Scientific Programming 17 (2009) 325–335
DOI 10.3233/SPR-2009-0292
IOS Press

Heterogeneous multicore parallel programming for graphics processing units

Francois Bodin and Stephane Bihan *
CAPS entreprise, 4 allée Marie Berhaut, 35000 Rennes, France

Abstract. Hybrid parallel multicore architectures based on graphics processing units (GPUs) can provide tremendous computing power. Current NVIDIA and AMD Graphics Product Group 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, a Heterogeneous Multicore Parallel Programming workbench with compilers, developed by CAPS entreprise, that allows the integration of heterogeneous hardware accelerators in a unintrusive manner while preserving the legacy code.

Keywords: Heterogeneous programming, GPUs, compiler, runtime

1. Introduction

Using graphics processing units (GPUs) for scientific computing is a recent and fast evolving trend [4]. The evolution has been so fast that the usual general-purpose computing on graphics processing units (GPGPU) that usually refers to programming vertex and fragment shaders, is now obsolete.

Many phenomena have been at the origin of the use of GPUs in scientific computing. The first one is the evolution toward multicore architecture that doubles the number of cores instead of doubling clock frequency every 18 months. As a consequence, this has driven a programming effort toward parallel programming and has offered opportunities to hardware accelerator based approaches. Performance of GPUs are about 750 times higher than a decade ago.

The second one has been the introduction of programmable vertex and fragment shaders [12] that have exposed a very high potential computing power to programmers outside the graphics area.

The third phenomenon is new programming languages like NVIDIA CUDA, Brook+ or soon OpenCL that are based on a stream model more suited to scientific programming than the obscure OpenGL or DirectX standards used by GPGPU pioneers.

However, the main objective of these languages is to expose some of the specifics of the stream architecture in order to better exploit their performance. While CUDA and Brook+ are vendor-specific programming languages, the new OpenCL initiative aims at becoming a programming standard for a large range of available accelerators.

Based on compiler directives, HMPP (heterogeneous multicore parallel programming) offers a higher level of abstraction but still allows developers to fine tune the programming of accelerators. HMPP provides developers with a heterogeneous C and Fortran compiler with CUDA and SSE code generators. Hardware-specific codings are dissociated from the legacy code as additional software plugins. Contrary to applications that have been specifically written for a target architecture, HMPP produces applications that run on various hardware platforms whether an accelerator is present or not.

1.1. GPUs as hardware accelerators

GPUs[1] can achieve very high performance and growth is promised to be more than Moore's law [14]. Many applications achieve [16] a large fraction of peak performance.

In that case, the advantage of GPUs over general purpose cores is undoubtedly interesting. To provide performance benefit an application must have the following properties:

*Corresponding author. Tel.: +33 222 511 600; E-mail: Stephane.Bihan@caps-entreprise.com.

[1] NVIDIAs GeForce 8800 GTX GPU produces a theoretical maximum of 346 GFLOPS single precision. Current generation already reaches 1 Teraflops.

1. Massive data parallelism.
2. Compute intensive kernels that represent a very large fraction of execution time (cf. Amdahl's law [9]).
3. Arithmetic intensity/density that favors the use of the many computing units.
4. Simple regular/local memory accesses (i.e. not pointer tracking code, etc.) that can exploit the pipeline structure of the GPU board.

As a consequence of these requirements, in most cases, some efforts at the algorithm level are needed to reorganize the computations in order to adapt them to GPU computing. For linear algebra (Cholesky, LU, QR), efficiency of 80–90% of the peak speed has been reported [10,18] for large matrices which corresponds to a 3–5 speedup compared to a 2.4 GHz Core2 Quad Q6600.

Communication between CPUs and GPUs is one of the main bottleneck when using a GPU. Current PCI Express $\times 16$[2] is able to deliver up to 4 GB/s of peak bandwidth per direction (in practice 3.0–3.3 GB/s is observed). This is to compare to the 76 GB/s peak memory bandwidth per GPU Tesla C870 (an Intel Core 2 Duo has a 8.5 GB/s peak memory bandwidth). Transferring data over PCI Express has also a fairly high latency of about 15 microseconds [18] that makes it inefficient to use GPUs for small problems. Usually using "pinned CPU memory" achieves better performance for data transfers.

