Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL

Roger Ferrer[1], Judit Planas[1], Pieter Bellens[1], Alejandro Duran[1], Marc Gonzalez[1,2], Xavier Martorell[1,2], Rosa M. Badia[1,3], Eduard Ayguade[1,2], and Jesus Labarta[1,2]

[1] Barcelona Supercomputing Center, Jordi Girona, 29. Barcelona, Spain
[2] Departament d'Arquitectura de Computadors, Univ. Politècnica de Catalunya, Jordi Girona, 1–3. Barcelona, Spain
[3] IIIA, Artificial Intelligence Research Institute, CSIC Spanish National Research Council, Spain
name.surname@bsc.es

Abstract. In this paper, we present OMPSs, a programming model based on OpenMP and StarSs, that can also incorporate the use of OpenCL or CUDA kernels. We evaluate the proposal on three different architectures, SMP, Cell/B.E. and GPUs, showing the wide usefulness of the approach. The evaluation is done with four different benchmarks, Matrix Multiply, BlackScholes, Perlin Noise, and Julia Set. We compare the results obtained with the execution of the same benchmarks written in OpenCL, in the same architectures. The results show that OMPSs greatly outperforms the OpenCL environment. It is more flexible to exploit multiple accelerators. And due to the simplicity of the annotations, it increases programmer's productivity.

1 Introduction

In this paper we present OMPSs, the programming model based on OpenMP and StarSs extensions, which can also include OpenCL or CUDA kernels, as a solution for easy programming of heterogeneous architectures. We have developed OMPSs to run on plain SMP machines, the Cell/B.E. processor, and SMP machines with GPUs. We are also making a port of the model for clusters.

From OpenMP we obtain high expressiveness to exploit parallelism using tasks. StarSs extensions allow runtime dependence analisys between tasks, and automatic data transfers. And OpenCL allows the programmer to easily write efficient and portable SIMD kernels to be exploited inside the tasks.

This approach is developed and evaluated for an Intel Xeon server (SMP, with 24 cores), a Cell/B.E.–based blade (with 2 Cell processors), and a machine with two NVIDIA GTX285 GPUs. Results show that OMPSs outperforms the equivalent code written in OpenCL, in the Xeon server, and the Cell/B.E. architectures. The proposal also outperforms the native CUDA implementations in NVIDIA GPUs.

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 215–229, 2011.

Our approach allows the same source code to run on all three architectures (currently, except for the CUDA implementation of the kernels in the NVIDIA architecture), showing that productivity and performance is achievable in these environments.

The rest of the paper is organized as follows: Section 2 shows the proposal presented in this paper and it introduces the benchmarks used in the evaluation of the proposal. Section 3 presents the evaluation of the proposal. Section 4 shows the comparison with the related work, and Section 5 concludes the paper, and presents our future work.

2 Proposal

Our proposal is to have a single programming model covering the different homogeneous and heterogeneous architectures in use today, and opened to future ones. To achieve this goal, we have proposed OpenMP extensions to deal with devices, data movement, and data dependences [3].

Using these extensions, applications are annotated with directives avoiding as much as possible runtime library calls. Avoiding library calls is important to keep the source code of the application clean, and still be able to compile and debug the functionality of the application in a serial manner.

2.1 Brief Description of the Programming Model

In the proposed OMPSs programming model, programmers use tasks to express parallelism, like in OpenMP 3.0. Task pragmas can annotate sections of code to be outlined as functions, or functions written by the programmer. A new pragma, *target*, is used preceeding a task or a user function to specify that their execution can be done in specific *device*s, and whether the data accessed by the task code should be *copy_in*, *copy_out*, or *copy_inout*, from the point of view of the accelerator. The task specification is completed with its needs for data *input*, *output*, or *inout*. The runtime system will then ensure that the task is not activated until the data has been produced (e.g. by another task), and the runtime system will activate dependent tasks when data is available.

As a summary of the syntax used, Listing 1.1 shows the grammar that our Mercurium compiler recognizes. Our support for accelerators currently includes the *smp*, *cell*, and *cuda* devices.

Listing 1.1. Grammar of the proposed OpenMP extensions

```
1  #pragma omp target device(devnam,...) [implements (function_name)] \
2      { [copy_deps] [copy_in(array_spec,...)] [copy_out(...)] [copy_inout(...)] }
3  #pragma omp task [input(...)] [output(...)] [inout(...)]
4      { function or code block }
```

In this paper we have used the benchmarks Matrix Multiply (AMD OpenCL SDK [2]), and BlackScholes, Perlin Noise and Julia Set (IBM OpenCL SDK [8]). In the next subsections, we explain how they have been rewritten using OMPSs.

2.2 Matrix Multiply

The Matrix Multiply benchmark works on 2 input matrices (A, B), and an input/output matrix C, that has been initialized to zero. All accelerators write part of the result on a block of C, after computing the result for this block based on the input blocks of A and B.

