HMPP™: A Hybrid Multi-core Parallel Programming Environment

Romain Dolbeau, Stéphane Bihan, and François Bodin, *CAPS entreprise*[1]

***Abstract*— Hybrid parallel multi-core architectures based on Graphical Processing Units (GPU) can provide tremendous computing power. Current NVIDIA and ATI hardware display a peak performance of hundreds of gigaflops. However, exploiting GPUs from existing applications is a difficult task that requires non-portable rewriting of the code. In this paper, we present HMPP, an Heterogenous Multi-core Parallel Programming environment that allows the integration of heterogeneous hardware accelerators in a seamless intrusive manner while preserving the legacy code.**

Keywords: GPGPU, Heterogeneous Programming Environment, Heterogeneous Core Integration.

I. Introduction

Due to their high potential computing power, the use of graphical processing units (GPUs) looks very attractive to speedup applications. Furthermore, new programming environments such as CUDA [1], RapidMind [2], PeakStream [3] or CTM [4] have made the use of GPUs, for general purpose programming, easier and more efficient. These devices achieve high performance with highly parallel microarchitecture and fast internal memories. This is illustrated in Figure 1. Data transfers are implemented using the PCI express bus or, in more coupled systems, via the HyperTransport channel [5]. The performance not only depends on the fast architecture but is also determined by the data communication overhead between the CPU and the GPU. Not all applications benefit from a GPU acceleration. This depends on the types of data used that have to be floating point and also on the amount of parallelism that can be extracted from the application to build a stream computation, i.e. a kernel applied on a stream of data. That's why vendors, such as NVIDIA and RapidMind, have extended the C language with new types in order to handle stream data. While it should allow to map computations more efficiently and transparently on the hardware accelerator architecture, it requires the developers to rewrite their algorithms in a new language and leads to different hardware specific versions of the same source to maintain.

[1] romain.dolbeau@caps-entreprise.com, stephane.bihan@caps-entreprise.com, françois.bodin@caps-entreprise.com

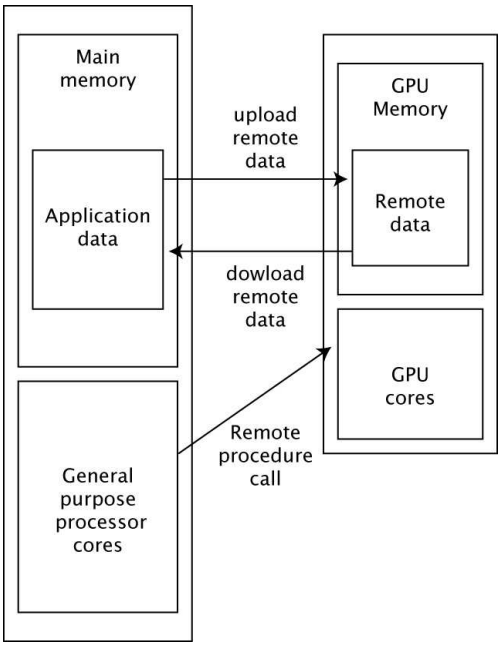

Figure 1: Use of a GPU as an Hardware Accelerator for General Purpose Applications.

1) Overview

This paper proposes a solution to not only simplify the use of hardware acceleration in conventional general purpose applications, but also to keep the application code portable. The goal is to integrate the use of hardware accelerators rather than porting the application to make use of them. The hardware-specific versions of the computations to be offloaded on an accelerator are dissociated from the original application source code. The CAPS HMPP toolkit [6] is a set of compiler directives, tools and software runtime that supports multi-core processor parallel programming in C and Fortran on Unix platforms. HMPP gives programmers a simple, flexible and portable interface for developing parallel applications whose critical computations are distributed, at runtime, over the available specialized and heterogeneous cores.

The chosen approach is similar to a widely available standard, OpenMP, but designed to handle hardware accelerators. As such, the application source code is kept portable and a sequential binary version can be built using a traditional compiler. Furthermore, if the hardware accelerator (HWA) is not available for any reason, the legacy code still can be executed and the application behavior is unchanged.

