iff\\ &S = E\ x; S' \wedge \sigma(x) = (X, Y) \wedge E : (X', Y') \mapsto (Z, T)\\ &\quad \wedge \textit{ there is no unification between } (X, Y) \textit{ and } (X', Y')\\ &\vee S = \textit{rem}(E\ x); S' \wedge \sigma(x) = (X, Y) \wedge E : (X', Y') \mapsto (Z, T)\\ &\quad \wedge \textit{ there is no unification between } (X, Y) \textit{ and } (Y', X')\end{aligned}$$

Note that this supposes that $\oplus$ always succeeds.

Proof sketch. By case analysis on the first statement of S, there is always one rule applicable provided the premises of the rule can be evaluated. In the case of the last four rules this requires the evaluation of *cond*. If $\oplus$ always succeeds then *cond* can always be evaluated. The only case remaining is if there is no unification possible between the effect of an operation and the current validity status of the affected variable, this concerns the rule EFFECT and REMOTE EFFECT and corresponds to the two cases expressed in the theorem. □

Property 2 (Safety)**.** *A state is said to be* unsafe *if at least one variable is mapped to (I, I). It is impossible to reach an unsafe state from the initial state.*

Proof sketch. Unsafe states are avoided because of the effects of operations: only effect rules modify the store and no effect can reach (I, I), except *Noop* starting from (I, I). □

Example: wx; *rem*$(r\ x)$ cannot be fully evaluated. Indeed, $(wx; \textit{rem}(r\ x), (x \mapsto (V, I))) \to (\textit{rem}(r\ x), (x \mapsto (V, I)))$, but *rem*$(r\ x)$ requires that x is mapped to (X, V) for some X which is not the case. However if we add a *Push* operation to ensure the validity of the read memory the program $w\ x; Push; \textit{rem}(r\ x)$ can be reduced:

$$\begin{aligned}&(wx; \textit{rem}(r\ x), (x \mapsto (V, I)))\\ &\quad \to (\textit{Push}; \textit{rem}(r\ x), (x \mapsto (V, I)))\\ &\quad \to (\textit{rem}(r\ x), (x \mapsto (V, V))) \to (\textit{Noop}, (x \mapsto (V, V)))\end{aligned}$$

C. Declaring access modes and adding an abstraction layer

The calculus defined above only considers simple memory locations and directly manipulates them. But VectorPU and similar libraries manipulate structures representing the memory. For example, VectorPU vectors act as an abstract representation of a set of memory locations. In this section, we add a declaration and abstraction layer to the calculus to represent the access mode declarations that will trigger data transfers according to the consistency mechanism. This abstraction layer is also a necessary first step to the modelling of array structures that we will present in Section III-E. Indeed, in array structures, the validity status of the array is abstracted away by a single validity status pair. Then a dynamic abstraction of the consistency status of the memory can be used. More technically, the abstraction and declaration layer relies on two principles:

[7]We say that there is no unification between X and Y if one of the two variables must have the value V, and the other one the value I. This relation is extended to pairs of variables.

$$\textsc{valid}\ \frac{\sigma(x) = (V, X)}{[\![\mathit{isValid}\ x]\!]_\sigma = \texttt{True}} \qquad \textsc{invalid}\ \frac{\sigma(x) = (I, X)}{[\![\mathit{isValid}\ x]\!]_\sigma = \texttt{False}}$$

$$\textsc{rem-valid}\ \frac{\sigma(x) = (X, V)}{[\![\mathit{remIsValid}\ x]\!]_\sigma = \texttt{True}} \qquad \textsc{rem-invalid}\ \frac{\sigma(x) = (X, I)}{[\![\mathit{remIsValid}\ x]\!]_\sigma = \texttt{False}}$$

$$\textsc{Effect}\ \frac{\sigma(x) = (X, Y) \qquad E : (X, Y) \mapsto (Z, T)}{(E\ x; S, \sigma) \to (S, \sigma[x \mapsto (Z, T)])}$$