Overlapping communication and computation can hide all or part of latency of the data transfers leading to extra speedup but this is in some cases insufficient to solve the communication issue. A successful use of GPUs assumes that a large part of the data will remain on the GPU during the execution of an application.

1.2. Low level GPU programming

GPU programming relies on a data parallel paradigm, frequently denoted *stream* computing. Stream computing is mostly based on a map operation that consists in applying the same computation to all elements of a stream (e.g. an array). This programming model is a consequence of the internal GPU architecture that relies on SIMD instructions and a memory hierarchy that does not perform scatter operation efficiently (operation involving multiple indirection, e.g. `A[B[i]] = ...`). Stream computing can be efficiently implemented in a massively parallel fashion. However, stream parallelism does not apply to many application algorithms.

A stream is an array, with one or more dimensions, of homogeneous elements. Stream programming is based on the following operations:

Map: The map operation consists in applying a kernel function to all elements of a stream. The kernel function can use elements from multiple input streams.

Reduce: A reduction operation uses stream elements to compute a single value (e.g. dot product).

Two types of memory access operations can be performed:

Gather: A gather is usually of the form `x = S1[S2[i]]`. It assumes that a kernel can read any elements of a stream.

Scatter: A scatter memory operation looks like `S1[S2[i]] = ...`. This assumes that a kernel can write any elements of a stream. Former hardware architecture of GPUs did not allow an efficient implementation of this operation. Recent architectures alleviate this limitation. It should however be noted that if `S2[i]` is not a permutation of `S1` indices then the result of the operation is nondeterministic: if `S2[i] = S2[j] = x`, say, when `i != j` then `S1[x]` will be assigned more than once, in an undefined order.

1.3. Overview of the paper

This paper is organized as follows: Section 2 describes the HMPP programming directives, Section 3 covers the compilation of hybrid applications using HMPP, Section 4 argues over the need for a runtime while Section 5 presents performance results for some application kernels and Section 6 gives an overview of current related works.

2. HMPP programming

HMPP proposes a solution to not only simplify the use of hardware accelerators (HWA) in conventional general purpose applications, but also to keep the application code portable.

The goal is to integrate the use of HWAs rather than porting the application to make use of them. The chosen programming approach is similar to the widely

[2]See http://www.pcisig.com/specifications/pciexpress/ or http://en.wikipedia.org/wiki/PCI_Express.

available OpenMP standard but designed to handle HWAs. The hardware-specific versions of the computations to be offloaded to a HWA are dissociated from the native application source code. As such, HMPP makes a programming *glue* between hardware-specific codings and standard programming languages.

Based on a set of directives, the HMPP Workbench provides developers with C and Fortran compilers, and a runtime library. It gives programmers a simple, flexible and portable interface for developing parallel applications whose critical computations are distributed, at runtime, over the available heterogeneous cores.

2.1. Codelet concept

HMPP is based on the concept of codelets, functions that can be remotely executed on a HWA.

A codelet has the following properties:

1. It is a pure function.
2. Its return value is `void`.
3. It does not contain `static` or `volatile` variable declaration.
4. The parameters are not `vararg`.
5. It is not recursive.
6. Its parameters are not aliased and can be copy-in or copy-out.
7. It does not contain `callsite` directives (i.e. remote procedure call to another codelet).
8. It does not contain any function calls such as library functions like `malloc`, `printf`, ...

Except for the aliasing property, all of these restrictions are checked by the HMPP compiler.

An example of a correct codelet candidate is shown in Listing 1, we assume here the parameters do not alias.

2.2. Directives for declaring and executing a codelet

The HMPP directives address the remote execution (RPC) of a codelet as well as the data transfers to and from the HWA memory if different from the host CPU memory.

```c
void codeletOk(int n,
               float v1[n],
               float v2[n],
               float v3[n]) {
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i];
  }
}
```

Listing 1. Correct codelet example.

By default, all the parameters are loaded in the HWA just before the RPC, and the main memory is updated when the RPC has completed. More directives are provided to upload and download data to and from HWAs before the remote execution of a codelet.