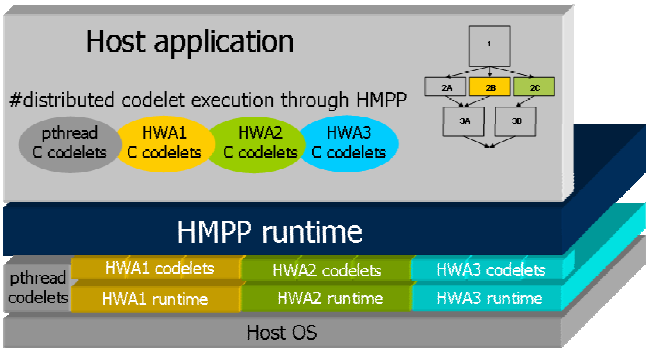

Figure 2: HMPP Integrated Application User View.

As depicted in the Figure 2, the proposed approach is to declare, by means of HMPP directives, functions (named codelets) which are suitable for hardware acceleration. The directives also specify the conditional execution of the codelets, their desired synchronous or asynchronous properties and the data transfers.

Many new programming environments for hybrid multi-core systems have been proposed [2][3][7]. These environments extend in one way or another, current programming standards such a C/C++, OpenMP, etc. Most of them rely on a stream programming style but a program written for a given platform cannot run on another one.

HMPP takes a radically different approach. A HMPP application can be compiled with an off-the-shelf compiler and run without any specific runtime to produce a conventional native binary. More over, thanks to its dynamic linking mechanism, a HMPP application is able to make use of either a new accelerator or an improved codelet without having to recompile the application source.

This way we aim at preserving legacy codes and insulate them from frequent hardware platform changes that tend to characterize hybrid multi-cores, e.g. fast GPU architecture evolution.

2) Integration with Third-party Tools

The HMPP directives address the remote execution of a piece of code as well as the download and upload of data to/from the hardware accelerator memory if it is different from the host CPU memory. HMPP can be seen as a programming glue between target specific programming environments and general purpose programming.

For instance, targeted to CUDA, HMPP expresses which parts in the application source should be executed in a NVIDIA card. The NVIDIA codelets are written in CUDA in a specific file while keeping the original computation in the main source. The developer also uses the NVIDIA provided tools such as the runtime and the compiler to program the codelets. The same applies to RapidMind target: the codelet may use the RapidMind vector types and needs to be linked against the RapidMind runtime. Contrarily to those solutions that support only one target at a time, HMPP is able to handle different accelerator targets in the same application.

The HMPP runtime handles the concurrent execution of the codelets that have been previously translated into the vendor programming model [2][4], either by hand or with an available code generator[2]. HMPP defines a standardized hardware specific interface between the runtime and the codelet implementation.

The HMPP toolkit comes with a C (and soon Fortran) preprocessor to translate the directives into appropriate sequences of calls to the HMPP runtime.

Figure 3 illustrates the build process of a HMPP application. The hardware version of the codelets is written using the specific vendor programming language. The application is firstly preprocessed with the HMPP preprocessor and linked with the HMPP runtime. The hardware versions of the codelets are separately compiled using the vendor programming tools and runtime support. The codelet is then transformed into a dynamic shared object file with the host compiler.

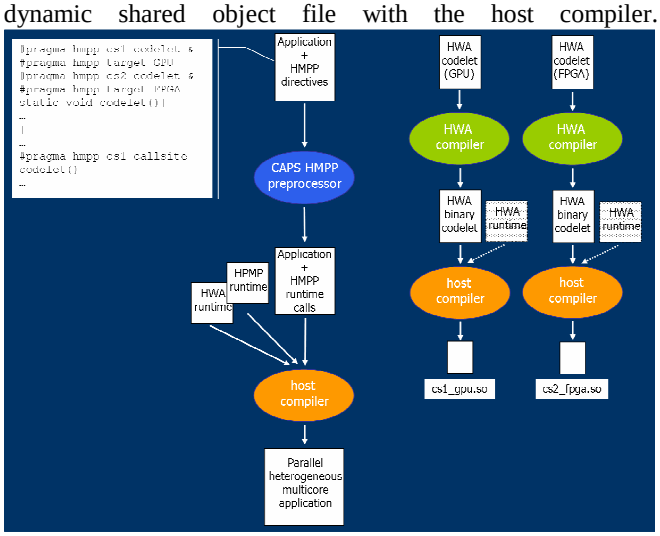

Figure 3: Build Process Overview.

This paper is organized as follows: section 2 presents the set of HMPP directives with simple examples. Section 3 describes the runtime functioning and the last section shows how HMPP handles homogeneous as well as distributed memory systems.

II. HMPP Programming Interface

HMPP directives are used to annotate the original code with instructions to execute a routine, the codelet, on a hardware accelerator.

