Parallelizing the Precomputed Scan Matching Method for Graphics Card Processing

Petr Schreiber, Vít Ondroušek, Stanislav Věchet, Jiří Krejsa
Brno University of Technology

Brno, Czech Republic
petrschreiber@gmail.com, ondrousek@fme.vutbr.cz, vechet.s@fme.vutbr.cz, krejsa@fme.vutbr.cz

Abstract— **Certain methods solving mobile robot localization problem are reliable, but computationally expensive. One possible way how to increase the speed of necessary calculations is to use parallel approach and use graphics card to speed the processes up. Methods such as Precomputed Scan Matching Method (PCSM) or Particle Filters are suitable for parallel processing. PCSM method requires processing the map prior to localization and so far such processing had to be done offline, while the new approach brings computational time reduction in order of a magnitude. The paper describes the modifications of PCSM method and its implementation suitable for modern ATi Radeon and NVIDIA GeForce graphics card series.**

Keywords- Localization; PCSM; GPGPU; OpenCL

I. INTRODUCTION

Determination the position of mobile robot is essential issue in both indoor and outdoor robotics. This issue, called localization, can be divided into two main groups: position tracking and global localization. Position tracking uses information about the motion of the robot (usually from IRC sensors) together with some kind of outer sensor to keep track of the robot position changes. Position tracking therefore requires the initial position of the robot to be known. Outer sensors depend on the environment, compass and GPS receiver are usually used in outdoor, beacons and some sort of feature extractor are usually used indoor. Some sort of probabilistic filter (Extended Kalman filter, Particle filter, etc.) is commonly used as the engine that estimates the robot position as the fusion of motion model and measurement from the sensors.

Global localization must provide the information about the position of the robot with little or no data regarding the initial estimate and it should cope well with multimodal distributions of the estimate (several hypothesis with high probability are possible). Global localization is usually computationally more expensive and when lower power computational means are used, high demands usually limit the speed of the robot or expand the time span of localization steps to higher values, or some of the computation must be performed offline, if possible. The Precomputed Scan Matching (PCSM) method described below is a representative of such a method.

In order to use computationally more demanding methods in commonly available hardware, that is available for students and university robot building teams the methods must be either optimally coded or if possible certain parallelization must be performed. The paper gives an overview of PCSM method computed using graphics card processing enabling to speed up necessary routines and run the method on commonly available hardware. The paper is organized as follows: first the method itself is described, possible ways of performance enhancement are mentioned, the OpenCL technology is overviewed and details regarding the implementation and results are finally given.

II. PRECOMPUTED SCAN MATCHING

Precomputed Scan Matching (PCSM) is method for indoor robot localization, originally developed by Stanislav Věchet on Brno University of Technology[1]. The method is designed for mobile robots equipped with some kind of proximity sensor, usually the laser rangefinder, with the scan range of 180° or more (see Fig 1.).

The execution of PCSM is split into two stages: the precomputing and the localization stage. The first stage performs virtual scan of the map of environment, by shooting 360° rays from points where robot is designed to operate. These rays collide with environment model, and the output data set represents cached information which can be easily accessed in localization stage, which is based on comparison of real scan and precomputed scans. The pair of two scans closest to each other indicates the expected position of the robot. Details about the method itself can be found in [1].