Listing 1.2 shows the code implementing this benchmark. Pragmas in lines 2, 8, and 14 annotate the alternative functions computing a block of the matrix, so that each invocation in line 26 gets created as a task. In lines 8 and 14, the *implements* clause indicates that the functions *matmul_block_cl* and *matmul_block_gpu* will be used in devices *cell* and *cuda*. The function at line 3 will be used in the SMP environment. The *copy_deps* clause in lines 8 and 14 indicates to the compiler that the same data areas checked for data dependences (*input/inout* clauses in line 2) should be moved into and out of the accelerators, in case of execution on the Cell SPUs or NVIDIA GPUs.

Listing 1.2. Annotated blocked Matrix Multiply. Each block is NBxNB float values

```
1  const int NB = 512;
2  #pragma omp task inout([NB*NB] C) input([NB*NB] A, [NB*NB] B)
3  void matmul_block(float * A,float * B,float * C)
4  {
5     // plain C kernel code for the SMP environment
6  }
7
8  #pragma omp target device(cell) copy_deps implements(matmul_block)
9  void matmul_block_cl(float * A,float * B,float * C)
10 {
11    // OpenCL kernel code
12 }
13
14 #pragma omp target device(cuda) copy_deps implements(matmul_block)
15 void matmul_block_gpu (float * A,float * B,float * C)
16 {
17    // CUDA kernel code
18 }
19
20 void matmul (int m, int l, int n, int mDIM, int lDIM, int nDIM,
21                float ** A, float ** B, float ** C)
22 {
23   for(i = 0;i < mDIM; i++) {
24    for (j = 0; j < nDIM; j++) {
25     for (k = 0; k < lDIM; k++) {
26      matmul_block (A[i*lDIM+k],B[k*nDIM+j], C[i*nDIM+j]);
27     }
28    }
29   }
30 #pragma omp taskwait
31 }
```

2.3 BlackScholes

The BlackScholes benchmark computes the pricing of European-style options. Its kernel has 6 input arrays, and a single output. Listing 1.3 shows the annotated code. In this example, the pragmas are the same as in Matrix Multiply, but instead of annotating a function, the annotation is done on inline code. Lines 1 to 12 show the parallel loop. It creates a task for each *work group* of iterations, in

the same way the OpenCL version does. Each task copies the 6 input parameters in (lines 3-5), executes the kernel (line 9), and at the end it copies the output array(*answer*) out of the accelerator (expressed in line 6).

The original version of this application was written in OpenCL. For the experiments with OMPSs in SMP and the Cell/B.E. processor we simply reuse the same OpenCL kernel code. To reuse the OpenCL code, we compile the code to an object file, and then the OMPSs task simply calls it as if it were a C function, as seen in line 9 of Listing 1.3. Listing 1.4 shows a portion of the OpenCL code. Observe the use of SIMD data types, *int4*, and *float4*, which direct the OpenCL compiler to use the SIMD units available, if any, in the processor cores. Observe also the expressiveness with respect to mathematical operators (addition, multiplication, etc.), and functions (log, sqrt) on SIMD data types.

Listing 1.3. Annotated BlackScholes

```
1  for (i=0; i<array_size; i+=work_group) {
2  #pragma omp target device(smp,cell,cuda) \
3      copy_in ( [work_group] &cpflag[i], [work_group] &S0[i], \
4                [work_group] &K[i],      [work_group] &r[i],  \
5                [work_group] &sigma[i],  [work_group] &T[i])  \
6     copy_out ( [work_group] &answer[i])
7  #pragma omp task shared (cpflag,S0,K,r,sigma,T,answer)
8     {
9        bsop_reference_float (&cpflag[i],&S0[i],&K[i],&r[i],
10                                  &sigma[i],&T[i],&answer[i]);
11    }
12 }
13 #pragma omp taskwait
```

Listing 1.4. BlackScholes OpenCL kernel. For reusing the OpenCL code, we compile the code to an object file, and then the OMPSs task simply calls it as if it were a C function

```
1  __kernel void bsop_reference_float(
2          int4 * cpflag, float4 * S0, float4 * K,
3       float4 * r,   float4 * sigma, float4 * T, float4 * answer)
4  {
5   float4 d1, expval, Nd1, Nd2, call, put;
6   ...
7   for (x=0; x<work_group; x++) {
8    d1 = log(S0[x]/K[x]) + (r[x] + HALF * sigma[x]*sigma[x])*T[x];
9    d1 /= (sigma[x] * sqrt(T[x]));
10   ...
11   call = S0[x] * Nd1 - K[x] * expval * Nd2
12   put =  K[0] * expval * (ONE - Nd2) - S0[x] * (ONE - Nd1);
13   answer[x] = bitselect(put, call, as_float4(cpflag));
14  }
15 }
```

For the GPU version, we have not been able to use this same technique yet, so we have translated the kernel into CUDA code. The translation is straightforward. And then we use a similar technique, shown in Listing 1.5 to exploit the CUDA code in the GPU from inside the task. We are currently working to overcome this limitation, and be able to use exactly the same OpenCL code also in the GPUs environment. Our compiler outlines the CUDA code, and then

compiles the task with the NVIDIA compiler for the GPU. Observe that in the GPU case, we use the plain, non-SIMD, data types, as those get better performance in the GPU.