An example of HMPP annotation is given in Extract 1. A codelet is a pure function, i.e. a function that always evaluates the same result value given the same argument value(s), and has no side effects and no I/O. Because argument values must be transferred into the accelerator, there are constraints on the codelet arguments. The argument coding rules permit to

[2] For instance, the CAPS tuner (http://www.caps-entreprise.com/) family of generators already target OpenGL and CUDA codelet programming.

compute the amount of data to transfer runtime.

```
#pragma hmpp trivial codelet, output=outv
  void trivial(int n, float a,
    float *inv, unsigned int N1[1],
    float *outv, unsigned int N3[1]) {
    int i, j;
    for (i = 0 ; i < n ; i++) {
      outv[i] = a * inv[i];
    }
  }
```

Extract 1: Trivial Codelet Definition.

Arguments are alias free (scalar arrays or pointers). Array and pointer arguments are followed by an array of unsigned integer argument that gives the size of each dimension of the array or pointer argument. For instance an argument A[][] is followed by a one dimensional array argument whose elements 0 and 1 give the size of the first and second dimensions of A.

The types of HMPP directives are the following:

- **Codelet**: define a function as hardware-assisted.
- **Execution**: specify the codelet remote execution in the program.
- **Data transfers**: data can be uploaded before the execution of the codelet and data download points can be inserted after the execution.

All directives are labelled with an identifier corresponding to exactly one codelet call site.

1) Codelet definition Directive

A codelet directive declares a computation to be remotely executed on a hardware accelerator. The favoured type of hardware accelerator can optionally be indicated, otherwise HMPP will take the first available compatible accelerator. For an efficient use of the accelerator, the computation can be specialized to the type of the parameters, their size, value etc. A parameter can either be read, written or both. They can be scalar or not. If not, their size argument must follow. A type conversion can also be inserted when copying the data on the hardware accelerator. A C or Fortran boolean expression can be specified in order to guard the execution of the codelet.

More than one codelet directive can be added to a function in order to specify different uses or different execution contexts.

The hardware-assisted corresponding codelet can either be produced using an appropriate code generator or hand-written in the vendor programming model. The argument values are used to generate specialized versions of the codelet for various accelerators. When the codelet is called, the appropriate variant of the code is selected according to the actual parameter values and performance issues. If an accelerator is not available or busy the original code of the codelet is used instead.

2) Remote Codelet Execution Directive

The remote codelet execution directive specifies how to use a codelet at a given point in the program. An example is shown in Extract 2. Data transfers and synchronization related to this call of the codelet both use the same label.

The default behaviour implies to upload the argument data before the call and to transfer them back after. This behaviour can be changed using the data transfer directives.

If the condition or the parameters values specified in the codelet definition are not checked, or if the hardware accelerator is not available, the original CPU codelet is used instead. The codelet execution can be asynchronous, for instance, it allows for concurrent execution between the CPU and the GPU.

```
#pragma hmpp trivial callsite
    trivial(n, 2.f, inc, N1, outv, N1);
```

Extract 2: Trivial Call Site Example.

3) Data Transfer Directives

Data can be uploaded before the execution of the codelet and data can also be declared constant so as to load them only once. This is particularly useful when the codelet is executed in a loop.

Data synchronization barriers can also be added for all asynchronous data retrieved back from the hardware accelerator. The program execution will block until all transfers are completed.

4) Synchronization Directive

When a codelet execution has been declared asynchronous, it is possible to use a synchronize directive to block the program until the corresponding codelet execution is complete. The hardware accelerator is then released and becomes available to another codelet.

III. Runtime Support

The HMPP runtime is in charge of carrying out the concurrent execution of the native and hardware versions of the codelets. It is also responsible for handling the exceptions such as the execution of a codelet whose input data have been indicated pre-fetched but not preloaded. This is the case, for instance, if the preload directive has been inserted in a path not taken up to the codelet call site directive (i.e. a program point with the directive was not reach during execution).

At execution, the HMPP runtime takes care to discover the available hardware accelerators. When a codelet is indicated to be run on a hardware accelerator, if the device is available and if the shared library corresponding codelet is present, HMPP loads it just as a software plug-in. Otherwise the native version is run on the host CPU or in a worker thread.

Note that it is not necessary to build a machine specific version of the binary. As long as the host CPU is identical, the application will make use of the available accelerators. The HMPP runtime is able to run several and different hardware accelerators.

