Heterogeneous Programming Library Programming Manual

HPL Team

August 23, 2015

Contents

1 Introduction

The Heterogeneous Programming Library (HPL) provides a programming environment for C++ whose aim is to maximize the programmability of heterogeneous systems while allowing low level control and performance similar to that of lower level approaches. The library is built on two key concepts:

- **Arrays** : special datatypes that can be used both in the main program, that is executed in a regular CPU, and in the code that runs in the heterogeneous devices.

- **Kernels** : functions that can run in any device. They can be written

  - in a language embedded in C++ provided by HPL. The library translates these kernels to OpenCL, enabling the execution in a wide range of devices. This language is extremely similar to regular C++ and it is defined in Sect. 4.2.
  - in regular OpenCL C. The procedure to use standard OpenCL kernels is explained in Sect. 4.3.3.

Users should define the data that they want to use in the kernels as Arrays. These Arrays can then be used as arguments in the host invocations to the kernel functions. This can be seen in Fig. 1, which shows the full code needed in HPL to execute the SAXPY function $Y = \alpha X + Y$ in an accelerator, where X and Y are vector and α is a scalar.

```
1  #include "HPL.h"
2
3  using namespace HPL;
4
5  void saxpy(Array<float,1> y, Array<float,1> x, Float alpha)
6  {
7      y[idx] = alpha * x[idx] + y[idx];
8  }
9
10 int main(int argc, char **argv)
11 { Array<float, 1> x(1000), y(1000);
12   float alpha;
13
14   for(int i = 0; i < 1000; i++) {
15     x(i) = ...;
16     y(i) = ...;
17   }
18
19   eval(saxpy)(y, x, alpha);
20 }
```

Figure 1: SAXPY code in HPL

This code follows these steps:

1. We begin including the header file for HPL in line 1 and declaring the usage of its namespace in line 3.

2. We then write the `saxpy` kernel in lines 5-8. The kernel takes as arguments two 1-dimensional Arrays (i.e., vectors) of elements of type `float` called `y` and `x` and a single-precision scalar called `alpha`. A scalar variable can be defined in HPL using an Array of 0 dimensions (e.g. `Array<float,0>`), or convenient alias such as `Float`.

3. The computation of the kernel is specified in line 7. HPL kernels use predefined variables to identify each parallel thread that runs a kernel, the number of threads in each dimension and other important properties. For example, `idx` is a predefined variable that identifies the thread that is running the current instance of the kernel in the first dimension of the problem domain. This way, line 7 specifies that the `idx`-th thread will compute `y[idx]` following the SAXPY algorithm.

4. The Arrays that the host program wants to use in the kernel are defined in line 11. Scalars do not need to be defined using the HPL datatypes, so the `alpha` variable in the host side can be a regular `float`.

5. The host code initialized the data of its Arrays in lines 15 and 16. Notice that host code must access Arrays using parenthesis (lines 15-16) while kernel must access them using brackets (line 7).

6. Finally, we run our `saxpy` kernel on our arrays using the syntax shown in line 19.

The rest of this document is organized as follows. First, Section 2 explains how to install HPL. Then, Section 3 introduces the hardware view and the programming model provided by HPL. The interface of the library is described in Section 4. Appendices A and B report compilation flags and environment variables that control the compilation and the execution of HPL programs, respectively.

2 Installation

HPL has the following requirements:

1. OpenCL 1.1 or above.
2. A a C++ compiler that supports C++11

There are two ways to install HPL:

- Based on the cmake tool[1]. **This is the mechanism recommended** for users because it is more portable and it gives place to a streamlined installation in a directory chosen by the user. The binaries, header files and libraries are placed in the `bin`, `include/hpl` and `lib` subdirectories, respectively.
- Based on traditional Unix makefiles. This mechanism builds HPL inside the directory were its tarball is unpacked, leaving the header files and executables in the `src` directory and the library in the `lib` directory.

Both mechanisms are now explained in turn.

2.1 Installation based on cmake

1. First, make sure that you fulfill the HPL build requirements.
2. Unpack the hpl tarball (`hpl_xxx.tar.gz`) and enter the just created directory:

   ```
   tar -xzf hpl_xxx.tar.gz
   cd hpl_xxx
   ```

3. Create the temporary directory where the project will be built and enter it:

   ```
   mkdir build
   cd build
   ```

4. Generate the files for building HPL in the format that you prefer (Visual Studio projects, nmake makefiles, UNIX makefiles, Mac Xcode projects, ...) using cmake.

   In this process you can use a graphical user interface for cmake such as `cmake-gui` in Unix/Mac OS X or `CMake-gui` in Windows, or a command-line interface such as `ccmake`. We will explain the process assuming this last possibility, as graphical user interfaces are not always available.

   Follow these steps:

   (a) Run `ccmake ..`
   This will generate the files for building HPL with the tool that cmake choses by default for your platform. Flag `-G` can be used to specify the kind of tool that you want to use. For example if you want to use Unix makefiles but they are not the default in you system, run `ccmake -G 'Unix Makefiles' ..`
   Run `ccmake --help` for additional options and details.

[1] Freely available for download from http://www.cmake.org

(b) Press letter 'c' to configure your build.

(c) Provide the values you wish for the variables that appear in the screen. The most relevant ones are:

- `CMAKE_BUILD_TYPE` : Specified the build type. Possible values are empty, Debug, Release, RelWithDebInfo and MinSizeRel.
- `CMAKE_INSTALL_PREFIX` : Directory where HPL will be installed
- `CLBLAS_DIR` : If you installed clBLAS[2] and you want to be able to use it on top of HPL, provide here the home directory of its installation. It is assumed that its headers will be found within directory `include` and its libraries in directory `lib`.

(d) When you are done, press 'c' to re-configure cmake with the new values.