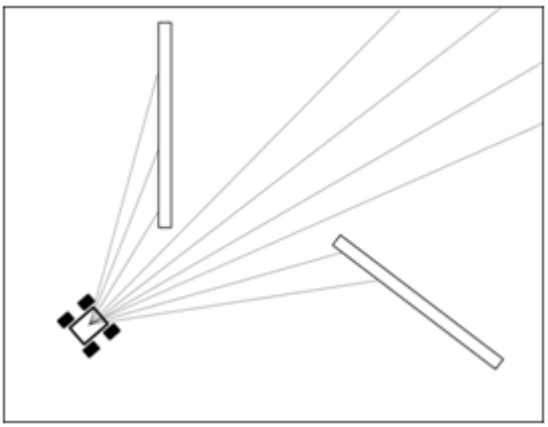

Fig. 1. Distances measured in real environment

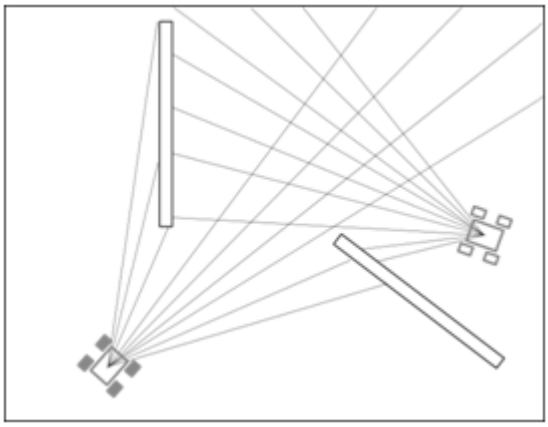

Fig. 2. Building the set of precomputed scans from numbers of virtual positions in environment map

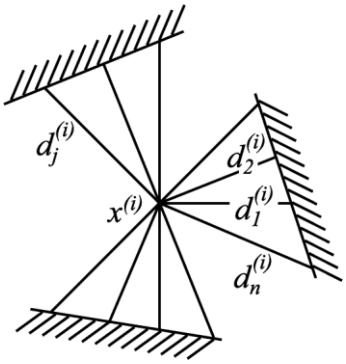

Fig. 3. Neighborhood scan definition

The comparison of real and precomputed scan is essential for PCSM method performance. Comparing function is called Match and it defines the difference of two scans by function

$$M(x) = r(x, a, S)$$

where *x* is the state of the robot, *a* is a real scan acquired by sensors and *S* is the set of precomputed scans. The Match is calculated based on complete neighborhood scan *d* defined as a set of single distances,

$$d = \{d_j\}_{j=1,\dots,n}$$

where *dj* is the distance of single beam.

The set *S* is a number of m precomputed scans which are stored as

$$S = \{x^{(i)}, d^{(i)}\}_{i=1,\dots,m}$$

d(i) is precomputed scan obtained from virtual position *x(i)*

Final match is calculated for all precomputed scans stored in set *S*. As a result, the one with the highest match (represented by the minimum of match function) is returned. The match function is defined as follows

$$r(x^{(i)}, a, S) = \sum_{j=0}^{n} \left(d_j^{(i)} - a_i\right)^2$$

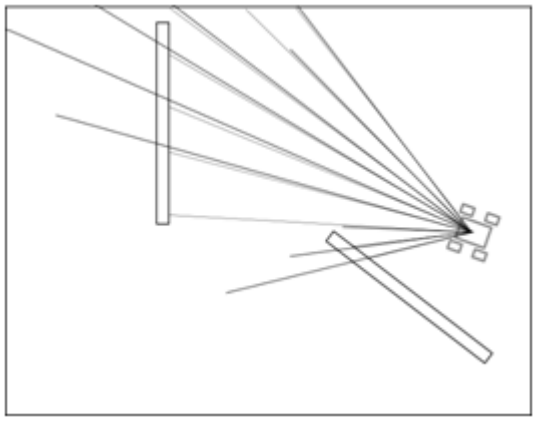

Fig. 4. Mismatch of the real and precomputed scan

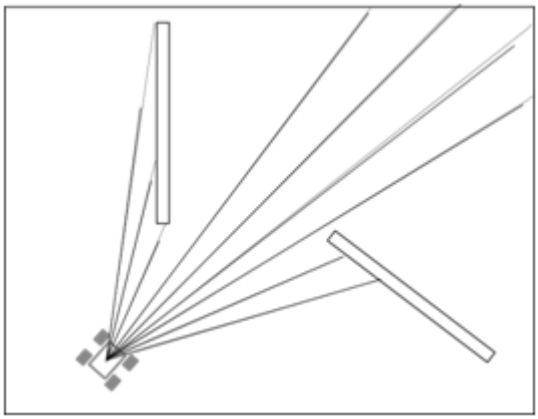

Fig. 5. Sufficient match of two scans

Two different examples of possible matches are shown on Fig. 4 and Fig. 5, illustrating mismatch of the scan (robot is in the position not corresponding with measured data) and successful match (robot is located in position corresponding with the measurement)..