$$\textsc{Remote Effect}\ \frac{\sigma(x) = (X, Y) \qquad E : (Y, X) \mapsto (Z, T)}{(\mathit{rem}(E\ x); S, \sigma) \to (S, \sigma[x \mapsto (T, Z)])}$$

$$\textsc{While-True}\ \frac{[\![\mathit{cond}]\!]_\sigma}{(\mathit{While}(\mathit{cond})S; S', \sigma) \to (S; \mathit{While}(\mathit{cond})S; S', \sigma)} \qquad \textsc{While-False}\ \frac{\neg[\![\mathit{cond}]\!]_\sigma}{(\mathit{While}(\mathit{cond})S; S', \sigma) \to (S', \sigma)}$$

$$\textsc{If-True}\ \frac{[\![\mathit{cond}]\!]_\sigma}{((\mathit{if}(\mathit{cond})\ S\ \mathit{else}\ S'); S'', \sigma) \to (S; S'', \sigma)} \qquad \textsc{If-False}\ \frac{\neg[\![\mathit{cond}]\!]_\sigma}{((\mathit{if}(\mathit{cond})\ S\ \mathit{else}\ S'); S'', \sigma) \to (S'; S'', \sigma)}$$

Fig. 2. Operational semantics of validity status.

- Each variable x has an abstract variable $x^\#$ that represents it. In this section there is a single variable for each representative, but when we deal with arrays we will have a single representative for the whole array.
- It is safe to "forget" that one memory space holds a valid copy of the data if the other memory space has a valid one. In other words, (V, I) (resp. (I, V)) is a safe abstraction of (V, V) and we denote $(V, I) \leq (V, V)$ (resp. $(I, V) \leq (V, V)$). Of course, we have $(X, Y) \leq (X, Y)$ for all X and Y.

a) Syntax: We now define access mode declarations:

$$\begin{aligned}\mathcal{M} ::= {} & R\ x^\# \mid W\ x^\# \mid RW\ x^\# \mid \\ & \mathit{rem}(R\ x^\#) \mid \mathit{rem}(W\ x^\#) \mid \mathit{rem}(RW\ x^\#) \mid \\ & \mathcal{M} \wedge \mathcal{M}' \quad \text{(where variables in } \mathcal{M} \text{ and } \mathcal{M}' \text{ are disjoint)}\end{aligned}$$

These access modes declare the kind of access (read R, write W, or read and/or write RW) that can be performed on the variable x represented by $x^\#$. In a set of access mode declarations the same variable cannot appear twice. There exist declared access modes for local accesses and for the remote memory space.

A program is a sequence of calls to functions or components (i.e., statements accessing only real variables) each protected by an access mode declaration:

$$\mathcal{P} ::= \mathcal{M}_1\{S_1\}; \mathcal{M}_2\{S_2\}; \ldots$$

We write that $S \in S'$ if S is one statement inside S' (i.e. S is a sub-term of S').

We define below the semantics of these programs and specify well-declared program by comparing the statements they contain with the declared access modes. The semantics relies on the translation of the access mode declarations into consistency mechanisms with checks and data transfers triggered before each function execution.

b) Extension of statements to abstract variables: When evaluating a program, the store contains both real and abstract variables, and the existing statements have the same effect on the abstract variables as on the real ones. However one should notice that even if the effect is the same, the meaning of a statement acting on a real variable or on its representative is different: in our calculus, the effect on a variable is an abstraction of the real effect that involves side effects and data transfers. On the contrary, only the validity status of abstract variables is stored by the library: the effect triggered by an operation on an abstract variable is exactly what happens when VectorPU updates the validity status of its internal structures.

For example, a *Pull* operation on a real variable consists in transferring data from a remote memory space to the local one. We abstracted it by changing the local validity status. A *Pull* operation on an abstract variable only changes the validity status, no data transfer has to be done because abstract variables only need to be stored in one memory space. The validity status is stored in the CPU address space in VectorPU. Comparing the validity status of real memory and their representative allow us to reason formally on the correctness of the validity tracking performed by VectorPU.