(e) Press 'g' to generate the files that will be used to build HPL and exit cmake.

5. The rest of this explanation assumes that UNIX makefiles were generated in step 4.

   Run `make`

   This builds the HPL library and its tests.

   The degree of optimization, debugging information and assertions enabled depends on the value you chose for variable `CMAKE_BUILD_TYPE`.

   Note: You can use the flag `-j` to speedup the building process. For example, `make -j4` will use 4 parallel processes.

6. (Optionally) run `make check`

   This will run the HPL tests.

   Notice that some tests may fail even if HPL works correctly (e.g. if they use double precision but it is not supported by the default device chosen).

7. (Optionally) run `make checkclBLAS`

   This will run the clBLAS integration tests if you provided the clBLAS related variables in the set up. Notice that some tests may fail even if HPL works correctly (e.g. if they use double precision but it is not supported by the default device chosen).

   Note: clBLAS may require loading its dynamic libraries at runtime. Please set up your environment (ususally by means of the environment variables `LD_LIBRARY_PATH` in Unix or `DYLD_LIBRARY_PATH` in Mac OS X) to allow clBLAS to find its libraries.

8. Run `make install`

   This installs HPL under the directory you specified for the `CMAKE_INSTALL_DIR` variable. If you left it empty, the default base directories will be `/usr/local` in Unix and `c:/Program Files` in Windows.

   The installation places the binaries, header files and libraries in the `bin`, `include/hpl` and `lib` subdirectories of the chosen directory, respectively.

9. You can remove the `hpl_xxx` directory generated by the unpacking of the hpl tarball.

[2]Freely available for download from http://github.com/clMathLibraries/clBLAS

2.2 Installation based on Makefiles

1. First, make sure that you fulfill the HPL build requirements.
2. Unpack the hpl tarball (`hpl_xxx.tar.gz`) and enter the just created directory:

   ```
   tar -xzf hpl_xxx.tar.gz
   cd hpl_xxx
   ```

3. run `./configure`

   This will generate a `common.mk` file with a default configuration

4. edit `common.mk` as needed to select your desired compiler, compilation flags, location of OpenCL libraries, etc. modifying the corresponding variables.
5. Run `make`

   This builds the HPL library in directory `lib` and its tests in directory `tests`.

   The library is built by default without optimization flags and with debugging information and assertions activated.

   If you define the environment variable `PRODUCTION` before you run `make`, HPL and its tests will be compiled with optimization flags, no debugging information and no assertions. For example, in the bash shell you can write `PRODUCTION=1 make`

   Note: You can use the flag `-j` to speedup the building process. For example, `make -j4` will use 4 parallel processes.
6. (Optionally) run `make check`

   This will run the HPL tests.

   Notice that some tests may fail even if HPL works correctly (e.g. if they use double precision but it is not supported by the default device chosen).
7. (Optionally) run `make checkclBLAS`

   This will run the clBLAS integration tests if you set up clBLAS in `common.mk`. Notice that some tests may fail even if HPL works correctly (e.g. if they use double precision but it is not supported by the default device chosen).

   Note: clBLAS may require loading its dynamic libraries at runtime. Please set up your environment (ususally by means of the environment variables `LD_LIBRARY_PATH` in Unix or `DYLD_LIBRARY_PATH` in Mac OS X) to allow clBLAS to find its libraries.
8. (Optionally) run `make clean`

   This removes the temporary object files generated during the build of HPL as well as the test binaries, leaving only `lib/libHPL.a`

   `make veryclean` also removes `lib/libHPL.a` and other internal HPL files.

2.3 Basic test: compiling with HPL

Once you have installed HPL, you can try compiling yourself any of the tests provided with the library. If, for example, the test `MatrixAdd.cpp` is chosen, and you let cmake install HPL in the default `/usr/local` location, this is achieved with the command

```
c++ -std=c++11 -I/usr/local/include/hpl -o MatrixAdd MatrixAdd.cpp
               -L/usr/local/lib -lHPL -lOpenCL
```

assuming that OpenCL is available in the libOpenCL library. In Mac OS X the correct flag to link with OpenCL would be `-framework OpenCL`.

2.4 Environment setup for runtime optimizations

HPL can apply some runtime optimizations depending on the characteristics of the system where the application is being run. These optimizations require an analysis of the system that is performed offline by means of a tool called `PerfAdapt` and is stored in a file. HPL tries to read this file at runtime if we provide its location by means of the environment variable `HPL_CONFIG_FILE`. Since `PerfAdapt` writes the results of it analysis as text to the console and it must be run in each system in which we want to HPL, the correct setup follows these steps:

1. Run `PerfAdapt` in each system where you expect to run HPL, accumulating the output of all the invocations in a single file. For example, in an UNIX system this can be achieved writing

   ```
   PerfAdapt >> hpl_config_file.txt
   ```

2. Set up the environment variable `HPL_CONFIG_FILE` to point to this file, for example with

   ```
   export HPL_CONFIG_FILE=hpl_config_file.txt
   ```

3 Hardware and Programming Model

The abstract view of the underlying hardware considered by HPL is depicted in Figure 2. There is a host with a memory and a CPU in which the main program runs. Attached to it, there are a number of computing devices, each one of them with its own memory and a number of processors that can only access the memory within their device. While different devices can run different codes, all the processors in the same device must execute the same code in an SPMD fashion. In some devices the processors are subdivided in groups that share a scratchpad memory of a limited size and can synchronize by means of barriers, this being the only mechanism available to synchronize processors within a device.

