Implementation of Stereo Matching Using High Level Compiler for Parallel Computing Acceleration

Jinglin Zhang, Jean François Nezan, Jean-Gabriel Cousin, Erwan Raffin

Implementation of Stereo Matching Using High Level Compiler for Parallel Computing Acceleration

Jinglin ZHANG Université
Européenne de Bretagne,
France
INSA, IETR, UMR CNRS 6164
20, Avenue des Buttes de
Coesmes, 35708 RENNES ,
France
jignlin.zhang@insa-rennes.fr

Jean-Francois NEZAN
Université Européenne de
Bretagne, France
INSA, IETR, UMR CNRS 6164
20, Avenue des Buttes de
Coesmes, 35708 RENNES ,
France
jnezan@insa-rennes.fr

Jean-Gabriel COUSIN
Université Européenne de
Bretagne, France
INSA, IETR, UMR CNRS 6164
20, Avenue des Buttes de
Coesmes, 35708 RENNES ,
France
jcousin@insa-rennes.fr

xxx Université Européenne
de Bretagne, France
INSA, IETR, UMR CNRS 6164
20, Avenue des Buttes de
Coesmes, 35708 RENNES ,
France
jcousin@insa-rennes.fr

ABSTRACT

Heterogeneous computing system increases the performance of parallel computing in many domain of general purpose computing with CPU, GPU and other accelerators. With Hardware developments, the software developments like Compute Unified Device Architecture(CUDA) and Open Computing Language (OpenCL) try to offer a simple and visualized tool for parallel computing. But it turn out to be more difficult than programming on CPU platform for optimization of performance. For one kind of parallel computing application, there are different configuration and parameters for various hardware platforms. In this paper, we apply the Hybrid Multi-cores Parallel Programming(HMPP)[1][2] to automatic-generates tunable code for GPU platform and show the result of implementation of Stereo Matching with detailed comparison with C code version and manual CUDA version. The experimental results show that the default and optimized HMPP have the approximative 1 compared with CUDA implementation. And the HMPP workbench can greatly reduce the time of application development using parallel computing device.

Keywords

HMPP, CUDA, OpenCL, Stereo Matching

1. INTRODUCTION

Stereo vision is a well suited technology which offer a precise description of 3D information[3]. Stereo Matching uses only two cameras and a processing unit to do the disparity matching and 3D reconstruction. In the same time, the exploitation of parallel programming tool on heterogenous system and architectures such as GPU, APU, FPGA and so on. For the GPU, there are great number of community employing the CUDA which proposed by Nvidia in 2007. Comparing with CUDA, Open Computing Language (OpenCL) provide better portable feature on multi hardware platform, make the heterogenous parallel computing possible not only on GPU platform[4]. Recently, Altera also propose their OpenCL solutions for FPGA in order to get the low power and quick to market[5]. In order to achieve the best performance in different platforms, we must follow some strict rules to transform the code form single core systems to heterogeneous or multi-core systems. All the procedures of transformation is complicated and much more redundancy. In our work, we use HMPP Workbench 3.0, a directive-based compiler targeted to GPUs and CPUs. Based on HMPP, programmers transform sequential code using paired directives to automatically generate CUDA or OpenCL kernels for various applications.

- codelet: routine implementation
- callsite: routine invocation

Basicly with HMPP, using only these two paired directives can replace the procedure of manual writing the complex CUDA or OpenCL kernel code. Scott proposed a method of auto-tuning and optimize the code with several applicable transformation configurations and then pick the optimized version of the code with the best performance[2]. Jianbin propose an auto-tuning solution to data streams clustering with OpenCL[6]. In our work, we convert the sequential C code to parallel versions of Stereo Matching kernel with optimized HMPP directives in order to obtain better performance compared with those default HMPP and manual CUDA kernels. These contributions of our work are:

- showing that automatic optimizing HMPP complier HMPP can be used to effectively parallelize and generate the GPU kernels.

- analyzing difference of performance of the HMPP CUDA (CUDA kernel generated by HMPP) and manual CUDA kernel

Section 2 elaborate the optimization strategies of convert C code to CUDA / OpenCL kernel code. Section 3 presents the Parallel Computing with HMPP. Section 4 presents the stereo matching algorithm and the implementation of HMPP. Section 5 illustrate the experiments results between the manual CUDA and HMPP CUDA. A brief conclusion is given in section 6.