As no data is accessed by the effects on abstract variables, they cannot create stuck configuration. We will not use $r\ x^\#$ as it does not change the validity status of variables. The statement that should get stuck in case of a read access is the read of *the real variable that cannot access a valid data*.

c) Semantics: Figure 3 defines the semantics of programs with access modes as a translation into the core calculus of Section III-B. This translation ensures that the validity status is correct and records the effect of the function on the abstract variable before running the function call that may read and write data (on the real variables). Similarly to the VectorPU library, the protected accesses can be considered as macros and the programs can be translated into the core syntax.

This encoding corresponds to the macros as they are implemented in VectorPU. It is indeed easy to check that VectorPU tracks the effects in the same way as our effect system does in the translation rules. These translation rules perform *Push* or *Pull* operations in order to ensure that the memory is in

$$[\![R\ x^{\#}]\!] = (\mathit{if}(\mathit{isValid}\ x^{\#})\ \mathit{Noop}\ \mathit{else}\ (\mathit{Pull}\ x; \mathit{Pull}\ x^{\#})) \qquad [\![\mathit{rem}(R\ x^{\#})]\!] = (\mathit{if}(\mathit{remIsValid}\ x)\ \mathit{Noop}\ \mathit{else}\ (\mathit{Push}\ x; \mathit{Push}\ x^{\#}))$$

$$[\![RW\ x^{\#}]\!] = (\mathit{if}(\mathit{isValid}\ x)\ \mathit{Noop}\ \mathit{else}\ (\mathit{Pull}\ x; \mathit{Pull}\ x^{\#})); w\ x^{\#}$$

$$[\![\mathit{rem}(RW\ x^{\#})]\!] = (\mathit{if}(\mathit{remIsValid}\ x)\ \mathit{Noop}\ \mathit{else}\ (\mathit{Push}\ x; \mathit{Push}\ x^{\#})); \mathit{rem}(w\ x^{\#}) \qquad [\![W\ x^{\#}]\!] = w\ x^{\#}$$

$$[\![\mathit{rem}(W\ x^{\#})]\!] = \mathit{rem}(w\ x^{\#}) \qquad [\![\mathcal{M}_1\{S_1\}; \mathcal{M}_2\{S_2\}; \ldots]\!] = [\![\mathcal{M}_1]\!]; S_1; [\![\mathcal{M}_2]\!]; S_2; \ldots$$

Fig. 3. Semantics of access modes and programs

a correct validity status for the read or write operation to be performed. When evaluating a program we create a store where the validity status of real and abstract variables are (V, I), corresponding to the fact that data is initially placed in one memory location; typically, in VectorPU, in the CPU memory space.

D. Well-declared Programs and their Properties

We now define formally what it means for an access mode declaration to be correct, i.e. to adequately specify the effect of a function. The principle is that each operation on a memory location must be declared on its representative. It is however possible to declare more read or RW accesses that what is done in practice, and one can declare a read and/or write access if only read or write is performed. Additionally, the annotation W denotes an *obligation* to write which allows the consistency mechanism to avoid any validity checks before running the function that will overwrite the data. To represent this concept, we need a first definition that states that an operation will be performed in all execution paths of a (bigger) statement. This definition formalises a classical static analysis concept that states that all branches of conditionals necessarily execute a given statement. It considers executions that run to completion and states that a given statement is necessarily evaluated in this execution.

Definition 1 (Occur in all execution paths)**.** *We state that a statement S* occur in all execution paths of S_0 *if, for any correct initial store σ_0, for all full reductions $(S_0, \sigma_0) \to (S_1, \sigma_1) \to \ldots \to (\mathit{Noop}, \sigma_n)$, there is an intermediate state (S_i, σ_i) such that $S_i = S; S''$ for some S''.*

Notice that an operation S may appear in some of the execution paths of S' if $S \in S'$: if $(S_0, \sigma_0) \to^* (S; S', \sigma)$ then $S \in S_0$.