As clear from the description above, the localization itself is based on comparing the real sensor data with precomputed data, and performs very well even on slower CPUs[2]. Originally the method was designed to have the precomputation done offline, because it is very computationally intensive, and can take lot of time to perform for larger environments with high precision. The precision is determined by how close the precomputation points are placed. The goal for further optimization was to make this stage fast enough to be performed directly on mobile robot, removing the need for using high performance workstation.

III. POSSIBLE OPTIMIZATION APPROACHES

While the problem of colliding the rays with obstacles seems to be trivial, the main problem consists in the fact line to line collision has to be performed millions of times. Optimizing this operation by breaking high level code into assembly with heavy use of SIMD instructions could be seen as evident choice, but this approach was completely avoided because it would lead to complication with optimized code

maintainability, not speaking of lack of brute force of Intel Atom chip.

The target robot is built on top of Nvidia ION platform, which besides the mentioned Atom processor provides programmer with Nvidia GeForce 9400 as well. While this GPGPU (General Purpose Graphics Processing Unit) is completely ignored by hardcore gamers for its sometimes insufficient graphics performance, the raw computional performance of its 16 stream processors still offers significant advantage over Atom.

Programming the GPGPU today has become much easier comparing to past. There are four major technologies to consider – ATi Stream[3], Nvidia CUDA[4], DirectCompute and OpenCL. The Stream and CUDA, while proven to deliver outstanding performance, are technologies locked to specific hardware, so making optimization like this could badly pay off in case we would choose to change platform in future. DirectCompute is promising technology, but limited to Windows OS.

For these reasons the OpenCL was chosen, as the most independent solution from both software and hardware standpoint.

IV. About OpenCL

OpenCL is technology currently maintained by Khronos organization, which also handles other well known standards, such as OpenGL. OpenCL stands for Open Computing Language and as a such you can use it for programming of both CPUs and GPUs. The range of supported GPGPUs is slightly wider on Nvidia side, where you can run OpenCL on anything from GeForce 8 and up, while on AMD side only the latest ATi Radeon HD 4000 or better the HD 5000 series are supported.

The GPGPUs are the device of choice for data parallel tasks, thanks to their big amount of processing cores, comparing to CPUs. OpenCL allows us to use data parallel programming, which is exactly what we needed for tweaking the PCSM. The architecture of OpenCL can be viewed in 2 parts: run-time host API and OpenCL C language with compiler. The API allows embedding this technology basically to any language, on Windows platform it is every language capable of interacting with industry standard DLL.

Programmer can write so called kernels (procedures executed in parallel) in OpenCL C which is high level language derived from C99, with few restrictions and extensions comparing to this standard. You can learn more about this language and OpenCL in general in[5].

The important point is that the code written in OpenCL is not graphics card specific at all. It is no longer necessary to use graphics specific shader programs for general purpose computations. OpenCL C is also relatively high level language, which, in author's opinion, means significant advantage in maintainability over optimizations written in assembly.

V. Limitations of GPGPU

While the OpenCL language itself is not GPU specific, when programming such a devices you have to keep few facts in mind:

- Double precision performs well only on high end models
- Excessive code branching can hit performance
- The resources usable on each GPU slightly differ
- Precompiling is not an safe option as of Q2 2010

Although the 9400GT is model which does not expose the double precision functionality, the first mentioned problem posed no danger for realization of PCSM. Changing the units of distance to millimeters reduced requirement of precision for many digits after decimal points.

The second mentioned issue is indeed observable when working with OpenCL, and is caused by fact GPU programming, even the pure graphic one, did not allowed dynamic branching for long time. Current models support this feature, but the performance hit is observable, yet not critical.

The problem of GPU running out of resources is the tricky one, and will be further discussed in paragraph VI.