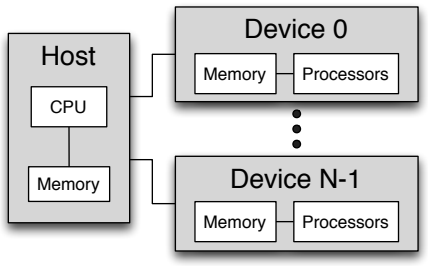

Figure 2: Underlying hardware model for HPL

Given this view, a HPL application consists of (regular) code executed in the host and portions of code that are run in an SPMD fashion in one or several devices under the request of the host. These parallel portions of the application are expressed as functions that are evaluated in parallel by the processors in the selected device. These functions are called *kernels*, since they are analogous to the kernels found in CUDA or OpenCL, for example.

Each thread that runs a copy of a kernel needs a unique identifier so that it can identify the work it is responsible for. To allow for this, kernels are executed on a domain of integers of up to three dimensions, called a *global domain*. Each point in this domain is assigned a unique identifier that is associated to an instance of the requested kernel, and therefore the size of this domain is the total number of parallel threads running the requested kernel.

The user can optionally specify a *local domain*, which must have the same number of dimensions as the global domain and whose size in every dimension must be a divisor of the size of the corresponding dimension of the global domain. The threads whose identifiers belong to the same local domain can share scratchpad memory and synchronize by means of local barriers. These threads form what we call a *group* of threads, each group also having an n-dimensional identifier.

Figure 3 represents the unique global identifiers of the 32 threads to run for a global domain of 4×8 threads. The identifiers of threads that belong to the same local domain (or group) of 2×4 threads are surrounded by a thicker line. The unique identifier of each thread group is also indicated.

Lastly, regarding memory, kernels can only access the processor's registers and the memory available inside the device where they are run. HPL distinguishes four kinds of device memory:

Global memory : It is the standard memory of the device, which is shared by all the

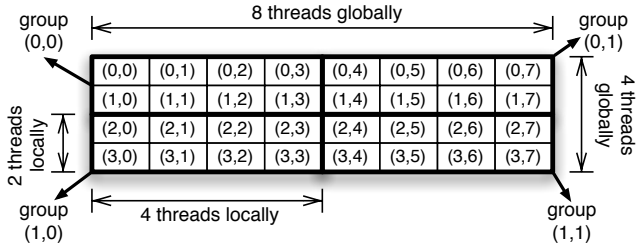

Figure 3: Global and local domains for the threads that execute in parallel a HPL kernel

processors in the device both for reading and writing.

Local memory : This is the scratchpad memory that can be only accessed by the threads that belong to the same local domain.

Constant memory : A memory for data that can be written by the host but which the kernels can only read.

Private memory : A memory that is separate for each thread running a kernel, so that only that thread can access it.

4 Programming Interface

The HPL interface is made available by the inclusion of the header file `HPL.h` and it is encapsulated inside the `HPL` namespace in order to avoid collisions with other program objects. The interface has three components:

- **Array** : a datatype to represent both scalars and n-dimensional arrays that can be used both in the host code and in the kernels.
- **Device-only interface** : it is only used to write the code of the kernels.
- **Host-only interface** : it is only used in the host code. Its main purpose is to identify the devices available in the system and request the execution of kernels on them.

We now describe these elements in turn.

4.1 Arrays

Like any function, HPL kernels have parameters and private variables. Both kinds of variables must have type `Array<`*type, ndim [, memoryFlag]*`>`. This is a C++ template class where

- **type** is the standard C++ type of the contents. The left column in Table 1 displays the types currently supported. `Array`s can also be made up of `struct`s (see Section 4.1.3).
- **ndim** is the number of dimensions of the array or 0 for scalars. The maximum number of dimensions currently supported is three.
- **memoryFlag** is optional. It either specifies one of the kinds of memory supported (`Global`, `Local` and `Constant`, in the order used in Section 3) or is `Private`, which indicates that the variable is private to each thread in its kernel instantiation.

  The default *memoryFlag* is `Global`, except for variables defined inside kernels. These latter variables can only be either `Local` or `Private`, which is their default. `Array`s with the `Local` flag are shared by all the threads in a group even if they are defined inside a kernel.

When the host code invokes a kernel, it provides the arguments for its execution, which must also be `Array`s. This way, the input and output kernel `Array`s must be declared in the host space.

standard C types supported in `Array`s	HPL type for a scalar
int	Int
unsigned int	UInt
size_t	Size_t
float	Float
double	Double

Table 1: Basic types supported in `Array`s

```
float v[100][100];

Array<double, 3> mx(20, 30, 40); //20 x 30 x 40 array of doubles
Array<float, 2> vx(100, 100, v); //v provides the storage
Array<int, 1> fastv(1000, HPL_FAST_MEM); //Allocate in pinned memory
Int i; //Same as Array<int, 0> i;
```

Figure 4: Construction of `Array`s

As Fig. 4 shows, the constructor for an n-dimensional `Array` takes as inputs the sizes of its dimensions. By default the library is responsible for allocating and deallocating the memory required to store the `Array` contents no matter it is defined in host code or in a kernel. `Array`s built in the host space accept in their constructor an optional final argument whose value can be

- a pointer to a previously allocated memory region that provides the space for the array in the host memory. If such memory is provided, HPL makes no host allocations or deallocations for this array. Instead, the user is responsible for its deallocation.
- `HPL_FAST_MEM`, which requests that the array is allocated in the host pinned memory. This can make the transfers between the host memory and the device faster
- `HPL_NOHOST_MEM`, which indicates that this array needs no host-side allocation, i.e., it will be only accessed in the devices.

While scalars can be defined using the `Array` template class with *ndim=0*, the convenience types shown in the right column of Table 1 simplify the definition of scalars of the obvious corresponding C++ type.