Definition 2 (Well-declared program)**.** *A program $\mathcal{P}$ is* well-declared *if for all $\mathcal{M}\{S\}$ in $\mathcal{P}$ we have:*

- *Push $x \notin S$ and Pull $x \notin S$ (for any x),*
- *$w\ x \in S \implies (W\ x^{\#} \in \mathcal{M} \vee RW\ x^{\#} \in \mathcal{M})$,*
- *$r\ x \in S \implies (R\ x^{\#} \in \mathcal{M} \vee RW\ x^{\#} \in \mathcal{M})$,*
- *$W\ x^{\#} \in \mathcal{M} \implies w\ x$ occurs in all execution paths of S,*
- *Plus the same rules for remote operations.*

Note that a well-declared program does not perform synchronisation operations (*Push* or *Pull*) manually, these operations are only performed when evaluating the access mode declarations. Also each variable accessed by a well-declared function has an abstract representative in the corresponding declaration block.

A direct consequence of the definition above is that a well-declared program cannot access, in the same function, the same variable in both address spaces. This is in accordance with VectorPU where each function is entirely executed either on a CPU or on a GPU, the formalisation is a bit more generic on this aspect. This is expressed by the following property.

Property 3 (Localised access)**.** *For a well-declared program containing $\mathcal{M}\{S\}$, for any x, we cannot have $\mathit{rem}(E\ x) \in S$ and $E'\ x \in S$.*

We now state and prove the two major properties ensured by our formalisation. The first property ensures that the abstraction is correct relatively to the execution. This corresponds to the fact that VectorPU tracks adequately the validity status of the memory. This is expressed as a theorem that is similar to subject-reduction in type systems, it states that if the status of the abstract variables represent correctly the validity status of the real variables, then the abstraction is also correct after the execution of a well-declared function. Let us say that we have a *correct abstraction of the memory state* if for each real memory location, the abstract representative of this location has a validity status that is an approximation, in the sense of $\leq$, of the validity status of the real memory. The theorem below states that the execution of a well-declared function maintains the correctness of the memory state abstraction.

Theorem 1 (Subject reduction)**.** *Suppose $\mathcal{M}\{S\}$ is well-declared, we have:*

$$\forall x \in \mathrm{dom}(\sigma).\, \sigma(x^{\#}) \leq \sigma(x) \wedge ([\![\mathcal{M}\{S\}]\!], \sigma) \to^* (\mathit{Noop}, \sigma') \implies \forall x \in \mathrm{dom}(\sigma').\, \sigma'(x^{\#}) \leq \sigma'(x)$$

This property is extended by a trivial induction to the execution of a well-protected program $\mathcal{P}$ in an initial store $\sigma_0 = (x \mapsto (V, I))^{x \in \mathrm{vars}(\mathcal{P})}$.

Proof. Notice that $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma)$, and if $\sigma(x) = (V, I)$ or $\sigma(x) = (I, V)$ then $\sigma(x) = \sigma(x^{\#})$, else $\sigma(x) = (V, V)$. We reason on the read and write access that occur in the considered reduction. Each variable x is either read or written or not accessed (or read and written). For each case we compare the status of abstract and local variable, and in particular we consider the status of the reduction after executing the synchronisation code $[\![\mathcal{M}\{S\}]\!]$ and call σ_s the corresponding