Listing 1.5. BlackScholes CUDA code, annotated with the task pragma, and using the CUDA syntax for kernel launching. The kernel has been written using non-SIMD data types for better performance in the GPUs

```
1  #pragma omp target device(cuda) ...
2  #pragma omp task ...
3     {
4     dim3 dimBlock (local_work_group);
5     dim3 dimGrid  (work_group);
6     cuda_bsop <<<dimGrid, dimBlock>>> (
7                             cpflag, S0, K, r, sigma, T, answer);
8     }
9  ...
10 __global__
11 void  cuda_bsop ( int * cpflag, float * S0, float * K,
12      float * r, float * sigma, float * T, float * answer)
13 {
14  int x = blockIdx.x * blockDim.x + threadIdx.x;
15  float d1, expval, Nd1, Nd2, call, put;
16  ...
17  d1 = log(S0[x]/K[x]) + (r[x] + HALF * sigma[x]*sigma[x])*T[x];
18  d1 /= (sigma[x] * sqrt(T[x]));
19  ...
20  call = S0[x] * Nd1 - K[x] * expval * Nd2
21  put =  K[0] * expval * (ONE - Nd2) - S0[x] * (ONE - Nd1);
22  answer[x] = (cpflag[x])? put, call;
23 }
```

2.4 Perlin Noise

Perlin Noise has a single output, an image that is filled with noise to improve the realistic view of moving graphics, for example in games. Listing 1.6 shows the annotations used for Perlin Noise. Each task created by the loop starting in line 1, generates an horizontal slice of the image of height BS lines.

Listing 1.6. Annotated Perlin Noise

```
1  for (j = 0; j < img_height; j+=BS) {
2  #pragma omp target device(smp,cell,cuda) \
3                copy_out([BS*rowstride] optr)
4  #pragma omp task shared(optr)
5    {
6         // OpenCL / CUDA kernel
7    }
8  }
9  #pragma omp taskwait
```

2.5 Julia Set

Julia Set computes a series of images of the Julia Set fractal. Listing 1.7 shows the annotations used for this benchmark. Each task has one input, *julia_context*, a structure with the characteristics of the julia image to be generated. Among others, the image number to be generated, light position and intensity, and the spectator position, are passed in julia_context. As output, each task delivers a horizontal slice of the Julia fractal of height BS lines.

Listing 1.7. Annotated Julia Set

```
1 for (j = 0; j < img_height; j+=BS) {
2 #pragma omp target device(smp,cell,cuda) \
3         copy_in(julia_context) copy_out([BS*rowstride] image)
4 #pragma omp task shared(out,julia_context) \
5   {
6       // OpenCL / CUDA kernel
7   }
8 }
9 #pragma omp taskwait
```

As it can be observed, the benchmarks are easily annotated. Also, the proposed environment does not require to use the low level OpenCL or CUDA runtime calls to allocate and copy memory, compile the kernel code at runtime, or copy the results back to main memory. Our runtime system takes care of implementing memory allocation and data transfers and optimizing them.

3 Evaluation

This section presents the execution environments used for evaluation, and the evaluation of the benchmarks.

3.1 Execution Environments

Three execution environments have been used to evaluate the benchmarks:

- **Intel Xeon server.** This is a machine with 4 Intel Xeon chips, with 6 cores each, for a total of 24 cores. Each chip runs at 2.4 Ghz, and it has a L2 cache of 12Mbytes, shared among the 6 cores, and a peak of 9.6 Gflops. The machine has 48 Gbytes of main memory (RAM). In the Intel server, we run the AMD/ATI OpenCL SDK [2], and we compare its performance to the OMPSs environment. In this SMP environment OMPSs relies on the shared memory, and it avoids all data copies. On the contrary, the native OpenCL runs still do data copying. This is the main source of improvement in this platform.
- **Cell/B.E.** The QS20 Cell/B.E.–based blades contain two Cell/B.E. processors, running at 3.2 Ghz, each with a L2 cache of 512Kbytes. The blade has 1 Gbyte of main memory. Each Cell/B.E. has 8 SPUs, for a total of 16 in the blade. In the Cell/B.E. we run the IBM OpenCL SDK [8], and we compare its performance to the StarSs environment for the Cell processor [17]. In this environment, OMPSs is able to exploit higher coarse granularity in the data transfers. This is because the OpenCL environment has a limit on the size of each work–item, and thus in the associated data used to compute on it.
- **NVIDIA GPUs**
  The NVIDIA GPUs are in a host with dual-chip dual-core Opteron AMD processors. It has 8 Gbytes of main memory. Two NVIDIA GeForce GTX 285 GPUs are connected through the PCI bus. The GPU clock is 1.476 Mhz, it contains 240 CUDA cores, and it has 1 GBytes of global memory. We had also the opportunity to run two of the benchmarks in an Intel host