HPL also provides vector types both in the kernels and in the host code. HPL currently supports vectors of 2, 4, 8 and 16 elements either of type `int` or `float`. The host native type of a vector of n elements of type *type* is *typen* (e.g. `float2` or `int16`), while the corresponding `Array` type that can be used in kernels begins with uppercase (e.g. `Float2` or `Int16`).

Both `Array`s and HPL vector types are indexed with the usual square brackets in kernels. However, their indexing in host code is made with parenthesis. This visually emphasizes the fact that their accesses in the host code experience overheads that do not exist in the kernels. The most important overhead is that the library tracks each access in the host code in order to learn whether it is only read, written or both. This information is used to minimize the number of transfers between the host and the devices using a lazy copying policy. Section 4.1.2 describes techniques to reduce this overhead in the host code.

4.1.1 Methods

`Array`s supports the following methods in the host code, but not in kernels:

- `T *data(AccessType at = HPL_RDWR)` returns a raw pointer to the contents of the `Array` in the host memory. The optional input flag indicates whether the following host code will only read (`HPL_RD`), write (`HPL_WR`) or perform both operations (`HPL_RDWR`) on the array through the pointer. If no flag is provided, `HPL_RDWR` is assumed.

```
Array<float, 1> a(N), b(N);
...
for(int i = 0; i < N; i++)
  a(i) = b(i) + 1;
```

(a) automated management

```
Array<float, 1> a(N), b(N);
float *pa, *pb;
...
pa = a.data(HPL_WR);
pb = b.data(HPL_RD);
for(int i = 0; i < N; i++)
  pa[i] = pb[i] + 1;
```

(b) manual management

```
Array<float, 1> a(N), b(N);
/* compile with
   -DHPL_NO_AUTOSYNC */
...
a.data(HPL_WR);
b.data(HPL_RD);
for(int i = 0; i < N; i++)
  a(i) = b(i) + 1;
```

(c) intermediate management

Figure 5: Usage of `Arrays` in host code

- `void *getData(AccessType at = HPL_RDWR)` same as `data` but the return type is `void *` instead of `T *`.
- `int getDimension(int d)` return size of dimension `d`.
- `T reduce(OP op)` returns the reduction of all the values in the array using functor `op`, which can be any binary functor. Not implemented in scalar `Arrays`.

> Example: `r = mx.reduce(std::plus<float>());`
>
> adds all the elements in `Array mx` in destination `r`.

- `T value()` returns the value stored in a constant scalar `Array<T, 0>`.
- `T& value()` returns a reference to the value stored in a non-constant scalar `Array<T, 0>`.

The following methods are supported both in host code and kernels:

- `size_t getDataItemSize()` return the size of one data item of the array.
- `int getNumberOfDimensions()` return number of dimensions.

4.1.2 Host side `Array` usage optimization

The overhead that the automated management generates in the accesses to `Arrays` in the host can be reduced in two ways. Both of them involve using the `Array` method `data` described in Section 4.1.1.

The first strategy consists in using method `data` to inform the system on the kind of access, and then directly performing the accesses through the pointer obtained. Fig. 5 compares a code (a) with totally automated management with a code (b) in which the user applies this strategy.

The second possibility consists in compiling our application with the flag `-DHPL_NO_AUTOSYNC`, which disables the detection of the kind of access performed when an `Array` is accessed, thereby strongly reducing the library overhead. Since the library cannot automatically track the accesses, the user must use method `data` to provide this information. Fig. 5(c) illustrates our example code written using this strategy.

```
HPL_DEFINE_STRUCT( mystruct_t,
                   { int i;
                     float f;
                   } );

Array<mystruct_t, 2> matrix(100, 100);
```

Figure 6: Declaring a `struct` type to HPL in order to use it in `Array`s

4.1.3 Arrays of structs

HPL also supports the usage of `Array`s of `struct`s. In order to allow the usage of a `struct` in a HPL `Array`, its definition must be made known to HPL. For this purpose HPL provides two macros that should be invoked outside any program function:

- `HPL_DEFINE_STRUCT(`*struct_name*`,` *struct_body*`)` defines the `struct` and registers it in the HPL internal system. This macro must be used in a unit of compilation that is compiled only once, such as a `.cpp` file.
- `HPL_DECLARE_STRUCT(`*struct_name*`,` *struct_body*`)` provides a declaration of the `struct` suitable to be included in header files.

A program using a given `struct` type in HPL `Array`s must include one and only one `HPL_DEFINE_STRUCT` that should only be compiled once. Additionally, the program can include any number of `HPL_DECLARE_STRUCT` macros for the associate type. Fig. 6, where `mystruct_t` is the name we want to give to the `struct`, illustrates the syntax of these macros.

The fields of a struct are accessed using the syntax

- *structVariable*`.at(`*field*`)` inside kernels (e.g. `matrix[0][0].at(f)`)
- `field(`*structVariable*`,` *field*`)` in host code (e.g. `field(matrix(0,0), f)`). Of course it is also possible to get a pointer `p` to the data of the array using method `data` and just apply the usual `p->f` notation.

4.1.4 AliasArrays

Sometimes it is useful to declare pointers to locations within existing `Array`s in kernels because this simplifies the indexing functions of these `Array`s. HPL supports this possitiblity by means of the kernel-only `AliasArray` class template, which is simply a pointer to a fixed position inside an `Array`. The only template argument to `AliasArray` is the type of the elements of the underlying array. `AliasArray`s do not support pointer arithmetic of dereferencing (i.e., via the `*` operator), but rather they are used like a normal 1-dimensional `Array`. Similarly, they get the position they point to in their constructor, and it cannot be changed during the lifetime of the `AliasArray`.