Also if an improved version of a codelet is made available, there is no need to recompile the overall application source, HMPP will just load it in place of the previous codelet implementation. Moreover, if a new hardware accelerator has been made available, a computation that has been indicated as hardware-assisted with no restriction on the type of accelerator can make use of that new device without the need to recompile the application. As the HMPP application is dynamically linked against the runtime support, only that piece of software provided by CAPS needs to be updated to take the accelerator into account.

The following code example defines a codelet in charge of running the simple function:

```
  #pragma hmpp simple codelet, inout=outv
  void simple(int n, int m,
              float *inv, unsigned int N1[1],
              float *inm, unsigned int N2[2],
              float *outv, unsigned int N3[1])
  {
    int i, j;
    for (i = 0 ; i < m ; i++) {
      float temp =  outv[i];
      for (j = 0 ; j < n ; j++) {
        temp += inv[j] * inm[i * n + j];
      }
      outv[i] = temp;
    }
  }
```

Extract 3: Simple Codelet Definition Example.

The corresponding call site is given below:

```
  #pragma hmpp simple callsite, asynchronous
      simple(n, m, myinc1, N1,
             inm, N2, myoutv1, N3);
      simple(n, m, myinc2, N1,
             inm, N2, myoutv2, N3);
  #pragma hmpp simple delegatedstore,
calleeArg=outv
```

Extract 4: Simple Codelet Use Example.

In this example, the simple function is called twice. Only the first call is candidate to be accelerated, so only that call will be offloaded to an accelerator or a worker thread.

At runtime, the sequence of events in the main thread of the application will be as follows:

- data for the input parameters of the first call will be submitted to the HMPP runtime for transfer to the dynamically chosen accelerator;
- asynchronous launch of the computation will be requested;
- the second call to the simple function will be executed;
- the main thread will then wait for the asynchronous execution of the first call to complete and will recover the data for the output parameter myoutv2.

If the dynamically chosen accelerator is capable of asynchronous execution (for instance, a simple worker pthread based accelerator), then both calls to simple will be executed in parallel.

HMPP is compatible with OpenMP and MPI. If a loop has been parallelized with OpenMP and contains HMPP directives, the application will fork threads that will make use of the available accelerators. If the vendor driver is able to handle the concurrent execution of several GPU, HMPP will use all the available accelerators.

HMPP also works with MPI: an already MPI parallelized binary can be spread over the nodes of a distributed system and will use the local node codelets to accelerate the computation of which it is in charge of.

IV. HMPP Examples

In this section we describe a few HMPP uses.

1) Specializing Data Transfers to Overcome Overhead

The benefit of offloading computations to hardware accelerators must be weighted against the overheads incurred by doing so. One of the most significant overhead for distributed memory accelerators is the cost of the data transfers between the host's memory and the accelerator's memory. For instance, constant data must be transferred only once. Non-constant data may not necessarily require to be transferred at every call, if the said data are only modified infrequently. Analyses of the updating patterns of the data and the insertion of the directives is outside the scope of this paper and can be made either by hand or with the automated tools Astex [9]. The ability to optimize the transfers is of paramount importance, as shown in code Extract 5.

```
  for (k = 0 ; k < iter ; k++) {
#pragma hmpp matvec callsite,
advancedload:calleeArg=inm,
advancedload:calleeArg=inv, asynchronous
    matvec(n, m, (inc+(k*n)), N1,
           inm, N2, (outv+(k*m)), N3);
#pragma hmpp matvec delegatedstore, calleeArg=outv
    if (k && !(k%RATECHANGE)) {
      for (i=0; i<m; i++) {
        inm[i*n + iter%m] = 0.1;
      }
#pragma hmpp matvec advancedload, calleeArg=inm
      uploadmat = 0;
    }
  }
```

Extract 5: Specifying Resident Data in the Accelerator.

In this example, a single matrix of 8192x8192 floating point elements is used in a sequence of 256 vector-matrix products. The matrix is updated every N iterations, N varying from 10 to 250. Every time the matrix is updated it is sent to the accelerator. The result for various rates is shown in the graph below, comparing the reference C implementation on an Intel Core 2 Duo processor and a straightforward CUDA based codelet running on a NVIDIA Quadro FX5600. As we can see, the less frequent the updates, the higher the speedup. The idea is not to show the performance but to demonstrate that HMPP enables the implementation of various communication strategies, leading to higher performance. The HMPP environment allows an easy, non-intrusive way of specifying when and where the data should be transferred in order to attain optimum performance for the entire applications.