(dual chip, each with 4 i7 975 cores at 3.33Ghz, and 24 Gbytes of main memory) with a Fermi GTX480 GPU. In the GPUs environment, we run the NVIDIA OpenCL SDK [14], and we compare its performance to the current porting of the OMPSs environment for GPUs. In this environment, a general comparison is more difficult to do. The coding of the kernels in CUDA seems to go against good performance in OMPSs. Nevertheless, the results achieved are still good.

3.2 On an Intel Xeon Server

Figure 1 presents the evaluation of Matrix Multiply on 512x512 float matrices, on the Xeon server. It shows the OpenCL version, and two versions running with OMPSs. 512-nb stands for the non-blocked version, and 512-b for the blocked version. As it can be observed, the OMPSs blocked version outperforms the other two versions. Blocked versions achieve better data locality, and it is the reason for the better performance. As soon as the matrix size is increased from 512x512 elements, the performance obtained by the OpenCL environment heavily decreases, as shown in Figures 2 and 3. Initially, we thought that the reason could be that the data set could exceed the L2 data cache, but this is not the case. We suspect that the ATI OpenCL implementation has some problem when dealing with the transfers of large data sizes.

Figure 4 shows the speedup obtained on the BlackScholes benchmark for the OpenCL and OMPSs versions, running from 1 to 24 cores in the Xeon–based machine. Both the OpenCL and OMPSs versions on a single core achieve a speedup of 2.6 over the serial version due to the vectorization achieved through the SIMD OpenCL kernel. OMPSs is consistently better than OpenCL. We attribute this benefit to the fact that in OMPSs for SMP machines, we do not need to copy any data to work in parallel, while in OpenCL the data copies are done in the same way as for heterogeneous machines, as the data movement is coded in the application itself by the programmer.

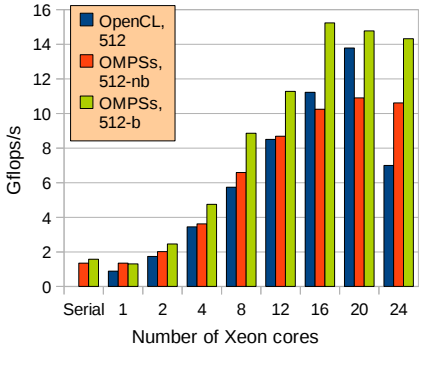

Fig. 1. Evaluation of Matrix Multiply (512x512) in an Intel Xeon server

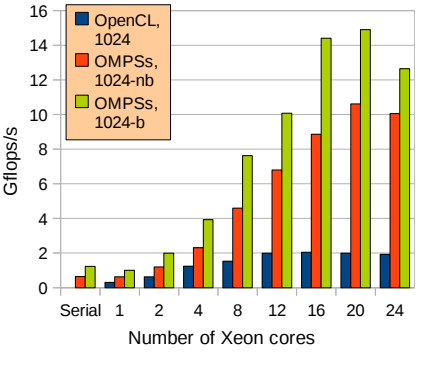

Fig. 2. Evaluation of Matrix Multiply (1024x1024) in an Intel Xeon server

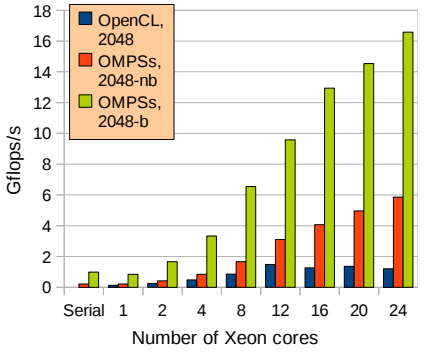

Fig. 3. Evaluation of Matrix Multiply (2048x2048) in an Intel Xeon server

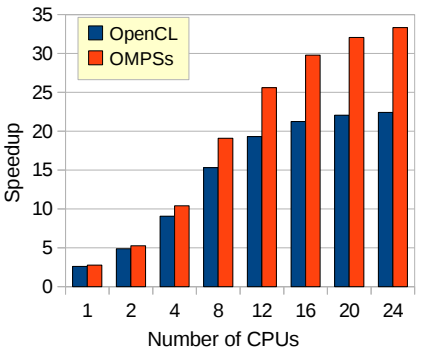

Fig. 4. Evaluation of BlackScholes in an Intel Xeon server

Figure 5 shows the results obtained from the Perlin Noise benchmark. The plot shows millions of pixels per second processed by the benchmark when running from 1 to 24 cores. In this case, we compared the performance of the OpenCL kernel with the C code compiled with gcc. We found out that gcc is achieving better performance on the inner kernel of this application. Our opinion is that for some reason gcc is able to exploit better the SIMD units of the Xeon processor. This situation does not happen in the other benchmarks. This shows the importance of being able to reuse the best code generated for an application, as we can do with OMPSs. This is something that it is not immediate for the OpenCL environment, as in it the kernel is compiled during runtime, and it will not be easy to have gcc generating the code for the kernel at runtime.