2. TRANSFORMATION RULES

Before we use HMPP to transform C code to CUDA / OpenCL kernel code, we must acquaint the rules about programming with the parallel computing device. we present some optimization strategies for transformation used in our work.

2.1 Data Transferring

In the image or video processing, there are several hundreds even thousand MB data to process. So we should avoid the unnecessary data transferring between host and device. The best choice is transfer as much as possible date at one time to fully utilize the bandwidth of GPU.

2.2 Fully Saturate The Computing Resource

Using parallel computing language, programmers should think about the practical infrastructure of different platforms. Because no one cannot ensure that the same kernel has the best performance on different platforms. Supposing that NVIDIA's GPU support 512 active threads per Compute Unit and ATI's GPU support 1024 active threads per Compute Unit, our experiments use 256 threads as one group which means only 33.3% and 25% capability of Compute Unit be used in NVIDIA's and ATI's GPU separately. Therefore Programmers should organize as much as possible threads to fully employ the compute resource of device. In the Section 5, we illustrate the occupancy of our experimental device in different conditions.

2.3 Use shared memory

Because of on chip, the shared memory is much faster than the device memory. In fact, accessing the shared memory is as faster as accessing a register, and the shared memory latency is roughly 100x lower than the device memory latency tested in our experiments. To get maximum performance, the most important point is to use shared memory but avoid banks conflict. In order to run faster and avoid refetching from device memory, programmers should put the data into the local memory ahead of complex and repeated computing.

3. PARALLEL COMPUTING WITH HMPP

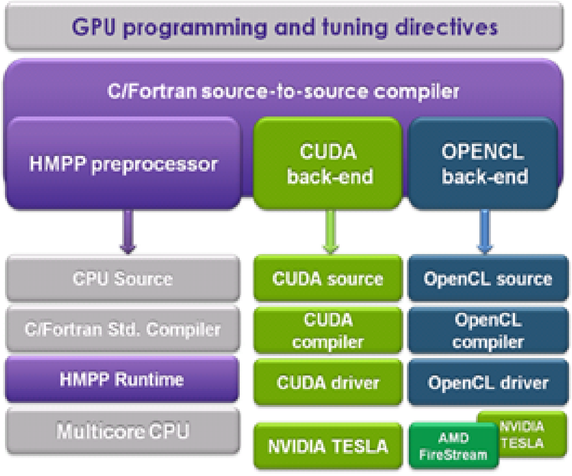

Figure 1: HMPP Workbench framework.

Recently more and more powerful multi-core intergraded architecture like Nvidia tegra3(Quad-Core CPU, 12 core GPU supporting 3D stereo)[7], Snapdragon S4 APQ8064(Quad-Core CPU, Adreno 320 GPU supporting OpenCL)[8] support the parallel programming model like OpenCL. In such All-in-one chips, GPU is not only play the part of rending and display, it also can be responsible for the general scientific computing based on their powerful computing capacity and the greater bandwidth. The HMPP directive-based programming model offers a powerful syntax to efficiently offload computations on hardware accelerators and keep the original C or Fortran codes. Figure 2 show the HMPP Workbench framework.

The basic paired directives syntax of HMPP are as follow:

```
...
//HMPP codelet
#pragma hmpp label1 codelet, target=CUDA
void test(int n, int A[n], int B[n]){
    for(int i = 0; i < n; i++){
        B[i] = A[i] * A[i];
        }
}
...
int main(int argc, char ** argv)
{
...
//HMPP callsite
#pragma hmpp test callsite
test(99, A, B);
...
}
```

Codelet is used before the definition of C functions which could be transformed in to parallel kernels. Callsite is the HMPP directive for invocation on such an accelerator device.