All the directives belonging to the declaration, execution, data transfers, etc., of a codelet are identified by a unique label. Below is the most trivial way of accelerating an application using only two directives: a `codelet` directive to declare a function as a codelet, and a `callsite` directive inserted before the function call to specify the potential use of the codelet.

In the following example (Listing 2), the `matvec` function is declared as a candidate for CUDA hardware acceleration. HMPP should produce the CUDA version of the function. The `args` parameter of the directive indicates that the `outv` parameter is used as input and output. By default, all parameters are input.

In this example, the device allocation, data upload, codelet execution and result download are performed at the call site. If the codelet is called in a loop, this leads to overhead that might inhibit the performance offered by HWAs.

2.3. Data transfers directives to optimize communication overhead

When using a HWA, the main bottleneck is often the data transfers between the HWA and the main processor. To limit the communication overhead, data transfers can be overlapped with successive executions of the same codelet by using the asynchronous property of the HWA. For this, two directives can be used:

1. The `allocate` directive locks the HWA and allocates the needed amount of memory.
2. The `advancedload` directive prefetches data before the remote execution of the codelet. Moreover, if the data variable is declared constant (const qualifier in the directive parameter) then it is loaded only once for all codelet executions and reused as long as the HWA is not released.
3. The `delegatedstore` directive is a synchronization barrier to wait for an asynchronous codelet execution to complete and to then download the results.

In the following example (Listing 3), the device initialization, memory allocation and upload of the input

```
#pragma hmpp simple codelet, args[outv].io=inout, target=CUDA
   static void matvec(int sn, int sm,
                      float inv[sm], float inm[sn][sm], float *outv){
     int i, j;
     for (i = 0 ; i < sm ; i++) {
       float temp = outv[i];
       for (j = 0 ; j < sn ; j++) {
         temp += inv[j] * inm[i][ j];
     }
  outv[i] = temp;
   }

   int main(int argc, char **argv) {
     int n;
     ........
#pragma hmpp simple callsite, args[outv].size={n}
   matvec(n, m, myinc, inm, myoutv);
   ........
}
```

Listing 2. Codelet and call site HMPP directive use example.

data are done only once outside the loop and not in each iteration of the loop.

The synchronize directive allows to wait for the asynchronous execution of the codelet to complete before launching another iteration. Finally the delegatedstore directive outside the loop uploads the sgemm result.

3. Compiling hybrid applications

In terms of use, the HMPP compiler workflow is really close to traditional compilers with two main passes as illustrated in Fig. 4: one that builds a standalone application running on the host processor and a second pass for producing the accelerated versions of the codelets as dynamic shared libraries.

3.1. Host application compilation

As shown in the left part of Fig. 4, the HMPP preprocessor translates the directives into calls to the HMPP runtime in charge of managing the execution of the codelets. The preprocessed application is then compiled using the generic host compiler. Note that the host application can run standalone without hardware accelerators.

3.2. Codelet generation

The right part in Fig. 4 illustrates the production of the codelets as shared libraries in order to be loaded by the HMPP runtime. The template generator produces the codelet skeleton that contains all the runtime callback functions such as HWA allocation, data transfers, etc.

The developer can either write the codelet kernel in the hardware programming language or let the target generator produce it automatically. The codelet binary is finally produced using the hardware-vendor compiler.

3.3. Codelet generation directives

The HMPP codelet generators take Fortran and a subset of C99 as input code. This restriction aims at ensuring a large portability of the code on most HWAs. For instance, allowing pointer arithmetic would forbid the generation of code for many hardware platforms.

Codelet generation can be further improved with the use of codelet generation directives which help the generator optimize the produced target code for specific platforms, such as CUDA.

For instance, the loop unroll and jam transformation is intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main memory accesses and allows better exploitation of some machine instructions.