Figure 6 shows the results obtained from the Julia Set benchmark. The plot displays millions of pixels per second, obtained when running the benchmark from 1 to 24 cores. From the results obtained we also conclude that our OMPSs approach is performing better with respect the OpenCL version of the benchmark, being much easier to code.

3.3 On the Cell/B.E. Processor

In the Cell processor environment, we have used the CellSs flavor [17] of OMPSs. In this environment, we have found that the OpenCL Matrix Multiply benchmark achieves very poor performance compared to the hand–tuned SDK version, so that it is not interesting to show results on this benchmark. We present the results obtained in BlackScholes, Perlin Noise and Julia Set.

Figure 7 presents the results obtained from the BlackScholes benchmark when run from 1 to 16 SPUs. The OpenCL version can be compiled using different techniques to execute the OpenCL kernel. Those techniques using the OpenCL intrinsic *async_work_group_copy* are the best behaving in the Cell processor. The bar labeled *OpenCL, awgc* shows the results when using OpenCL ranges and such intrinsic. The bar labeled *OpenCL, db* presents the results when using the OpenCL task approach and also such intrinsic, in this case to implement *double*

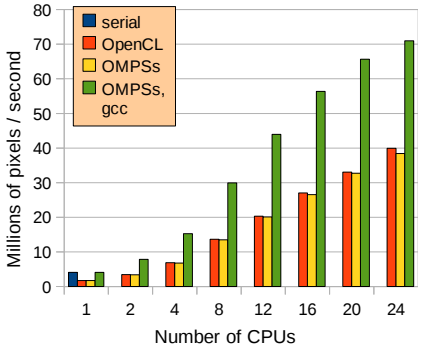

Fig. 5. Evaluation of Perlin-Noise in an Intel Xeon server

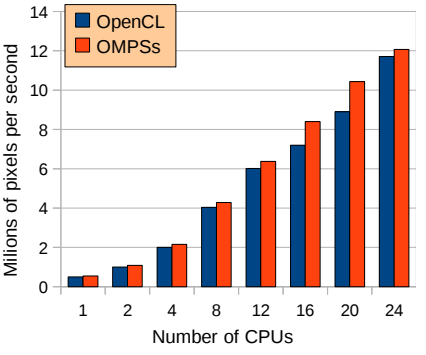

Fig. 6. Evaluation of Julia-Set in an Intel Xeon server

buffering. It is interesting to compare these results, as tasks with double buffering perform a little worse than using ranges and single buffering. This seems to be due to the overhead of creating the individual tasks in OpenCL, compared to the overhead of using OpenCL ranges.

The comparison of the OpenCL versions against OMPSs results highly beneficial for our approach, that reaches a 2x performance increase when running on 12 and 16 SPUs. This is again because our approach is able to reduce the number of data transfers. In OpenCL the application has to create special buffers to put the data on, an then the OpenCL kernel running on the SPUs issues the work group copies that trigger the SPUs DMA data transfers. This two–level copy is avoided in OMPSs, where the SPUs access the application data directly through the DMAs managed by the SPU runtime system.

Figure 8 shows the results obtained from the Perlin Noise benchmark. It shows millions of pixels per second, depending on the number of SPUs used. We also show two different alternatives for OpenCL, ranges with *asynchronous work group copies*, and tasks with *load/store* operations. For this benchmark, the latter performs better, but it is far from the performance of OMPSs. As in BlackScholes, the reduced number of copies done in OMPSs benefit performance.

Figure 9 compares the performance obtained from OpenCL and OMPSs on the Julia Set benchmark. In this case, the kernel of the benchmark is highly computational, so the impact of the data transfers is not so high. Nevertheless, the figure shows that OMPSs is having better scalability than OpenCL, as there is a consistent increase in the difference of performance between both approaches, as long as the number of SPUs is increased.

3.4 On NVIDIA GPUs

In this section we present the first evaluation of the OMPSs proposal on GPUs.