store (note that $\sigma_s(x^\#) = \sigma'(x^\#)$). We detail operations on the local address space, cases for remote operations are similar:
• If x is written, we have: $([\![\mathcal{M}\{S\}]\!], \sigma) \rightarrow^* (w\ x; S', \sigma'') \rightarrow^* (\mathit{Noop}, \sigma')$. Whatever the initial value of $\sigma(x)$, we have $\sigma'(x) = (V, I)$. Two cases are possible:
(1) $W\ x^\# \in \mathcal{M}$ then the value cannot be read and we have $\sigma_s(x^\#) = (V, I)$. $\sigma'(x^\#) = \sigma'(x)$.
(2) $RW\ x^\# \in \mathcal{M}$ then a data-transfer (*Pull*) may occur. Knowing that $\sigma(x^\#) \leq \sigma(x)$, by a case analysis on $\sigma(x)$ and $\sigma(x^\#)$ we have: $\sigma_s(x^\#) = (V, I)$ and $\sigma_s(x) = (V, I)$ or (V, V). Whether x is read or not we have $\sigma'(x^\#) = \sigma'(x)$.
• If x is read but not written, its validity status is not changed.
(1) $R\ x^\# \in \mathcal{M}$. By a case analysis on $\sigma(x)$ and $\sigma(x^\#)$ we have: $\sigma_s(x) = (V, I)$ and $\sigma_s(x^\#) = (V, I)$, or $\sigma_s(x) = (V, V)$ and $\sigma_s(x^\#) = (V, I)$ or (V, V). Reading has no effect on validity status and in all cases we have $\sigma'(x^\#) \leq \sigma'(x) = \sigma_s(x)$.
(2) $RW\ x^\# \in \mathcal{M}$ then similarly to the case (2) above we have $\sigma_s(x^\#) = (V, I)$, additionally $\sigma'(x) = \sigma_s(x) = (V, I)$ or (V, V). In all cases $\sigma'(x^\#) \leq \sigma'(x)$.
• If x is not accessed but is in the declaration, the reasoning is the same as if it was only read. Note that the variable cannot be declared in write mode, $W\ x^\# \in \mathcal{M}$, by Definition 2. □

Finally, a well-declared program always runs to completion: it never tries to access an invalid memory location.

Theorem 2 (Progress for well-declared programs)**.** *If a program $\mathcal{P}$ is well-declared, then its execution cannot reach a stuck configuration.*

Proof. By Property 1, it is sufficient to prove that unification on the validity status is always possible. We consider a reduction $([\![\mathcal{M}\{S\}]\!], \sigma) \rightarrow^* (S, \sigma_s) \rightarrow^* \ldots$ similarly to the proof above.

By definition of well-declared programs and because of the signature of effects ($w\ x$ cannot be stuck), only two cases have to be analysed for the local operations (plus two similar cases for remote statements):

- *Pull* operations (on x and $x^\#$) in the translation of $R\ x^\#$ or $RW\ x^\#$. Unification requires that $\sigma(x) = (X, V)$ and $\sigma(x^\#) = (Y, V)$.
- $r\ x$ operation in the evaluation of S. Unification requires that $\sigma'(x) = (V, X)$ where σ' is the store in which the read access is to be evaluated.

Indeed, access mode declarations do not generate reading operations, and by definition function statements contain no *Push* or *Pull*.

Concerning the first case, because of Theorem 1, we have $\sigma(x^\#) \leq \sigma(x)$, and because of property 2 none of them is (I, I). By case analysis on the possible values of $\sigma(x^\#)$ and $\sigma(x)$, it is easy to show that $\sigma(x) = (X, V)$ and $\sigma(x^\#) = (Y, V)$ if we reach the two *Pull* statements that perform data transfers before the execution of the function.

Concerning read access, they should be verified by an induction on the reduction steps following the state (S, σ_s) showing that, for any variable x that is declared R or RW, in all states we have $\sigma'(x) = (V, X)$. Indeed, by the same analysis as in the proof of Theorem 1 we know that $\sigma'(x) = (V, X)$. Because of Property 3 no remote operation is possible on x and thus only $r\ x$ and $w\ x$ operations are possible on x, both maintain the invariant $\sigma'(x) = (V, X)$ for some X. □

Considering the example above of a variable written on the CPU, and then read on the GPU, a well-declared program encoding this behaviour would be $RW\ x^\#\{w\ x\}; \mathit{rem}(R\ x^\#)\{\mathit{rem}(r\ x)\}$. This code automatically generates the *Push* instruction that prevents the program from being stuck.