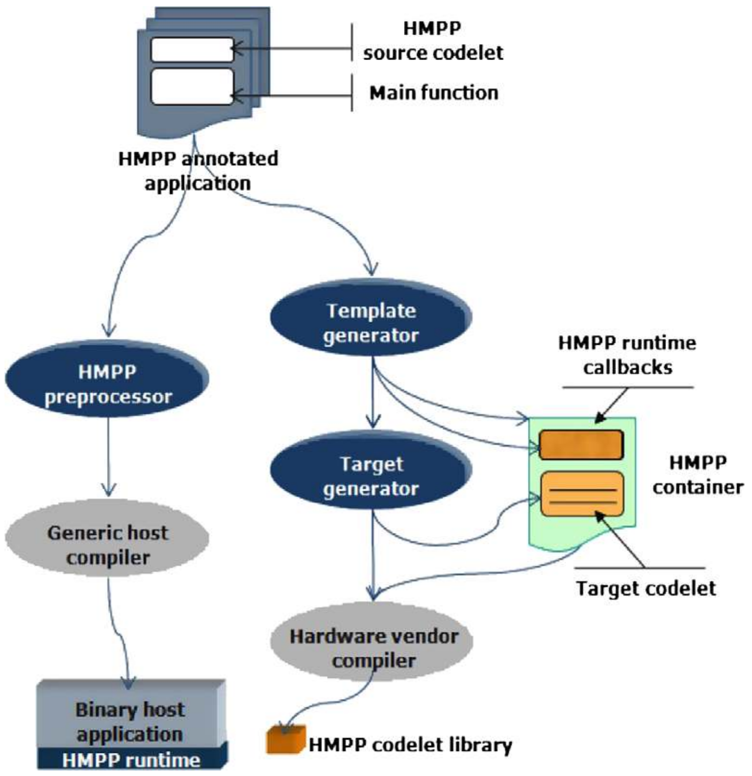
```
int main(int argc, char **argv) {

#pragma hmpp sgemm allocate, args[vin1;vin2;vout].size={size,size}
#pragma hmpp sgemm advancedload, args[vin1;vin2;vout] &
#pragma hmpp sgemm advancedload, args[m;n;k;alpha;beta]

  for( j = 0 ; j < 2 ; j++ ) {
#pragma hmpp sgemm callsite, asynchronous, &
#pragma hmpp sgemm args[vin1;vin2;vout].advancedload=true &
#pragma hmpp sgemm args[m;n;k;alpha;beta].advancedload=true
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
#pragma hmpp sgemm synchronize
}

#pragma hmpp sgemm delegatedstore, args[vout]
#pragma hmpp sgemm release
```

Listing 3. Advanced asynchronous HMPP programming.



Listing 4. HMPP compilation flow.

4. Runtime library

Hardware resources management is a critical issue that is not tackled by current exploitation systems. Supercomputers have been avoiding this problem by running only one application at a time or by partitioning the machine nodes. Unfortunately, programming and resource allocation cannot be considered separately. Sharing a GPU between applications, when it does work, usually results in very poor performance due to context switching on the device.

Current available GPUs leave to the user the sharing of a given device. This is fine when the nodes of an HPC machine are partitioned among users running a single application. However, in less controlled environments (HPC workstations, laptops) this is an issue

```
!$HMPP sgemm3 codelet, target=CUDA, args[vout].io=inout
SUBROUTINE sgemm(m,n,k2,alpha,vin1,vin2,beta,vout)
INTEGER, INTENT(IN)  :: m,n,k2
REAL, INTENT(IN)  :: alpha,beta
REAL, INTENT(IN)  :: vin1(n,n), vin2(n,n)
REAL, INTENT(INOUT) :: vout(n,n)
REAL    :: prod
INTEGER :: i,j,k
!$HMPPCG unroll(8), jam(2), noremainder
!$HMPPCG parallel
DO j=1,n
  !$HMPPCG unroll(8), splitted, noremainder
  !$HMPPCG parallel
  DO i=1,n
    prod = 0.0
    DO k=1,n
      prod = prod + vin1(i,k) * vin2(k,j)
    ENDDO
    vout(i,j) = alpha * prod + beta * vout(i,j) ;
  END DO
END DO
END SUBROUTINE sgemm
```

Listing 4. Use example of the `unroll` and `jam` directive.

that has to be addressed at run-time level.