4. STEREO MATCHING ALGORITHMS

Stereo vision aim to reconstruct a disparity map (depth information) from two views. For real-time stereo systems both speed and accuracy are crucial criterions. In order to satisfy the demand of real time, it must be with the benefit of hardware acceleration, especially data parallel architectures-GPU. After analyzing the sequential C code of stereo vision, the most stereo matching algorithms spent a large percent of workload data parallel computing around every pixels. So the stereo matching is suitable for CUDA or HMPP accelerating. Here we adopt the ESAW (exponential step size adaptive weight)[9] and ESMP (exponential step size message propagation)[10]stereo algorithm, because the author offer the source code of CUDA to download and we can compare the performance of the manual CUDA and CUDA generated by HMPP directly.

4.1 Default HMPP Transformation

For the version of HMPP CUDA, we only insert these two lines directives into the source C code without any modification like that:

```
...
#pragma hmpp stereo codelet , target=CUDA,
extern void stereo()
{...}
...
int main(int argc , char ** argv)
{
...
#pragma hmpp stereo callsite
stereo();
...
}
```

Only two lines HMPP directives codelet and callsite can replace writing complicated manual CUDA code. And the difference of performance between the manual CUDA and HMPP CUDA will be described in the section 5.

4.2 Optimized HMPP Transformation

As we illustrated in the Section2, we should use the shared memory and even L1 cache to optimize the HMPP code. In our Optimized version of HMPP code:

1. we use the hmppcg directives to determine which loop should be parallelized.

   ```
   //Exemple: HMPP directives:unroll forloop
   #pragma hmppcg gridify(j,i)
   for (j = 0; j < HEIGHT; j++){
           for (i = 0; i < WIDTH; i++){
       ...
       }
   }
   ```

2. we use the hmppcg grid to allcate the buffer to shared memory.

   ```
   #pragma hmppcg grid shared buffer_name
   ```

3. Through the configuration of L1 cache preference, there is obvious improvement of performance as described in the section 5.

   ```
   cudaDeviceSetCacheConfig(args);
   args = cudaFuncCachePreferL1;
   ```

5. EXPERIMENTAL RESULTS

Table 1: Consumed time (ms)

	C code	Manual CUDA	HMPP CUDA
ESAW	3520	190	290
ESMP	6230	220	320

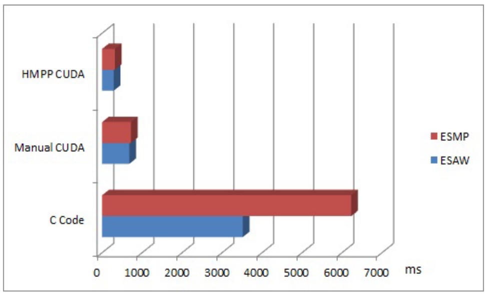

Figure 2: Performance of ESAW and ESMP.

5.1 Experiment Environment

Our experiment environment is configured as follow:

- CPU: Inter Xeon W3540 (8 cores, 2.93GHz)
- GPU: Nvidia Quadro FX 3700 (128 cuda cores, 1 GB memory)
- OS: ubuntu 12.04 LTS
- CUDA: v4.2.9
- HMPP-Workbench: v3.2.1

5.2 Consumed Time Measure

We test the ESAW and ESMP algorithms in such three implementations:

- C code
- Manual written CUDA code
- HMPP CUDA code

And we set the the iteration times = 9 in both ESAW and ESMP algorithms, keep the others parameters default. From the experimental results as described in the Table 1 and Figure 2, we can see that both manual CUDA and HMPP CUDA have the obvious speed-up compared with C code implementation of ESAW and ESMP algorithms. The manual CUDA obtains 18.5x to 28.3x speed-up, corresponding with HMPP CUDA's 12.1x to 19.5x speed-up. After analyzing the difference performance between the manual CUDA and HMPP CUDA, there are two main factors:

- Different transferring between GPU and host
- Different occupancy of GPU

By the Nvidia CUDA profiler, both the time consumed of manual CUDA implementation and HMPP CUDA implementation are classified into three groups: 1. H2D (transferring data from host to device like GPU), 2. kernel (functions executed on GPU to realize the algorithms), 3. D2H (transferring data from device to host). As described in the Figure 3, we can see that the manual CUDA cost two much time on D2H(almost 50% time consumed total) which even close to the kernel consumed time.