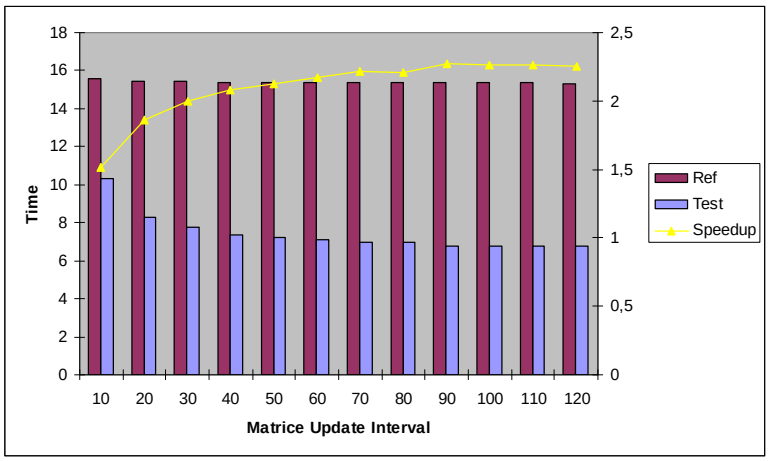

Figure 4: Data Transfer Overhead.

2) Codelet in Case of Shared Memory Systems

The HMPP environment, while designed for heterogeneous accelerators using a distributed memory model, can also take advantage of both homogeneous and/or shared-memory accelerators. One such example is the traditional symetric multi-processors system: HMPP can automatically extract a tagged function into an asynchronous codelet, whose execution is performed by a worker thread using the POSIX thread interface (pthread).

Another possibility is to use a fully-synchronous execution on a shared-memory system. In this case, the codelet itself will be responsible for exploiting the hardware resources. The best example here is the Intel® Threading Building Blocks [8], a "library that abstracts the low-level threading details necessary for optimal multi-core performance". In this case, the codelet is re-implemented using the TBB library to exploit SMP hardware The runtime will then synchronously (with regards to the main application's thread) execute the parallelized code. The function can therefore take advantage of the library's features such as the automatic spawning of worker threads on all available processors and dynamic partitioning, without the need for the application to become dependant on the library.

3) Lightweight MPI Backend

An interesting aspect of HMPP is the automatic generation of memory transfers to and from the tagged functions. Such memory transfers can be to, or from, not only dedicated hardware but also from any computing resource, including for instance a remote host. HMPP offers a virtual execution back-end based on the MPI (Message Passing Interface) library. Instead of running a codelet on the application host system, HMPP is able to run any codelet on a remote host. Rather than loading a local codelet, the HMPP runtime will use the MPI library to spawn a lightweight daemon on the remote host. This will ensure proper transfers to and from the daemon and will request the daemon to execute an instance of the codelet. The daemon itself is a HMPP application and will use the exact same mechanisms to execute an optimized codelet on the remote host. It exploits available resources such as GPUs, FPGAs or homogenous acceleration through e.g. the aforementioned Intel TBB library.

V. Conclusion

To our knowledge HMPP is the first programming tool that addresses hybrid multi-core programming with the aim of preserving the legacy code. On one hand, HMPP helps to exploit the potential power of the hardware accelerators by addressing, not only remote execution, but also data communication between the general-purpose cores and the specialized ones. On the other hand, HMPP preserves the legacy code while allowing the use of the target specific software development toolkit. The use of the currently available beta version of HMPP has demonstrated that very high performance can be achieved. A Fortran version of HMPP is currently being developed.

References

[1] NVIDIA developers site, CUDA homepage, http://developer.nvidia.com/object/cuda.html.
[2] RapidMind Corp. home page, http://www.rapidmind.net/.
[3] PeakStream Corp. home page, http://www.peakstream.com/.
[4] ATI CTM Guide, Technical Reference Manual, http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.
[5] Hypertransport consortium home page, http://www.hypertransport.org/.
[6] Romain Dolbeau and François Bodin, *HMPP description documentation*. http://www.caps-entreprise.com.
[7] Perry H Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang, “EXOCHI: Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System, PLDI’07.
[8] Intel® Threading Blocks, Intel®, http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm.
[9] A Hot Path Based Code Partitioning For Distributed Memory System on a Chip, http://www.irisa.fr/caps/projects/Astex/index.html