The HMPP runtime handles the execution of codelets for different and various hardware (GPUs, SIMD units, FPGAs) but also for specific execution contexts. The appropriate execution hardware is selected at run-time depending on the system configuration, the resource availability and data dependent conditions. If a hardware accelerator is not present or not available, HMPP runs the native codelet function on the host system instead. If a codelet is attempting to run on a hardware, let say GPU, that is locked by another codelet, HMPP will execute it in the second GPU if there is one, then in the second target the codelet has been declared for, if any, or lastly in the host system otherwise.

5. Experimental results

In this section we give the performance of a few C and Fortran kernel examples accelerated and compiled with HMPP. The used target platform is an Intel Core 2 Duo CPU@2,66GHz and an NVIDIA Tesla C1060. The kernels running on the host system have been compiled using the latest Intel icc and ifortran compilers with `-O3` compiler flags.

The four codelets are given:

SGEMM: The performance of the matrix multiplication generated with HMPP is given in Fig. 2. It compares to the performance of the Intel MKL library implementation running on four cores.

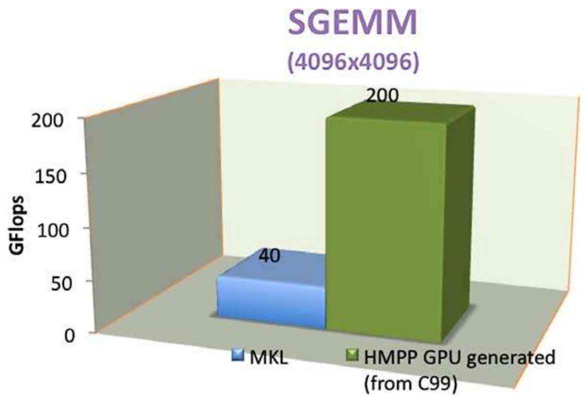


Fig. 2. Performance results for a MxM codelet.

As we want to show the raw performance of the HMPP generated code, these figures do not include the communication time. However, for large matrices, the data transfers overhead is marginal. The code is given in Listing 5.

Convolution: This linear algebra operation is widely used in signal processing. The performance of the generated code is given in Fig. 3.

Black–Scholes: Black–Scholes is a partial difference equation which model is used in finance. The performance of the generated code is given in Fig. 4.

Sobel: Sobel [11] filter is an image processing algorithm. The performance of the generated code is given in Fig. 5.

The sizes of the input data are given in each figure.

```
typedef struct{ float r, i;} Complex;
#pragma hmpp convolution2d codelet, args[data; opx].io=in, args[convr].io=out, target=CUDA
void convolution2d( Complex *data, int nx, int ny, Complex *opx,
   int oplx, int oply, Complex *convr) {
   int hoplx = (oplx+1)/2; int hoply = (oply+1)/2;
   int iy, ix;
#pragma hmppcg parallel
   for (iy = 0; iy < ny; iy++) {
#pragma hmppcg parallel
      for (ix = 0; ix < nx; ix++) {
         float dumr =0.0, dumi = 0.0; int ky;
         for(ky = -(oply - hoply - 1); ky <= hoply; ky++) {
            int kx;
            for(kx = -(oplx - hoplx - 1); kx <= hoplx; kx++){
               int dx = min( max(ix+kx, 0), (nx - 1) );
               int dy = min( max(iy+ky, 0), (ny - 1) );
              dumr += data[dy * nx + dx].r * opx[(hoply - ky) * oplx + (hoplx - kx)].r;
              dumr -= data[dy * nx + dx].i * opx[(hoply - ky) * oplx + (hoplx - kx)].i;
              dumi += data[dy * nx + dx].r * opx[(hoply - ky) * oplx + (hoplx - kx)].i;
              dumi += data[dy * nx + dx].i * opx[(hoply - ky) * oplx + (hoplx - kx)].r;
            }
         }
         convr[iy*nx+ix].r = dumr; convr[iy*nx+ix].i = dumi;
      }
   }
}
```

Listing 5. Convolution codelet.