Figure 7 illustrates the usage of `AliasArray`s with an example in with each thread `idx` (See Section 4.2.1 and Table 2 for the meaning of this variable) builds an `AliasArray ptr` that points to the beginning of the row `idx` of the input `AliasArray matrix` and then uses it to multiply the elements in the row by 2.

```
void kernel(Array<float, 2> matrix, Int M)
{ Int i;

  AliasArray<float> ptr(matrix[idx][0]);
  for_( i = 0, i < M, i++) {
   ptr[i] *= 2.f;
  }
}
```

Figure 7: Example of `AliasArray` definition and usage.

Meaning	First dimension	Second dimension	Third dimension
Global id	`idx`	`idy`	`idz`
Local id	`lidx`	`lidy`	`lidz`
Group id	`gidx`	`gidy`	`gidz`
Global domain size	`szx`	`szy`	`szz`
Local domain size	`lszx`	`lszy`	`lszz`
Number of groups	`ngroupsx`	`ngroupsy`	`ngroupsz`

Table 2: Predefined HPL variables.

4.2 Kernel interface

The HPL interface used to write the code of the kernels includes

- Predefined functions to uniquely identify each thread and group of threads as well as the dimensions of the global and the local domain.
- Macros to express conditional and iterative constructs.
- Generic helper functions.
- Arithmetic functions.
- Atomic functions.

The following sections describe these components in turn.

4.2.1 HPL automatic variables

Table 2 described the predefined variables provided by HPL in order to obtain the global id of the thread within the global space (first row), the local id within the thread group (second row) and the identification of the thread's group (third row). It also provides the size of the global domain, the local domain and the number of groups of threads per dimension (fourth, fifth an sixth rows, respectively).

4.2.2 Control flow macros

Table 3 shows the C++ constructs supported in the HPL kenels with the corresponding HPL translation. Notice that the arguments to `for_` are separated by commas, not semicolons. A convenience `else_if_`(expr) macro is also provided.

C++ construction	HPL construction
`for`(...; ...; ...) { ... }	`for_`(..., ..., ...) { ... }
`if`(exp) { ... }	`if_`(exp) { ... }
`if`(exp) { ... } `else` { ... }	`if_`(exp) { ... } `else_` { ... }
`return;`	`return_`();
`return(exp);`	`return_`(exp);
`while`(exp) { ... }	`while_`(exp) { ... }

Table 3: Correspondence between C++ and HPL constructs

Older versions of HPL required marks at the end of the blocks of code (`endif_`, `endfor_` and `endwhile_`) which were interchangeable and equivalent, and any of them could be replaced with the generic keyword `end_`. Now these marks are deprecated since they are no longer needed.

4.2.3 Generic functions

- `barrier(Sync_t flag)` performs a barrier synchronization between the threads in a group. The `flag` indicates wheter a consistent view of the local memory, the global memory or both is needed after the barrier, the corresponding values being `LOCAL`, `GLOBAL` and `LOCAL|GLOBAL`, respectively.
- `call(f)(...)` invoke function f, which must be written in HPL, using the arguments specified.
- `cast<T>(expr)` casts the result of the evaluation of expression `expr` to the type `T`.

> Example: `call(ftr)(a,b,c);`
>
> invokes HPL function `ftr` with arguments `a`, `b` and `c`.

- `reduce(dest, input, op)` is a cooperative function run by all the threads in a group. It reduces into destination `dest` the `input` value provided by each thread of the group using the operation `op`, which is provided as a regular C string (`"max"`, `"+"`, ...). Only the thread 0 of the group writes the result to the destination.

> Example: `reduce(v[gidx], f, "max");`
>
> stores into `v[gidx]` the maximum `f` among all the threads in the group.

  Optional modifiers that can allow to optimize the implementation:

  - `ndims(n)` : the kernel will be run using `n`-dimensional thread domains.
  - `groupSize(n)` : the kernel will be using thread groups of `n` threads.
  - `minGroupSize(n)` : the kernel will be using thread groups of at least `n` threads.
  - `localMem(n)` : use at most `n` items of the same type as `dest` and `input` in the local memory to perform the reduction.

- `toAll()` : all the threads, rather than only thread 0, must write the reduction result to `dest`.
- `inTree()` : perform the reduction of the local array as a parallel binary tree. Otherwise it is made sequentially.
- `unroll(b)` : boolean that indicates whether loops should be unrolled or not. As of now it only applies to the binary reduction and defaults to true.
- `syncReq(n)` : request to generate synchronizations only when `n` or more threads are involved in a part of the algorithm. This is useful when we know the code is going to be run in devices where a given number of threads operate in lockstep, so that their synchronization is ensured.
- `nElems(n)` : the reduction will take `n` consecutive elements from the memory position of the input of each thread and reduce the corresponding elements, giving place to a final vector of `n` elements which will be stored beginning at the address of the destination.
- `nTeams(n)` : the threads in each group will be divided in `n` teams of consecutive threads according to their id. Each team will cooperate in a different reduction.

Example: `reduce(r, v[idx], "*").ndims(1).minGroupSize(32).toAll();`

multiply the `v[idx]` values provided by all the threads in a group. All of them store the outcome in `r`. The code is generated knowing that this is a one-dimensional problem in which the minimum number of threads in a group will be 32.

Example: `reduce(r, v[idx], "+").nTeams(4).groupSize(128).inTree().syncReq(33).toAll();`

each group of threads has 128 threads, and they are organized in four teams (threads 0 to 31, 32 to 63, 64 to 95 and 96 to 127). The threads in each team reduce the value they provide in `v[idx]` by adding it, and all of them get the result in variable `r`. The reduction is made in binary tree in a device that only requires explicit synchronizations between threads in steps in which 33 or more threads have to cooperate.

Example: `reduce(r[N*gidx], q[0], "+").groupSize(32).nElems(10);`

in a group of 32 threads, all of them will add their `q[0]` to generate `r[N*gidx]`, their `q[1]` to generate `r[N*gidx+1]`, and so on up to `q[9]`, whose addition will be stored in `r[N*gidx+9]`.