The compilation of GPU code is performed by driver, and as the OpenCL devices are very diverse, there is no standard for binaries. In theory, when using the same code on the same device, it would be possible to rely on once compiled binaries. But as dramatic progress in quality of OpenCL implementations can still be observed today, the preferred way is to perform fresh compile before execution. As the kernel code is usually very brief, with no include files, the compilation is performed in times typically under 1 second, which does not represent big problem.

VI. Precomputation Design with OpenCL

As mentioned in paragraph II, the precomputation phase first find the points where robot can appear during its journey and then calculates the 360° collision in each of this point. This allows splitting the calculation into two kernels. Another reason to do this is to simplify the kernels to reduce GPU resources usage. The resources question is one of the problematic sides of GPU implementations at the time of writing this article, as they fail to report whether the compiled kernel is too resources intensive for the given device. Any auxiliary functions the kernel uses are currently handled as inline, which means the modularization of executions may suffer from excessively large kernel size after the expansion of inline calls.

A. FindPointsForCalculation Kernel

The first kernel, as the name suggests, seeks for the points where robot will operate. That means any place which is not covered by visible or invisible obstacle. The result of the process is shown on Fig. 6.

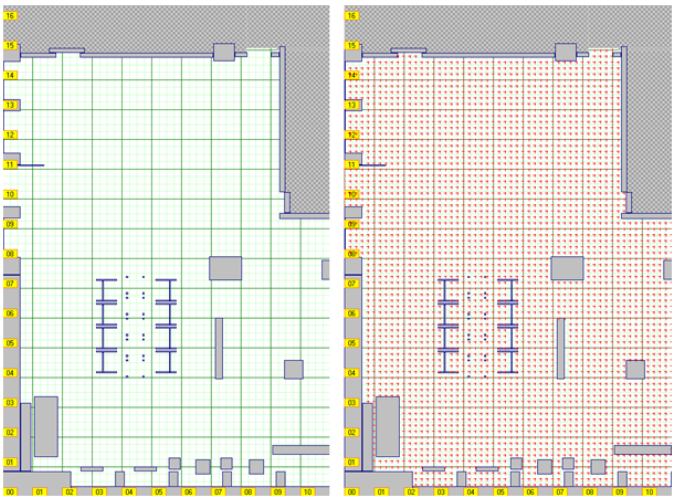

Fig. 6. Initial search for possible robot position

This task is so trivial, it would be possible to perform this on CPU with similar execution time. The reason why this is done on GPU is obvious – to eliminate later transfer over the PCI-E bus, which would become unnecessary bottleneck for our calculation.

The data passed to the kernel are array of points where we need to determine whether they are in obstacle or not, step size determining distance of points from each other on rectangular grid and then set of structures describing the obstacle transformation and visibility. The problem is specified as 2D, which allows filling the point structures with coordinates on the GPU, deducing them from kernel internal information IDs. The result of the calculation, array of points with field "usable" set to 0 or 1, is left in OpenCL buffer object on the GPU, to eliminate transfer over the bus.

B. *CollideRays Kernel*

The second kernel used performs the collision in given points. To modularize the code, two additional subroutines are used by the kernel: *aux_crossDistance* and *aux_rayObstacleDistance*. The first subroutine calculates the intersection between two lines. The second one decomposes the obstacles to set of lines and calls the first one to retrieve the intersection with whole object.

The arguments are array of validated points, information on obstacles and empty array to retrieve the precomputed rays. The design of OpenCL kernels is very flexible, allowing reusing the data already present on the GPU. Thanks to this the validated points are results of the FindPointsForCalculation kernel. The same could be done with the obstacle information, but here the expected problem appeared – excessive resource usage of kernels. Thanks to the nested function calls and dynamic branching, the kernel did not manage to process all the obstacles at once, resulting in *cl_out_of_resources* error.

This was solved by multipass execution of *CollideRays* kernels. In classic programming it would mean to pass all parameters over and over, with OpenCL we can update only the parameters which change. In this case, the only updated parameter will be smaller batch of objects to collide with. The rest of data stays on GPU and is continuously updated during the run of the passes.