Figure 10 shows the results obtained in Matrix Multiply using the example in the CUDA SDK, the OpenCL example from the ATI SDK, and the OMPSs approach with the CUDA kernel. It presents Gflop/s, on matrix sizes of 512x512,

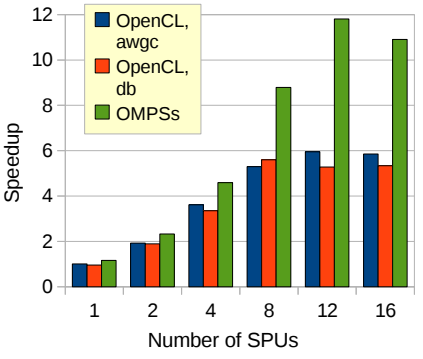

Fig. 7. Evaluation of BlackScholes in a Cell/B.E. blade

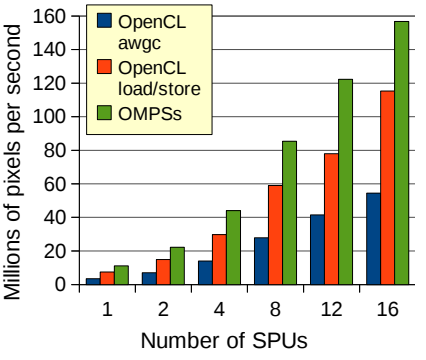

Fig. 8. Evaluation of Perlin-Noise in a Cell/B.E. blade

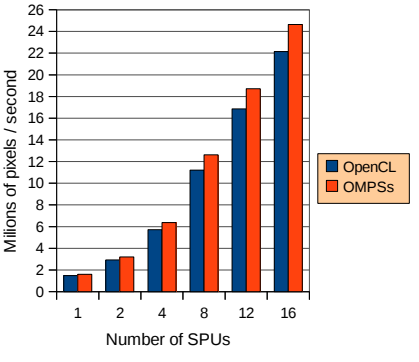

Fig. 9. Evaluation of Julia-Set in a Cell/B.E. blade

1024x1024 and 2048x2048, and in the case of OMPSs also using 1 and 2 GPUs. For the 512x512 matrix size, OMPSs performs very similarly to OpenCL, and both are a little bit better than CUDA. This is despite the fact that our OMPSs approach uses the CUDA kernel. For the 1024x1024 matrix size, OMPSs outperforms the CUDA and OpenCL environments. For the 2048x2048 matrix size, the three environments behave very similarly. In the NVIDIA CUDA, and OpenCL environments, it is not immediate to exploit more than one GPU. The source code of the application needs to be modified to access to the additional devices, and do the memory allocations and transfers.

Instead, with OMPSs it is our runtime system that accesses the number of devices indicated by the user through an environment variable (NX_GPUS, similarly to the traditional OMP_NUM_THREADS), and the application can automatically be exploited in several GPUs.

For matrix sizes larger than 1024x1024, OMPSs scales nicely when going from 1 to 2 GPUs. For the 1024x1024 matrix size and below, there is no gain in using

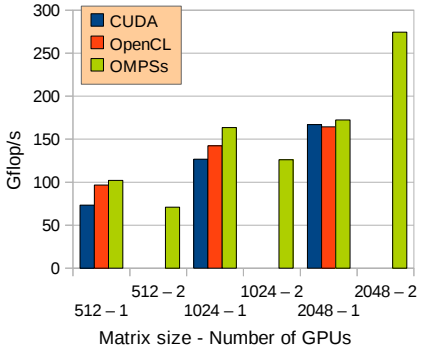

Fig. 10. Evaluation of Matrix Multiply in NVIDIA GTX285 GPUs

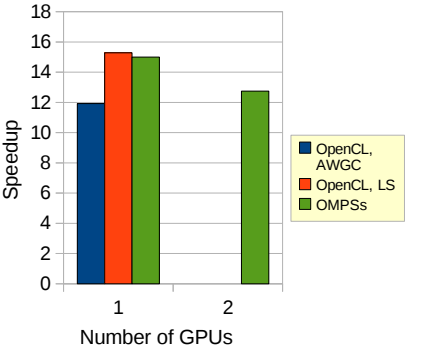

Fig. 11. Evaluation of BlackScholes in NVIDIA GTX285 GPUs

more GPUs because the data set is so small, and the overhead of having several data transfers to different devices is noticeable.

Figure 11 shows the speedup obtained from BlackScholes. It compares two alternative implementations of the OpenCL kernel, with the OMPSs approach. The latter uses the kernel translated to CUDA. As in the case of OMPSs on SMPs, the performance achieved is similar. OMPSs does not obtain any improvement when executing on two GPUs. This is because the data transfers dominate the execution time, as the parallel region has 6 input data arrays. Again, if we compare the results obtained in a single GPU we can state that the performance is the same, and the programming effort will be much less.