- `where(a, b, c)` corresponds to the C construction `(a) ? (b) : (c)`

4.2.4 Arithmetic functions

Besides the usual +, -, *, /, !, ~, %, &, |, ^, &&, ||, <, >, ==, <=, >=, !=, << and >> operators, the following functions are currently available:

- `acos`
- `acosh`
- `acospi`
- `asin`
- `asinh`
- `asinpi`
- `atan`
- `atanh`
- `atanpi`
- `cbrt`
- `ceil`
- `cos`
- `cosh`
- `cospi`
- `erfc`
- `erf`
- `exp`
- `exp2`
- `exp10`
- `expm1`
- `fabs`
- `floor`
- `ilogb`
- `lgamma`
- `log`
- `log2`
- `log10`
- `log1p`
- `logb`
- `nan`
- `native_rsqrt`
- `native_sqrt`
- `rint`
- `round`
- `rsqrt`
- `sin`
- `sinh`
- `sinpi`
- `sqrt`
- `tan`
- `tanh`
- `tanpi`
- `tgamma`
- `trunc`
- `dot`
- `max`
- `native_divide`
- `pow`

4.2.5 Atomic functions

In all these functions `var` is the name of a memory position of underlying type `int` or `unsigned int`. This way, it can be either an `Int` or `UInt` scalar variable or an element of an `Array` of `int` or `unsigned int`. As for `val`, it is an expression of the same underlying type as `var`.

- `atomic_add(var, val)` atomically adds value `val` to variable `var`. The old value of `var` is returned.
- `atomic_and(var, val)` atomically performs the bitwise and of the value `val` and variable `var` and stores the result in `var`. The old value of `var` is returned.
- `atomic_cmpxchg(var, cmp, val)` atomically compares `var` with `cmp`, changing the value of `var` to `val` if they are equal, and leaving it unchanged otherwise. The old value of `var` is returned.
- `atomic_dec(var)` atomically decrements integer variable `var`. The old value of `var` is returned.
- `atomic_inc(var)` atomically increments integer variable `var`. The old value of `var` is returned.
- `atomic_max(var, val)` atomically computes the maximum of variable `var` and the value `var` and stores the result in `var`. The old value of `var` is returned.
- `atomic_min(var, val)` atomically computes the minimum of variable `var` and the value `var` and stores the result in `var`. The old value of `var` is returned.

Type	Meaning	Values	Description
Device_t	Type of physical device	CPU	General purpose CPU
		GPU	Programmable GPU
Platform_t	Type of backend	AMD	AMD OpenCL
		APPLE	Apple OpenCL
		INTEL	Intel OpenCL
		NVIDIA	Nvidia OpenCL

Table 4: Enumerated types associated to kinds of supported devices and backends

- `atomic_or(var, val)` atomically performs the bitwise or of the value `val` and variable `var` and stores the result in `var`. The old value of `var` is returned.
- `atomic_sub(var, val)` atomically subtracts value `val` from variable `var`. The old value of `var` is returned.
- `atomic_xchg(var, val)` atomically swaps the old value of variable `var` with the new value `val`. The old value of `var` is returned.
- `atomic_xor(var, val)` atomically performs the bitwise xor of the value `val` and variable `var` and stores the result in `var`. The old value of `var` is returned.

4.3 Host interface

The HPL interface for the host application includes mechanisms to

- discover the devices available in the system and their attributes.
- request the execution of kernels on the devices.

The following sections describe these components in turn.

4.3.1 Device management

HPL can provide access to different kinds of devices and on top of different backend platforms. This way, HPL provides the enumeration types `Device_t` and `Platform_t` to refer to the kinds of hardware devices it can give access to, and the backends on top of which it can access them. Table 4 describes these two types and the values they can take. The user can learn the number of specific devices found in the system through the functions

- `unsigned int getDeviceNumber(Device_t type_id)` returns the number of devices of type `type_id`.
- `unsigned int getDeviceNumber(Platform_t platform_id, Device_t type_id)` returns the number of devices of type `type_id` that can be accessed on top of platform `platform_id`.

The class used to refer to a specific device is called `Device`. It has the following constructors and methods:

- `Device()` (default constructor) refers to the first GPU in the systems, or if not available, to the first CPU in the system. If there are several platforms that can support the chosen device, it chooses the one provided by the vendor if available, otherwise any can be chosen.
- `Device(Device_t type_id, int n = 0)` refers to the `n`-th device of type `type_id` available in the system. If no number is provided, it refers to the first one (`n`= 0). The platform is chosen using the same algorithm as in the default constructor.
- `Device(Platform_t platform_id, Device_t type_id, int n = 0)` refers to the `n`-th device of type `type_id` available on top of platform `platform_id` in the system. If no number is provided, it refers to the first one (`n`= 0).
- `sync()` waits for all pending taks on the device to finish.
- `getProperties(DeviceProperties& dp)` obtains the properties of the device in `dp`.