The kernel code can be observed on Fig. 7, clearly demonstrating ease of use of OpenCL C.

```
__kernel void
CollideRays(__global const PointValidated2D* pointV,
            __global const Box2D* boxes,
            __global const int* boxCount,
            __global       PrecomputedPoint2D* pointC
           )
{
  int n = get_global_id(0);

  if (pointV[n].validated == 1) return;

  float2 origin;

  origin.x = pointV[n].x;
  origin.y = pointV[n].y;

  float i;
  float angle;
  float dist;

  #pragma unroll
  for(i = 0; i <= 359; i++ )
  {
    angle = radians(i);

    pointC[n].x        = origin.x;
    pointC[n].y        = origin.y;
    pointC[n].rayCount = 360;
    dist = aux_rayRectangleDistance(origin,
                                    angle,
                                  80000.0,
                                    boxes,
                                 *boxCount);

    if (pointC[n].pRay[convert_int(i)] == 0)
    {
      pointC[n].pRay[convert_int(i)] = dist;
    }
    else
    {
      if (dist <= pointC[n].pRay[convert_int(i)])
      {
       pointC[n].pRay[convert_int(i)] = dist;
      }
    }
  }
}
```

Fig. 7. Kernel code in OpenCL C

The ideal batch size had to be evaluated experimentally. Generally it was observed the amount of 200 obstacles per pass is acceptable, but it is recommended to fine tune this value on your target platform manually.

On most Nvidia cards the batch size seemed to have some kind of direct relationship with maximum parameter size, which is information reported by OpenCL run time API. But this was pure coincidence and this method cannot be used for batch size evaluation for real cases. In case of ATi hardware there was no such an observable coincidence.

During the adaptation of the method for the mobile robot it was observed that bigger batches of obstacles used (in amount not causing the mentioned resource problem) the better the performance was.

This indicates the transfer of data over the PCI-E bus still makes the difference, even in case of updating single, relatively small parameter.