Figure 12 shows the results obtained from the Perlin Noise benchmark. Recently, we have had access to a NVIDIA Fermi GTX480 card, and we have executed Perlin Noise and Julia Set on it. The plot compares the performance obtained from the OpenCL and OMPSs versions of the benchmark, on one and two GPUs (two only for the GTX285). Looking at the evaluation on one GPU, we can appreciate that OMPSs outperforms OpenCL for both the GTX285 and the GTX480 GPUs. It is outstanding the increase of performance that the GTX480 hardware represents, and it is also outstanding the 2x speedup that OMPSs obtains in a single GPU over the OpenCL version of the benchmark. Finally, the benchmark scales from 1 to 2 GTX285 GPUs. We have not had access to hardware with two GTX480 GPUs yet.

Figure 13 shows the results obtained from the Julia Set benchmark. The plot shows the OpenCL and OMPSs versions of the benchmark on the GTX285 and GTX480 GPUs. The lower performance that OMPSs gets in the GTX285 GPU is due to the translation of the OpenCL kernel code into CUDA. We had to translate the OpenCL kernel code into CUDA because of the the limitation described in section 2.3, regarding the lack of ability to compile the OpenCL kernel code to be used in the OMPSs applications for the GPUs environment. It is interesting to note that even if the CUDA code obtains lower performance in

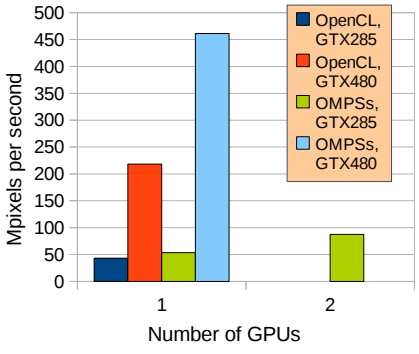

Fig. 12. Evaluation of Perlin Noise in NVIDIA GTX285 and GTX480 GPUs

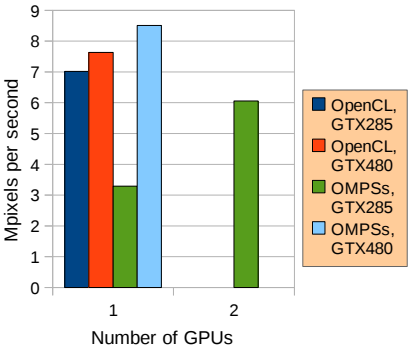

Fig. 13. Evaluation of Julia Set in NVIDIA GTX285 and GTX480 GPUs

GTX285 GPUs, OMPSs outperforms the OpenCL version of the benchmark in the GTX480 GPU. We think that this shows that our approach is well positioned for future GPU hardware. Notice also, that Julia Set scales nicely when executed on two GTX285 GPUs.

4 Related Work

New computer architecture designs based on heterogeneous multicores have raised the question about their programmability. The CAPS HMPP [6] toolkit is a set of compiler directives, tools and software runtime that supports parallel programming in C and Fortran. HMPP works based on *codelets* that define functions that will be run in a hardware accelerator. These codelets can either be hand–written for a specific architecture or be generated by some code generator. Offload [5] is a programming model for offloading portions of C++ applications to run on accelerators. Code to be offloaded is wrapped in an *offload* block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate thread. Call graphs rooted at an offload block are automatically identified and compiled for the accelerator. Data movement between host and accelerator memories is also handled automatically.

Recently, general purpose computation on graphic processors has received a lot of attention as it delivers high performance computing at rather low price. Major processor vendors have showed their intent to integrate GPUs as a GPU–core in the CPU chip [9,1]. CUDA [13] is an extension to C++ proposed by NVIDIA. It is based on *kernels* that are run n times in parallel by n threads. Tools to better map the algorithms to the memory hierarchy have been proposed [19]. They advocate for that programmers should provide straight-forward implementations of the application kernels using only global memory, and that tools like CUDA-lite will do the transformations automatically to exploit local memories.

Most of the programming models suitable for heterogeneous multicores allow to express some form of task based parallelism. OpenMP 3.0 [16], the industry standard for parallelism in shared memory machines, introduces a *task* suitable for parallelization of irregular applications [4]. The Sequoia [11] alternative focuses on the mapping of the application kernels onto the appropriate engines to exploit the memory hierarchy. RapidMind [18] is a development and runtime platform that uses dynamic compilation to accelerate code for the accelerators available, being those GPUs or the Cell SPUs. The programmer encapsulates functions amenable for acceleration into program containers. The code in containers is only compiled during the execution of the application, so that it can be optimized dynamically depending on the input data and the target architecture.

Merge [12] encapsulates specialized languages targeting specialized accelerators (GPUs, FPGAs) in C/C++ functions to provide a uniform interface for them. Encapsulation is based on EXOCHI [20], which uses pragmas to offload the domain specific language to be compiled with the compiler of the target device. Merge allows the specification of the same function for different targets, as new intrinsic functions, and it provides the mechanism for dynamic function selection at runtime.