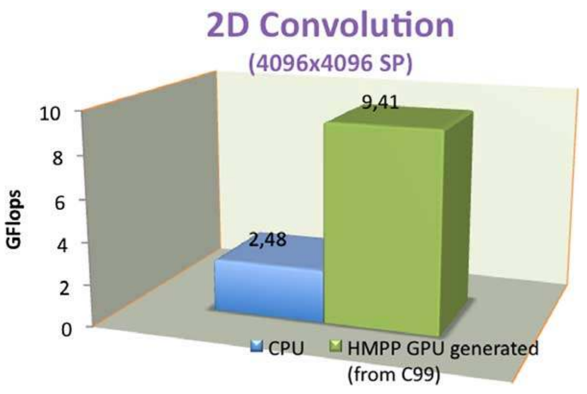


Fig. 3. Performance results for a convolution codelet.

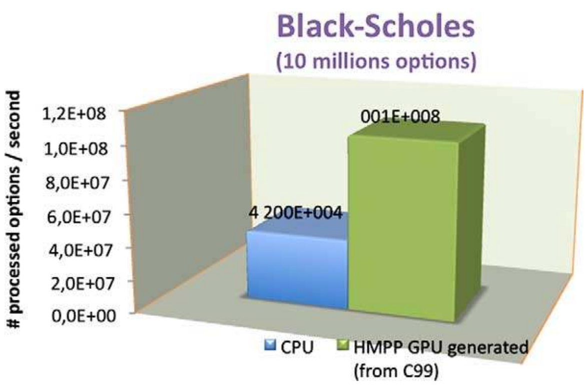


Fig. 4. Performance results for a Black–Scholes codelet.

6. Related works

Many stream programming languages have been proposed to program GPUs [7,8,13,15,17]. There is also quite a number of software solutions. In this section, we briefly expose these solutions and compare them to HMPP.

6.1. *PGI code generation*

PGI has been developing x86 compilers for years. They have recently announced the programming of GPUs as part of a pre-release compiler feature [5].

The developer also uses directives to annotate sections of the application to offload to a GPU (Listing 6). All communications rely on the compiler analyses to determine when to prefetch data and to download the result back from the GPU. Also all the GPU programming parts should not be open to the developer and would therefore not be tuneable.

6.2. *NVIDIA CUDA*

CUDA [3] organizes the computation in a hierarchical model that can easily be mapped on the NVIDIA multiprocessor architecture:

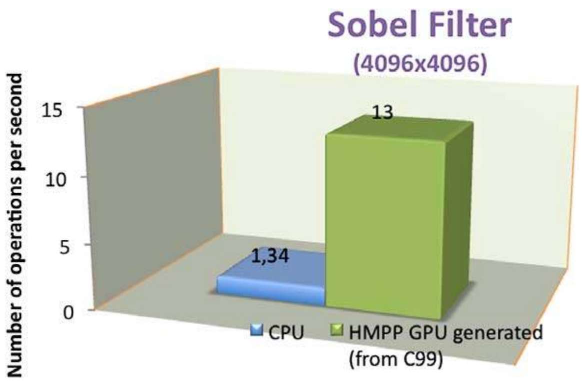


Fig. 5. Performance results for a Sobel codelet.

```
module mymm
  contains
  subroutine mm1( a, b, c, m )
    real, dimension(:,:) :: a,b,c
    integer i,j,k,m
    !$acc region
        do j = 1,m
          do i = 1,n
            a(i,j) = 0.0
          enddo
          do k = 1,p
            do i = 1,n
              a(i,j) = a(i,j) + b(i,k) * c(k,j)
            enddo
          enddo
        enddo
    !$acc end region
end subroutine
end module
```

Listing 6. PGI annotation example.

Grid: Organize the computation to be distributed on the multiprocessors.

Block: A group of threads to be executed on the same multiprocessors. This group of threads can access the same shared memory (see Listing 7).

Thread: A kernel function that is executed on all PE. A thread has a unique identifier built with the block identifier in the grid and a thread identifier in the block.

For instance, the following code shows an example of CUDA program. Variables v1, v2, v3 are arrays in the device memory. Line 13 creates the CUDA grid 4×1, line 14 creates 100×1 threads per grid element, line 27 shows how memory transfers are programmed. Function in line 1 is the kernel that will constitute the threads body.

6.3. AMD Brook+

Brook+ [1] is an implementation by AMD of the Brook GPU [2] language. It is an extension to the C-language. Brook+ is a flatter programming model compared to CUDA. It is based on a multidimensional stream data structure.