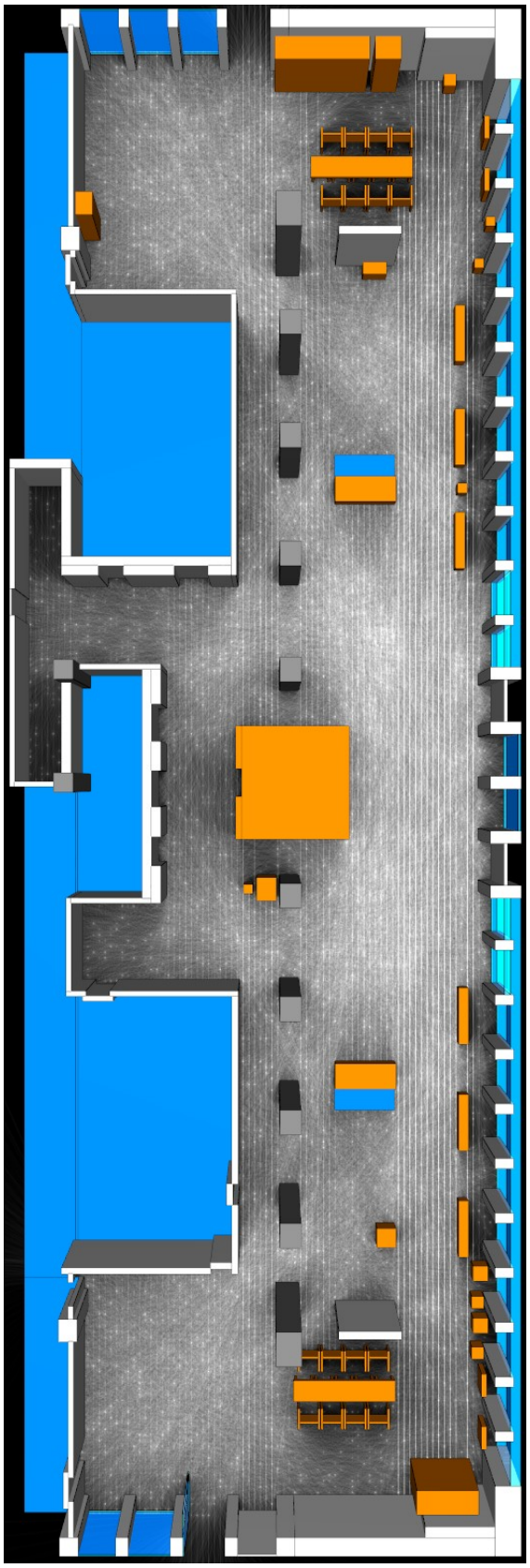

Fig. 8. Visualization of the precomputed rays

VII. Results of The Optimization

To compare the performance gain, we executed the original C# implementation, OpenCL CPU and OpenCL GPU implementation on AMD Sempron processor, which we observed to be approximately twice as fast as Intel Atom present in our mobile robot. It was very interesting to see the demonstrated OpenCL version on CPU with single core Sempron performs worse than our C# implementation, as seen on Fig. 9. To get the picture why we needed to optimize the precomputation, you can see the calculation time for 120 millimeters precision. It takes almost half an hour for Sempron CPU, which is still significantly faster comparing to CPU platform the robot was equipped with.

As much as the OpenCL CPU performance showed to be insuffucient, the OpenCL GPU implementation demonstrated desired speed boost, which shortened the execution time from tens of minutes to tens of seconds, being viable option for practical use.

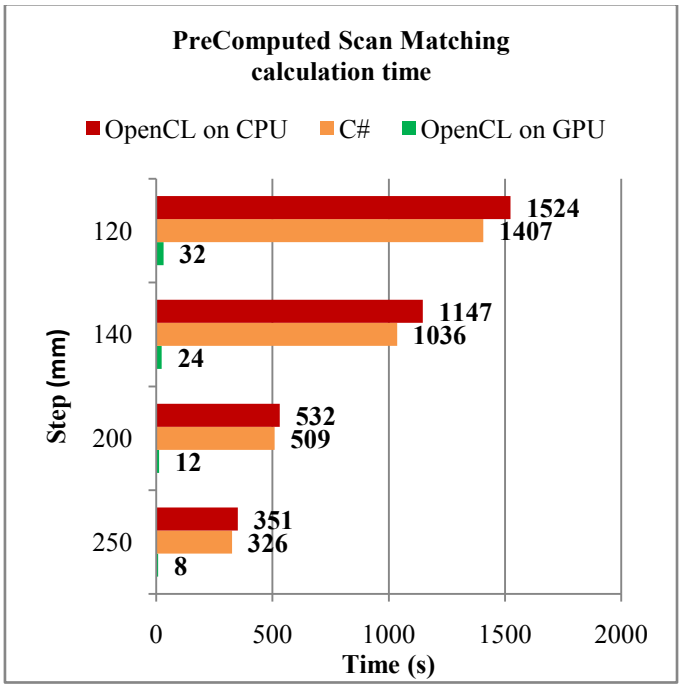

Fig. 9. Timing of the computation on GPU and two CPU implementations

The problem scales well on the GPGPU hardware. During the testing authors observed direct relationship between growing numbers of stream processors and shortening execution time. Even the low end models of graphic cards can provide highly competitive performance for data parallel calculations.

From theoretical point of view, using stronger graphic card could lead to unwanted power consumption problems. In reality this might not be an issue. With robot based on platform with high performance dedicated mobile GPGPU, such as *Nvidia GeForce GTX275M*, we can observe the execution time shortens to just few seconds even for high detail of

precomputation. This results in very short peak in power consumption, which finally does not pose big problem for practical use of the technology.