We have found that all solutions make the programmer to split the application in pieces to provide the low level kernels to the acceleration engines. As we propose with OMPSs, we think that the compiler must be the responsible to address this issue. This will increase productivity in multicore processors, specially in the heterogeneous ones. This is also the case of Offload [5] and the IBM compiler [7,15], also known as Octopiler, which takes OpenMP code and places the parallel regions on the Cell SPUs. We propose to augment the C/C++ languages to incorporate vector types like OpenCL does. This will allow to easily obtain performance from OpenMP parallel regions and vectorization at the same time, which is currently difficult with current programming models.

5 Conclusions and Future Work

This paper presents OMPSs, a proposal to improve programming on multicore processors and GPUs. The proposal improves productivity and achieves a performance similar or better than existing environments based on CUDA/OpenCL.

OMPSs is based on program annotations taking the best features from the tasks of OpenMP, StarSs dependence analysis and automatic generation of data transfers, and the expression of SIMD operations in OpenCL kernel codes.

The OMPSs proposal is evaluated with four benchmarks: Matrix Multiply, BlackScholes, Perlin Noise, and Julia Set; and using three distinct architectures: an Intel SMP, an IBM Cell/B.E.–based blade, and NVIDIA GPUs.

Results show that OMPSs outperforms OpenCL/CUDA implementations of the same benchmarks, in the three execution environments. In addition, it achieves a unified way of programming them in different environments. We propose to augment the C/C++ languages to incorporate vector types. This will allow to exploit parallelization with OpenMP and vectorization at the same time.

Our future work is focused on the improvement of OMPSs, including research on task scheduling in heterogenous environments, automatic tuning of data transfer sizes, and task granularity. In addition, we are porting larger applications to the programming model, to show that it is applicable in a general way to a variety of algorithms.

Acknowledgments

This work utilized the AC cluster [10] operated by the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. The cluster was funded by NSF SCI 05-25308 and CNS 05-51665 grants along with generous donations of hardware from NVIDIA, Nallatech, and AMD.

We thankfully acknowledge the support of the European Commission through the ENCORE project (FP7-248647), the TERAFLUX project (FP7-249013), the TEXT project (FP7-261580), the SARC IP project (FP6-27648), and the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the support of the CSIC through the intramural project no. 200850I237, and the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), and the BSC-IBM MareIncognito project.

References

1. AMD Corporation. The AMD Fusion Family of APUs, `http://fusion.amd.com`
2. AMD/ATI. OpenCL: The Open Standard for Parallel Programming of GPUs and Multi–core CPUs (2010), `http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl.aspx`
3. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Orti, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009)
4. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., Zhang, G.: A proposal for task parallelism in openMP. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 1–12. Springer, Heidelberg (2008)
5. Cooper, P., Dolinsky, U., Donaldson, A.F., Richards, A., Riley, C., Russell, G.: Offload – automating code migration to heterogeneous multicore systems. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 337–352. Springer, Heidelberg (2010)
6. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A Hybrid Multi-core Parallel Programming Environment. In: Workshop on General Processing Using GPUs (2006)
7. Eichenberger, A.E., O'Brien, K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M., Archambault, R., Gao, Y., Koo, R.: Using advanced compiler technology to exploit the performance of the cell broadband engine$^{(tm)}$ architecture. IBM Systems Journal 45(1), 59–84 (2006)

8. IBM Corporation. OpenCL (2010), http://www.alphaworks.ibm.com/tech/opencl
9. Intel Corporation. Intel Unveils Product Plans for HPC (May 2010), http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm
10. Kindratenko, V., Enos, J., Shi, G., Showerman, M., Stone, G.A.J., Phillips, J., Hwu, W.: GPU Clusters for High-Performance Computing. In: IEEE Int. Conf. on Cluster Comp. Workshop on Parallel Programming on Accelerator Clusters (2009)
11. Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2007)
12. Linderman, M., Collins, J., Wang, H., Meng, T.: Merge: A Programming Model for Heterogeneous Multi-core Systems. In: Proc. of the 14th Int. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS) (March 2009)
13. NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Version 2.0 (2008)
14. NVIDIA Corporation. OpenCL (2010), http://www.nvidia.com/object/cuda_opencl_new.html
15. O'Brien, K., O'Brien, K.M., Sura, Z., Chen, T., Zhang, T.: Supporting openmp on cell. International Journal of Parallel Programming 36(3), 289–311 (2008)
16. OpenMP Architecture Review Board. OpenMP Application Program Interface. Version 3.0 (May 2008)
17. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM Journal of Research and Development 51(5), 593–604 (2007)
18. RapidMind. RapidMind Multi-core Development Platform, http://www.rapidmind.com/pdfs/RapidmindDatasheet.pdf
19. Ueng, S.-Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.-m.W.: CUDA-Lite: Reducing GPU Programming Complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)
20. Wang, P., Collins, J., Chinya, G., Jiang, H., Tian, X., Girkar, M., Yang, N., Lueh, G.-Y., Wang, H.: EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In: Proc. of PLDI, pp. 156–166 (2007)