4.3.2 Kernel execution

In order to run a kernel written in a function `f` with a list of arguments `arg0`, `arg1`, `...`, the user must write in the host code `eval(f)(arg0, arg1, ...)`. The execution will take place in the device chosen by default by the library (see Section 4.3.1) using a global domain with the number of dimensions and sizes of the first argument of the evaluation (`arg0`) and a local domain chosen by the library. The device to use and the size of the global and the local domains, can be controlled using these methods to `eval`:

- `device(Device d)` run the kernel in the device `d`.
- `device(Device_t type_id, int n = 0)` run the kernel in the device built by `Device::Device(type_id, n)`.
- `device(Platform_t platform_id, Device_t type_id, int n = 0)` run the kernel in the device given by `Device::Device(platform_id, type_id, n)`
- `global(unsigned int x)` run the kernel using a 1-D global domain of size `x`.
- `global(unsigned int x, unsigned int y)` run the kernel using a 2-D global domain of size `x`×`y`.
- `global(unsigned int x, unsigned int y, unsigned int z)` run the kernel using a 3-D global domain of size `x`×`y`×`z`.
- `global(Domain d)` run the kernel using the global domain specified by `d`.
- `local(unsigned int x)` run the kernel using a 1-D local domain of size `x`.
- `local(unsigned int x, unsigned int y)` run the kernel using a 2-D local domain of size `x`×`y`.
- `local(unsigned int x, unsigned int y, unsigned int z)` run the kernel using a 3-D local domain of size `x`×`y`×`z`.
- `local(Domain d)` run the kernel using the local domain specified by `d`.

```
for(int i = 0; i < N; ++i)
  eval(f).device(d).global(gx,gy)
         .local(lx,ly)(a,b,c);
```

(a) Loop with `eval()`

```
FRunner r = eval(f).device(d).global(gx,gy)
                   .local(lx,ly)(a,b,c);
for(int i = 1; i < N; ++i)
  r.run();
```

(b) Loop with `run()`

Figure 8: Usage of `run()` method

Example: `eval(f).device(d).global(3200,3200).local(32,32)(a,b,c);`

runs the HPL kernel in function `f` in the device `d` using a global domain of 3200×3200 threads decomposed in groups of 32×32 threads, using the arguments `a`, `b` and `c`.

The `Domain` class mentioned above is a helper that allow to define 1-D, 2-D and 3-D domains with the simple syntax `Domain(a)`, `Domain(a,b)` and `Domain(a,b,c)`, respectively.

Example: `eval(mykernel).device(GPU, 0).global(Domain(1024,1024))(a,b);`

runs the HPL kernel in function `mykernel` in the first GPU found in the system using a global domain of 1024×1024 threads using the arguments `a` and `b` and a local domain chosen by the library.

Advanced kernel execution: In order to accelerate the execution of a kernel in a loop, the user can avoid the repetitive setting of buffers or the dimensions of the problem using the method `run()` onto the object returned by `eval()`, which has type `FRunner` (see Figure 8). The `run()` method is called in a loop body and onto an object whose configuration was set previously outside of the loop. The behavior of the `run()` is exactly the same that `eval()` from the programmer point of view but it does not perform as many checks as this one.

4.3.3 Using native kernels

HPL allows to run kernels defined as strings in OpenCL C. For this purpose, the user must provide the kernel name, a string with its definition and code, and finally a function that describes the type and the purpose (input, output or both) of each argument to the kernel by means of its list of parameters. This function is used as handle for the native in the `eval()` invocations, and its body is disregarded. The function parameters must be `Array`s of the appropriate number of dimensions, just as in any HPL kernel. By default an argument will be considered as both input and output to the kernel. In order to indicate the purpose for an input of type `T`, the formal parameter will be of type:

- `In<T>`: the argument is only an input.
- `Out<T>`: the argument is only an output.

```
1  // Native OpenCL C kernel string
2  const char * const kernel_code = TOSTRING(
3    __kernel void mxmul_simple(__global float *a, __global float *b,
4                               __global float *c, const int n)
5    { int i, j, m, p, k;
6      float f = 0.;
7      m = get_global_size(0);
8      p = get_global_size(1);
9      i = get_global_id(0);
10     j = get_global_id(1);
11     if(i < m && j < p) {
12       for(k = 0; k < n; k++)
13         f += a[i*n+k] * b[k*p+j];
14       c[i*p+j] = f;
15     }
16   } );
17
18 // Define function that specifies the type of each kernel argument
19 void matmulGPU(In< Array<float, 2> > a,
20                In< Array<float, 2> > b,
21                Out< Array<float, 2> > c,
22                Int n)
23 { }
24
25 // Associate handle with kernel name and its string
26 nativeHandle(matmulGPU, "mxmul_simple", kernel_code);
27
28 int m, n, p;
29 Array<float, 2> a(m, n), b(n, p), c(m, p);
30
31 // Ready to use by means of its handle funcion!
32 eval(matmulGPU).global(m, p)(a, b, c, n);
```

Figure 9: Example of usage of a native kernel

- `InOut<T>`: the argument is an input and an output.

Before the first time the function handle is used in an HPL evaluation it must be associated to the native kernel by means of function `void nativeHandle(f, name, native_code)` where `f` is the C++ function that represents the kernel and provides the information on its parameters, `name` is the kernel name and `native_code` contains the kernel code. Figure 9 illustrates how to use a native OpenCL code for matrix multiplication.

A convenient macro `TOSTRING` that turns into a C string (`const char *`) its argument is provided to simplify the creation of the string that contains a native kernel code. Also, if the last argument of `nativeHandle` does not look like code, the function interprets it as a file name from which the actual code must be read.

4.3.4 Profiling

In order to profile HPL runs, both the library and the user code to profile must be compiled with the `HPL_PROFILE` macro defined (i.e., using `-DHPL_PROFILE`). If you installed HPL using the cmake-based approach explained in Section 2.1, you already have a version of

Type	Field	Meaning
double	secsKernelCreation	time used in building the kernel code
double	secsKernelCompilation	time used in compiling the kernel
double	secsDataHostToDevice	time spent in the transfers from host to the device
double	secsDataDeviceToHost	time spent in the transfers from device to host
double	secsKernelExecution	time spent in the execution of the kernel

Table 5: Fields of struct `ProfilingData`

```
-----------------------------------------
|       HPL PROFILER OUTPUT             |
-----------------------------------------
|       STAGE          |     TIME (s)     |
-----------------------------------------
| Kernels creation     |     0.000340     |
| Kernels compilation  |     0.006370     |
| CPU->GPU             |     0.020566     |
| GPU->CPU             |     0.011292     |
| Kernels execution    |     0.398723     |
-----------------------------------------
```

Figure 10: Example HPL profiling output

the library compiled with profiling enabled called `HPLprofile.a` in the library directory. If you followed the makefiles-based approach explained in Section 2.2, you must make sure you build a version of the library with profiling mode enabled. This is achieved building the library with the environment variable `HPL_PROFILE` defined (see Appendix A). This environment variable also applies this macro in the compilation chain of the tests provided with the library.