A simple Brook+ program is shown in Listing 8. Streams are declared in line 8, set in line 12 and read in line 15. Line 14 shows the mapping of the kernel function over all elements of the streams.

6.4. RapidMind

While RapidMind [6] is a development platform that comes with an Application Programming Interface and a runtime, HMPP is a workbench based on a set of directives that comes with code generation tools and a runtime.

RapidMind mainly addresses C++ applications with a set of RapidMind C++ classes. The application source code is semantically and syntactically modified to make use of the classes, thus becoming RapidMind dependent. The application is compiled using general-purpose compilers and linked with RapidMind runtime.

6.5. OpenCL initiative

OpenCL[3] is an initiative launched by Apple to ensure application portability across various GPUs. OpenCL aims at being an open standard (royalty free and vendor neutral) developed by the Khronos OpenCL working group.[4] OpenCL shares many features with CUDA and exposes data and task parallelism. It specifies accuracy of floating-point computations. The language for writing kernels is derived from ISO C99 with a few restrictions (recursion, function pointers, functions in C99 standard headers, ...). Beta version of OpenCL implementation from Apple is announced for Q1 2009.

[3] A good overview of the language has been provided at Sisgraph 2008 by Aaftab Munshi.

[4] See http://www.khronos.org.

```
__global__ void simplefunc(float *v1, float *v2, float *v3){
  /* compute array index blockIdx.y and threadIdx.y are 0
    since the y dimension of the grid and blocks is 1.
  */
  int i = blockIdx.x * 100 + threadIdx.x;
  v1[i] = v2[i] * v3[i];
}

int main(int argc, char **argv) {
  unsigned int n = 400;
  float *t1 = NULL, *t2 = NULL, *t3 = NULL;
  unsigned int i, j, k, seed = 2, iter = 3;
  dim3 grid(4,1);
  dim3 thread(100,1);
  t1 = (float *) calloc(n*iter, sizeof(float));
  t2 = (float *) calloc(n*iter, sizeof(float));
  t3 = (float *) calloc(n*iter, sizeof(float));
  CUT_DEVICE_INIT();
  ...
  // allocate gpu_tx buffers for memory transfer
  cudaMalloc((void**) &gpu_t1, n*sizeof(float));
  cudaMalloc((void**) &gpu_t2, n*sizeof(float));
  cudaMalloc((void**) &gpu_t3, n*sizeof(float));
  ...
  for (k = 0 ; k < iter ; k++) {
  // copy data from host memory to GPU
    cudaMemcpy(gpu_t2,&(t2[k*n]), n*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_t3,&(t3[k*n]), n*sizeof(float), cudaMemcpyHostToDevice);
    simplefunc<<<grid,thread>>>(gpu_t1,gpu_t2,gpu_t3);
  // copy data from GPU memory to host memory
    cudaMemcpy(&(t1[k*n]),gpu_t1, n*sizeof(float), cudaMemcpyDeviceToHost);
  }
...
return 0;
}
```

Listing 7. CUDA code example.

```
// kernel to be apply, on the GPU, to all elements of the stream
kernel void multiply(float a<>, float b<>, out float c<>) {
   c = a * b;
}

// host program
int main(intargc, char **argv) {
   float a<10, 10>, b<10, 10>, c<10, 10>;
   float A[10][10], B[10][10], C[10][10];
   // Initialize input arrays

   streamRead(a, A);
   streamRead(b, B);
   multiply(a, b, c);
   streamWrite(c, C);

}
```

Listing 8. Brook+ code example.

7. Conclusion

GPU computing coupled with Intel or AMD multicores are very cost-effective and promising high performance general-purpose heterogeneous hardware platforms. However, they today represent the perfect example of the dilemma developers have to face. GPUs can provide a 10x and more performance improvement but their programming is still quite challenging.