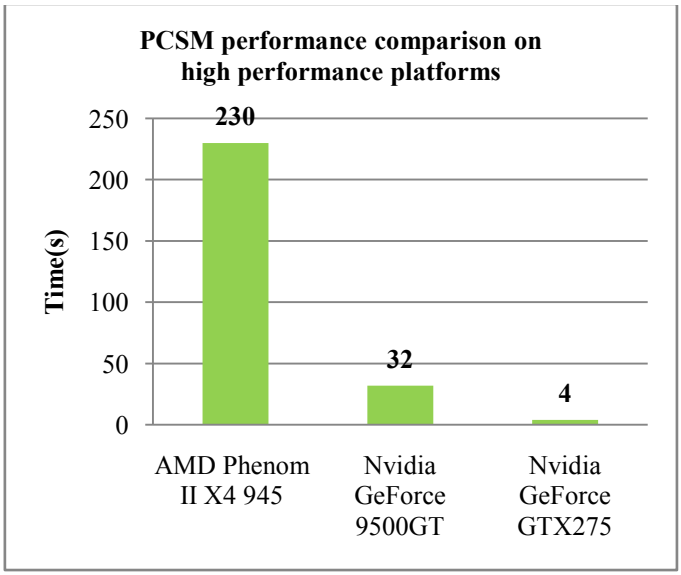

Fig. 10. Timing of the precomputation on high end CPU and two GPUs for 120 mm precision

The same observation cannot be confirmed for the case of using stronger CPU instead. The tested desktop high end 3GHz quad core processor from AMD proven to be still over 7 times slower than the low end GPGPU model implementation, while the power consumption, thanks to fully using power of all 4 cores, raised dramatically. When comparing the CPU performance to the high end GPU, the gap becomes even more significant as shows Fig. 10.

VIII. Conclusion

In the end, the presented GPGPU solution delivers significant acceleration of operation which was traditionally handled as offline task, while using conventional processor based approach. The optimization of the calculation has been realized thanks to using the latest OpenCL technology after very short adaptation time. The authors realize the solution could be further tweaked for even better performance.

The authors believe this successful application of GPGPU programming will encourage students and educators to focus more on the benefits of GPGPU computing for data parallel applications. The ease of use and high level syntax based programming make OpenCL an interesting technology worth the time of study. During the experiments OpenCL was used directly from multiple languages, including interpreted ThinBASIC[6]. This fact confirms that the students can use the technology from within language of their choice.

Solutions realized on OpenCL platform can be targeted to products of both major graphics card vendors, without writing any platform specific code. This makes OpenCL technology easily usable in the university environment, which very often provides highly heterogeneous hardware resources.

Building computationally intensive routines on top of GPGPU brings some significant cost savings when designing the mobile robot platforms as well. The performed tests demonstrated that even cheap GPU solutions can outperform costly modern CPUs of today at the fraction of the price.

IX. Acknowledgement

Published results were acquired with the support of the Ministry of Education, Youth and Sports of the Czech Republic, research plan MSM 0021630518 “Simulation modeling of mechatronic systems”.
While the system was designed for Bender 2[7] mobile robot platform, authors would also like to thank to Ing. Pavel Houška, PhD, Jakub Vodrážka, Jakub Drábek, Petr Liška, Jiří Michalčík, Charles Pegge and Kent Sarikaya for making the extensive testing possible on alternative hardware platforms.

[1] S. Věchet, J. Krejsa, “Potential-Based Scan Matching in Mobile Robot Localization”, in proceedings of Modelling and Optimization of Physical Systems. Gliwice, pp. 185-188, 2007

[2] Krejsa J., Věchet S., Hrbáček J., Schreiber P.: High Level Software Architecture for Autonomous Mobile Robot. In Recent Advances in Mechatronics. Berlin, Springer. 2009. p. 185 - 190. ISBN 978-3-642-05021-3.

[3] “ATI Stream Software Development Kit (SDK) v2.1” *Welcome to AMD Developer Central*. Advanced Micro Devices, Inc. n.d. Web. 30 May 2010. <http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx>

[4] “CUDA Programming Guide for CUDA Toolkit 3.0” *NVIDIA GPU Computing Developer Home Page*. Nvidia. n.d. Web. 30 May 2010. <http://developer.nvidia.com/object/gpucomputing.html>

[5] Khronos Group, “The OpenCL Specification”, Web. 30 May 2010. <http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf, 2009>

[6] E. Olmi, R. Biancchi, ”Basic Programming Language :: thinBasic”. n.d. Web. 30 May 2010. <http://www.thinbasic.com/>

[7] J. Hrbáček, T. Ripel, J, Krejsa, “Ackermann mobile robot chassis with independent rear wheel drives”, EPE-PEMC 2010, Ohrid, Macedonia, in print.