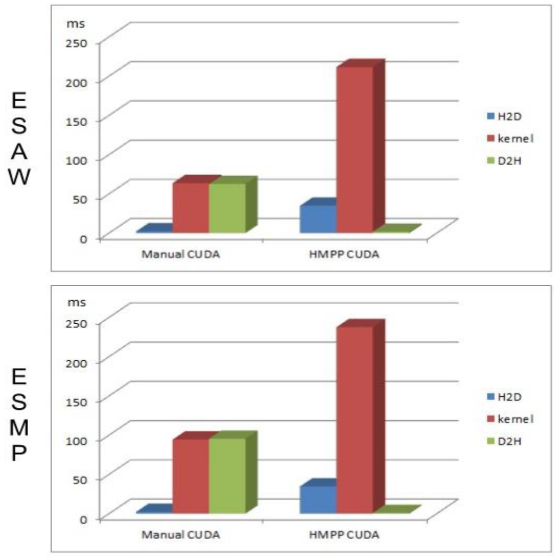

Figure 3: Performance Details (H2D (transferring data from host to device like GPU), kernel (functions executed on GPU to realize the algorithms), D2H (transferring data from device to host))

5.3 Occupancy

As discussed in the Section 2, occupancy is the key factor to determine whether these kernels can fully saturate the computing device. According to the Nvidia profiler, the manual CUDA didn't make use of the GPU computing unit very well as shown in the Figure 4 and 5. Most of time, the manual CUDA only take 33% or 66% occupancy of our GPU device. And the HMPP CUDA take 66% occupancy in the most time which is better than the manual CUDA. And the same kernel maybe has the different occupancy on different device. This is the problems of determining the best code transformation at a high level compiler like HMPP.

5.4 Optimized HMPP Transformation

Because the configuration of L1 cache preference need the compute capability 2.x, but our GPU-Quadro FX 3700 only has compute capability 1.1. So we change to another GPU platform GeForce GTX 550 Ti for the esperiment of Optimized HMPP. For the ESAW algorithm, we get the result of 60ms for manual CUDA and 80ms for HMPP CUDA. It mean that the HMPP CUDA can obtain the close performance as manual CUDA even have the chance to exceed it. The millions of disparity per second(MDS) is another criterion of real-time stereo vision system. The optimized HMPP CUDA can achieve 79.5 MDS which meets the demand of real-time stereo vision.

5.5 Comparison of Disparity Map

We use the two series image: Cones and Teddy for the test as shown in the Figure 6, and executed using GPU, we get the final visual disparity map of the manual CUDA and HMPP CUDA separately as shown in the Figure 7. There are little difference between the two version Disparity Map and the error rate of Disparity Map are as follow:(waiting the result)

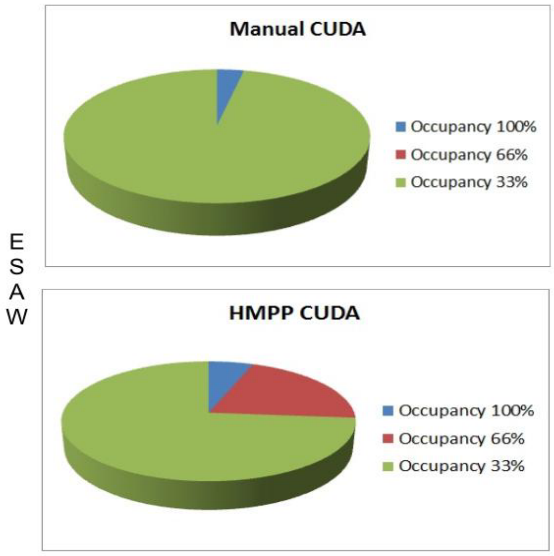

Figure 4: Occupancy details of manual and HMPP CUDA for ESAW.

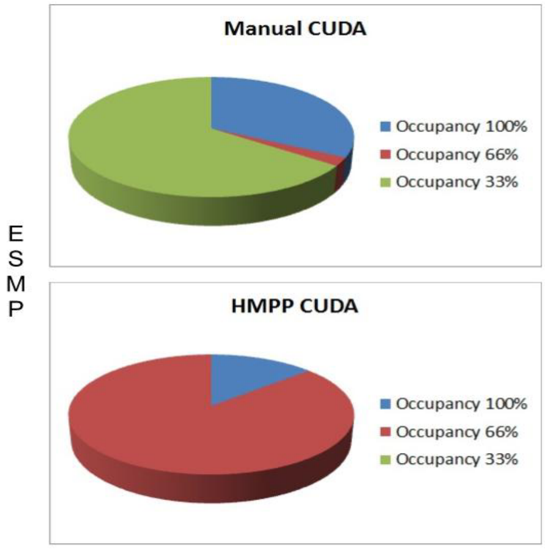

Figure 5: Occupancy details of manual and HMPP CUDA for ESMP.