Important: It is inconsistent to compile a program without the `HPL_PROFILE` flag with a HPL library compiled in profile mode, or to try to compile a program in profile mode with a HPL library not built in profiling mode. The results are undefined.

In profiling mode, HPL gathers five types of statistics that are stored in a struct of type `ProfilingData`, whose fields are illustrated in Table 5 with their meaning. HPL provides two host functions to access this profiling information:

- `ProfilingData getProfile()` returns a `ProfilingData` struct with the profiling of the most recent kernel execution.
- `ProfilingData getTotalProfile()` returns a `ProfilingData` struct that summarizes (adds) the profiling statistics of all the kernel runs performed.

In addition, applications compiled in profiling mode always generate when they finish their execution a text file called `HPL_PROFILER_OUTPUT.txt`. This file contains a human readable representation of the global statistics gathered for all the kernel runs with the form shown in Fig. 10.

A Library compile time flags and environment variables

The compilation definitions/macros that change the behavior of the library are:

- `NDEBUG` : removes `assert` macros
- `DEBUG` : enables additional messages, checks and debugging macros defined in `HPL_utils.h`
- `HPL_NO_AUTOSYNC` : disables autodetection of read or write accesses through the operator() accesses to the HPL `Array`s. The user can still express the usage of an `Array` with a flag to method `getData`.
- `HPL_PROFILE` : Compiles the library (and tests provided with it) in profiling mode. See Section 4.3.4 for more details.

The environment variables that can be set to configure the compilation of the library when using the makefiles-based mechanism described in Section 2.2 are:

- `DEBUG` : defines the `DEBUG` flag in the compilation of the library.
- `PRODUCTION` : defines `NDEBUG` and optimization flags such as `O3`. It also removes debugging information.
- `HPL_PROFILE` : defines the `HPL_PROFILE` flag in the compilation of the library and tests.

Example: (assuming `bash` shell): `DEBUG=1 make clean all`

builds the HPL library in debug mode.

B Runtime variables

These environment variables change the behavior of HPL at runtime :

- `HPL_CONFIG_FILE` : location of the configuration file that allow to apply the runtime optimizations detected by the `PerfAdapt` utility and stored in this file.
- `HPL_KERNEL_FILE_ENABLE` : kernels generated are dumped to a file called `HPL_kernels.cl`.
- `HPL_KERNEL_FINISH_ENABLE` : force to wait kernel completion.
- `HPL_OPT_COPY_OUT_DISABLE` : disable to automatic data transfer optimization such as copy out argument even the kernel did not change the argument.
- `HPL_PRINT_COPY` : prints data transfer information (`copy_in`/`copy_out`) between device memory to host memory when it is occurring.
- `HPL_UNIFIEDMEMORY_ENABLE` : This flag allows to share the same data pointer between the host and the OpenCL device. This flag should be activated always when the OpenCL device is a CPU to avoid the memory copies between the host process and the OpenCL device. The data pointer of the HPL arrays are already aligned to 4KB to take the maximum advantage of the pinned memory created by mean of the CL_USE_HOST_PTR flag.

In addition, during the execution the following global variables control certain aspects of the execution:

- `bool AutoManageExceptions`: When true, which is the default, the library catches `eval` exceptions and exits the program. When false, all the exceptions are thrown to the user code.

C clBLAS integration

HPL can be optionally compiled with support for clBLAS by providing the directory where clBLAS is installed before compiling HPL. The integration provides two interface functions for each clBLAS operation: one simplified that only contains the most basic arguments, and another one complete that includes all of them. The names of the functions follow the pattern `clblas` + $<$ *data − type letter* $>$ +*function_name*, where the data-type letters are

`S` : Single precision.

`D` : Double precision.

`C` : Complex in single precision.

`D` : Double in single precision.

For example, the interface provided to the single precision `gemm` operation are the functions:

```
clblasStatus clblasSgemm(const HPL::Array<float, 2>& x,
                         const HPL::Array<float, 2>& y,
                         HPL::Array<float, 2>& z,
                         const HPL::Device& d = HPL::Device());

clblasStatus clblasSgemm(clblasOrder order,
                         clblasTranspose transA,
                         clblasTranspose transB,
                         size_t M,
                         size_t N,
                         size_t K,
                         cl_float alpha,
                         const HPL::Array<float, 2>& A,
                         size_t offA,
                         size_t lda,
                         const HPL::Array<float, 2>& B,
                         size_t offB,
                         size_t ldb,
                         cl_float beta,
                         HPL::Array<float, 2>& C,
                         size_t offC,
                         size_t ldc,
                         const HPL::Device& d = HPL::Device(),
                         cl_uint numEventsInWaitList = 0,
                         const cl_event * eventWaitList = 0,
                         cl_event * events = 0);
```

In order to use the interface the programmer must ensure that:

1. The header file `HPL_clBLAS.h` is included.
2. Function `HPL_clblasSetup()` is invoked before any clBLAS call is performed.
3. Function `HPL_clblasTeardown()` should be called after clBLAS is used.

The compilation of the application requires the clBLAS header files and library in addition to those of HPL, thus appropriate flags must be provided to specify all of them.