Parallel programming environments and languages represent a deep issue at the heart of the barely understood hardware/software interaction. Computer scientists have been tackling this issue, mainly for scientific computing, for years with mitigated success. For instance, High Performance Fortran that integrates many features for parallel computers (data distribution, ...), results from a large research community effort. This initiative has pushed forward compilation techniques but has not reached the market. Industry is also in a very difficult position. It has to balance the advantages between bringing an advanced programming technology that will target its own hardware approach (such as NVIDIAs CUDA language, for instance) at the cost of portability, and contributing to a standard that might help competition but increases the overall portfolio of parallel applications.

A consensus is emerging between major hardware constructors with the arrival of the OpenCL standard language that opens programming to many accelerators, but contrary to the C and Fortran high level standard languages, OpenCL will still require the developers to have a good understanding of the hardware architecture details to get performance.

There is also commercial solutions like the RapidMind platform offering a proprietary APIs that hides all the details of the GPU architecture. While the RapidMind platform makes the application portable accross GPUs from different vendors, it is locked to a software vendor. Moreover, fine tuning for performance the GPU programming is not possible.

Resources management is also an open question and critical issue that is not tackled by current exploitation systems. Supercomputers have been avoiding this problem by running only one application at a time or by partitioning the machine nodes. Unfortunately, programming and resource allocation cannot be considered separately. Virtualizing hardware resources is going to help but the performance issue remains. How to make sure that competition for hardware resources between applications will not degrade performance?

To address code portability as well as performance on heterogeneous multicore platforms, CAPS (http://www.caps-entreprise.com) proposes HMPP, a standard directive-based programming workbench that ensures code portability, offers a high level of abstraction for programming the GPUs and is open to fine tuning GPU programming by either plugin written kernels or library functions. Besides being a complete software development toolchain, HMPP also deals with resource management at run-time.

Even if the future of GPUs is not clear since graphic cores could be integrated on the processor die, they are surely a good example of how thousands of cores might be running and be used by applications. Different approaches are emerging and heterogeneous compilers are ready for takeoff.

References

[1] Brook+ sc07 bof session, amd, http://ati.amd.com/technology/streamcomputing/amd-brookplus.pdf.

[2] Brookgpu research project at the Stanford university, http://graphics.stanford.edu/projects/brookgpu/.

[3] Download cuda, http://developer.nvidia.com/object/cuda.htm.

[4] General-purpose computation using graphics hardware, http://www.gpgpu.org/.

[5] Pgi fortran & c accelerator programming model, http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf.

[6] Rapidmind corp. home page, http://www.rapidmind.net/.

[7] Sh: A high-level metaprogramming language for modern gpus, http://libsh.org/.

[8] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen and B. Chen, Future-proof data parallel algorithms and software on intelĺ multi-core architecture, Technical report, Intel Technology Journal.

[9] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: *Proc. AFIPS*, Vol. 30, AFIPS Press, Reston, VA, 1967, pp. 483–485.

[10] N. Galoppo, N.K. Govindaraju, M. Henson and D. Manocha, Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware, in: *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, Washington, DC, 2005, p. 3.

[11] A.K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[12] E.S. Larsen and D. McAllister, Fast matrix multiplies using graphics hardware, in: *Supercomputing'01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ACM, New York, NY, 2001, p. 55.

[13] W.R. Mark, R.S. Glanville, K. Akeley and M.J. Kilgard, Cg: A system for programming graphics hardware in a c-like language, in: *SIGGRAPH'03: ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, 2003, pp. 896–907.

[14] G.E. Moore, Cramming more components onto integrated circuits, *Electronics* **38**(8) (1965), 114–117.

[15] OpenGL, D. Shreiner, M. Woo, J. Neider and T. Davis, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*, Addison-Wesley, Reading, MA, August 2005.

[16] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn and T.J. Purcell, A survey of general-purpose computation on graphics hardware, *Computer Graphics Forum* **26**(1) (2007), 80–113.

[17] M. Strengert, C. Müller, C. Dachsbacher and T. Ertl, CUDASA: Compute unified device and systems architecture, in: *Proc. Eurographics Symposium on Parallel Graphics and Visualization (EGPGV08)*, Eurographics Association, 2008, pp. 49–56.

[18] V. Volkov and J. Demmel, Lu, qr and Cholesky factorizations using vector capabilities of gpus, Technical Report No. UCB/EECS-2008-49, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2008.