6. CONCLUSION

In this work, we use the directive-based HMPP Workbench to generate the CUDA kernel code and optimized the HMPP directive for bettter performance. With HMPP, we were able to quickly generate different versions of portable code on different device NVIDIA GPU (CUDA target), AMD GPU (OpenCL), Intel MIC(coming soon). The experimental results show that the default and optimized HMPP have the approximative performance compared with CUDA implementation. And the HMPP workbench can greatly reduce the time of application development using parallel computing device.

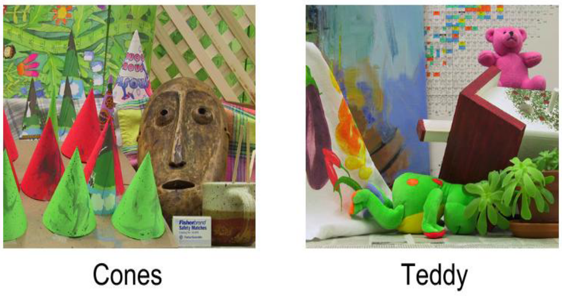

Figure 6: Source image of Cones and Teddy.

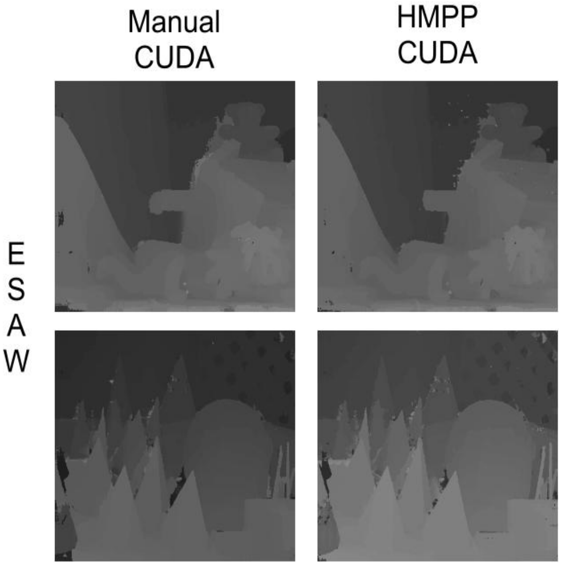

Figure 7: ESAW.

7. REFERENCES

[1] CAPS. Hmpp. http://www.caps-entreprise.com/technology/hmpp/.

[2] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. 2012.

[3] Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. Stereo matching using belief propagation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(7):787 – 800, july 2003.

[4] Jinglin ZHANG, Jean-Francois NEZAN, and Jean-Gabriel COUSIN. Parallel implementation of motion estimation based on heterogeneous parallel computing with opencl. In *High Performance Computing And Communications (HPCC), 2012 IEEE 14th International Conference on*, 2012.

[5] Altera. Opencl. http://www.altera.com/b/opencl.html.

[6] Jianbin Fang, A.L. Varbanescu, and H. Sips. An auto-tuning solution to data streams clustering in opencl. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pages 587 –594, aug. 2011.

[7] Nvidia. Tegra3. http://www.nvidia.com/object/tegra.html.

[8] Qualcomm. snapdragon. http://www.qualcomm.com.au/products/snapdragon.

[9] Wei Yu, Tsuhan Chen, and J.C. Hoe. Real time stereo vision using exponential step cost aggregation on gpu. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 4281 –4284, nov. 2009.

[10] Wei Yu, Tsuhan Chen, F. Franchetti, and J.C. Hoe. High performance stereo vision designed for massively data parallel platforms. *Circuits and Systems for Video Technology, IEEE Transactions on*, 20(11):1509 –1519, nov. 2010.