E. Effects and Access Mode Declarations for Arrays

In array structures, the validity status of the whole array is abstracted away by a single validity status pair. We extend the syntax for arrays as follows, $x[i]$ denotes the indexed access to an element of the array. More precisely the new operations on arrays and their elements are (we still have the previous operations on non-array and abstract variables):

$$S ::= \ldots \mid r\ x[i] \mid w\ x[i]$$

Synchronisation operations (*Push* and *Pull*) exist for arrays but the whole array is synchronised, and we write *Push* x and *Pull* x as above. All the elements of the array are represented by a single abstract variable: $x^\#$ represents the validity status of all $x[i]$.

The semantics of access mode declarations and programs is unchanged because synchronisation operations and access mode declarations do not concern array elements. The concept of well-protected programs must be adapted to the case of array structures, and more precisely to the fact that several memory locations are represented by a single abstract variable.

Definition 3 (Well-declared program with array access)**.** *A program $\mathcal{P}$ is* well-declared *if for all $\mathcal{M}\{S\}$ in $\mathcal{P}$, additionally to the rules of Definition 2, we have:*

- $w\ x[i] \in S \implies (W\ x^\# \in \mathcal{M} \vee RW\ x^\# \in \mathcal{M})$,
- $r\ x[i] \in S \implies (R\ x^\# \in \mathcal{M} \vee RW\ x^\# \in \mathcal{M})$,
- $W\ x^\# \in \mathcal{M} \implies \forall i \in \mathrm{range}(x).w\ x[i]$ *occurs in all execution paths of S,*
- *Plus the same rules for remote operations.*

Where $\mathrm{range}(x)$ *is the set of valid indexes for an array x.*

The other properties are expressed similarly and both *subject-reduction*, Theorem 1, and *progress*, Theorem 2, are still valid. The only change is the "correct abstraction of the memory state" criteria that becomes $\forall x \in \mathrm{vars}(S). \forall i \in \mathrm{range}(x). \sigma(x^\#) \leq \sigma(x[i])$ instead of $\forall i \in \mathrm{range}(x)$ for arrays. The proofs are similar except in the case of $W\ x$ declarations where the fact that all elements of the array must be written is necessary to ensure that no element is in the status (I, V) (which could not be safely represented by (V, I)) at the end of the function execution. If we focus on the proof of Theorem 1, case "x is written, sub-case (1) we have $\sigma'(x^\#) = (V, I)$ which is a safe abstraction because *all elements have been written*, and thus $\sigma'(x[i]) = (V, I)$ for all i. If one element j was not written, we could have had $\sigma'(x[i]) = (I, V)$ which would invalidate the theorem.

IV. A few related works

Most of the verification works related to memory consistency focus on coherence protocols and/or weak memory models. Among them, one could cite [8], a formal specification of a caching algorithm, and its verification in TLA [9]. These works shows the difficulty on reasoning on memory coherency, but also that specifications in these models should rely on a few simple instructions on the type of memory accessed, a bit similarly to this proposal. Coherence protocols have also been verified using CCS specifications [2]. These various works are quite different from the approach presented in this paper because we rely here on a declarative approach for memory accesses: the programmer declares the kind of memory accesses performed by a component, and the consistency mechanism ensures that each component accesses a valid memory space.

More recently, and adopting a more language oriented approach, Crary and Sullivan [5] designed a calculus for expressing ordering of memory accesses in weak memory models, however we are interested here in a much simpler problem where memory access is somehow sequential and clearly identified. Even a extension of this work for parallel processes would result in a simpler model than the ones that exist for weak memory models because of the explicit consistency points introduced in the execution by the start/end of each function.

The closest work to ours are probably [4] that define a memory access calculus similar to ours and prove the correctness of a generic cache coherence protocol expressed as part of the semantics of the calculus. Compared to this work, we are interested in explicit statements on memory accesses and thus the cache consistency is partially ensured by the programmer annotations, making the approach and the properties proven significantly different. Some aspects of the approaches could however been made more similar, e.g. by extending our work to more than two address spaces or adopting a different syntax. However our problem and formalisation are quite simpler, and we believe easier to read, while sufficient for our study. The same authors also designed a formal model written in Maude [3] to better understand the possible optimisations and the impact of the memory organisation on performance in the context of cache coherent multicore architectures. This could be an interesting starting point for future works, especially if we extend our work to better model the performance aspects of VectorPU and want to reason formally on the improved performance obtained by the library.

V. Conclusion and future works

In this article we provided a formal approach to verify the consistency of the memory accesses in heterogeneous computer systems made of two memory spaces. We formalise the operations of memory accesses and memory synchronisation between the two memory spaces and prove that a program adequately annotated with informations on the memory accesses always access valid memory spaces and tracks correctly which of the memory space contains the up-to-date data.

The practical result is that we can verify the coherency mechanism used by the VectorPU library and ensure that, additionally to the significant performance benefits of the approach, the VectorPU mechanisms is correct and ensures the consistency of the memory accesses.

We envision several extensions to this work, the most promising is the study of the operations made on overlapping arrays. The current implementation of VectorPU supposes that the annotated memory accesses deal with disjoint memory locations, it does not take into account overlapping arrays. Designing an extension of the library that could deal safely with overlapping array is one of the future direction we would like to pursue. Additionally, the current paper only deals with two memory spaces, the extension to many memory spaces seem relatively simple but the mechanism dealing with memory transfers between several memory locations becomes a bit more complex; its formalisation should be similar. Finally, we are interested in the application of our approach to the verification of other frameworks. Indeed VectorPU uses the most primitive cache coherence protocol, the VI-protocol. More elaborated coherence protocols like MSI or MESI introduce additional states where the number of readers has to be tracked for example. Verifying such framework would require a modification of our abstract state representation and a modification of the access mode translational semantics.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, February 2011.

[2] Manuel Barrio-Solórzano, M. Encarnación Beato, Carlos E. Cuesta, and Pablo de la Fuente. Formal verification of coherence for a shared memory multiprocessor model. In Victor Malyshkin, editor, *Parallel Computing Technologies*, pages 17–26. Springer Berlin Heidelberg, 2001.

[3] Shiji Bijo, Einar Broch Johnsen, Ka I. Pun, and Silvia Lizeth Tapia Tarifa. A maude framework for cache coherent multicore architectures. In Dorel Lucanu, editor, *Rewriting Logic and Its Applications*, Cham, 2016. Springer International Publishing.

[4] Shiji Bijo, Einar Broch Johnsen, Ka I. Pun, and S. Lizeth Tapia Tarifa. An operational semantics of cache coherent multicore architectures. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1219–1224, New York, NY, USA, 2016. ACM.

[5] Karl Crary and Michael J. Sullivan. A calculus for relaxed memory. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 623–636, New York, NY, USA, 2015. ACM.

[6] Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for GPU-based systems. *Int. J. of Par. Progr.*, 44(3):506–530, June 2016.

[7] Johan Enmyren and Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, Maryland, USA*, pages 5–14. ACM, September 2010. doi: 10.1145/1863482.1863487.

[8] Rob Gerth. Sequential consistency and the lazy caching algorithm. *Distributed Computing*, 12(2):57–59, May 1999.

[9] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in TLA. *Distributed Computing*, 12(2):151–174, May 1999.

[10] Lu Li and Christoph Kessler. VectorPU: A generic and efficient data-container and component model for transparent data transfer on GPU-based heterogeneous systems. In *Proc. PARMA-DITAM'17, ACM.*, 2017.

[11] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Type and Effect Systems*, pages 283–363. Springer Berlin Heidelberg, 1999.