ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **18**, Number 2, 2015, 182–196

Utilization of GPU Acceleration in Le Bail Fitting Method

I. ŠIMEČEK, O. MAŘÍK, M. JELÍNEK

Department of Computer Systems, Faculty of Information Technology,
Czech Technical University in Prague
Zikova 1903/4, 166 36 Praha 6, Czech Republic
xsimecek@fit.cvut.cz

Abstract. Le Bail fitting method is a process used in applied crystallography. It can be employed in several phases of crystal structure determination and as it is only one step in a more complex process, it needs to be as fast as possible. This article begins with a short explanation of crystallography terms needed to understand the Le Bail fitting, then continues with the description of the Le Bail fitting method itself and basic principles on which it is based. Then the parallelization method is explained, starting with a more general process, followed by specifics of GPU accelerated computing including short part on optimization. Finally, achieved results are presented along with comparison to sequential implementation and alternative parallelization approaches.

Key-words: Le Bail fitting method; GPGPU; parallel execution; applied crystallography.

1. Introduction and preliminary results

The first glance at Le Bail fitting as a part of crystal structure determination doesn't offer much insight into the problem this paper[1] attempts to address. Before we delve deeper into use of massively parallel platforms in improving performance

[1] A very preliminary version of this article was published in paper Mařík, Šimeček: *Acceleration of Le Bail fitting method on parallel platforms*, http://panm17.math.cas.cz/PANM17_proceedings.pdf pages 136-141. We have revised and extended (from 6 to 19 pages) the whole text of this paper. We included a more in-depth analysis of the parallelism in the method. The original paper was strictly focused on OpenMP technology, this article is also includes implementation and optimization for CUDA API.

of this process, some basic explanation of Le Bail fitting method and related areas of crystallography are required. We will however try to keep those to a bare minimum, since the crystallography is not the main focus of this work. It starts with the more general principles that are more widely known, followed by crystal structure determination from powder diffraction and concludes with the Le Bail method itself.

1.1. Crystalline matter

The main subject of crystal structure determination are of course solid matters, which take on the crystalline form when changing from liquid to solid. The other form of solid matter is amorphous. The most significant difference between these two is the configuration of atoms in the solid matter, which is ordered or even periodic in the case of crystalline form. This fact gives it certain properties such as ability to represent each material with its unit cell, the basic part of crystal lattice. It also enables us to use diffraction in the study of these materials.

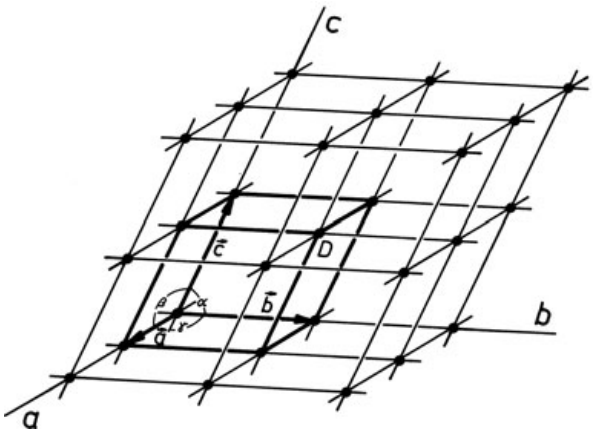

Fig. 1. Space lattice with the unit cell (reprinted from [2]).

As mentioned previously, the atoms in crystalline matter are ordered in such way that their pattern repeats periodically. The concepts of crystal lattice and unit cell are depicted in Fig. 1. Each crystalline material can be defined by some unit cell, more specifically by its parameters also apparent from Fig. 1. These are lengths of the cell edges, usually labeled a, b, c and angles between the edges, α, β, γ. The combination of both can also be used, in the form of vectors $\vec{a}$, $\vec{b}$, $\vec{c}$.

Based on the crystal structure parameters (unit cell properties a, b, c, α, β, γ), the crystal lattices can be divided into several categories. These are called crystal systems, for example the most simple cubic system has all three edges of the same length and all angles equal 90°. Complete list and properties of the usual seven crystal systems can be found in [2] or [3]. The most important consequence of the symmetry found in certain crystal system is the simplification of the parameters determination.

For the purposes of diffraction explained in the following sections, there are also

defined geometrical constructs in the form of crystallographic planes (consisting of lines and subsequently points). Each crystallographic plane can be specified by three points (not on the same line) and therefore usually represented by the integral multiples of each of the three basic vectors to specify points in which the axes specified by these vectors intersect the plane. Furthermore, parallel planes are considered effectively as one in the context of crystallography, in distance d given by the periodicity of the lattice. These planes are labeled by the Miller indices, which are denoted as (hkl) triplet. These can are the smallest integer multiplies of the reciprocals of the plane intercepts on the axes [2] or the number of equal parts into which the plane set divide each basic vector.

1.2. Powder diffraction

The term crystal structure determination from powder diffraction data is almost self-explanatory for people familiar with crystallography. That is something that cannot be assumed here however and therefore this section briefly explains the basic principles and background into which Le Bail fitting belongs.

First important knowledge that relates to previous section is that crystalline matter itself can be in slightly different state. There are basically two recognized states, single crystal (or monocrystal) and polycrystal. The monocrystal is basically the ideal state, in which the matter forms the ordered structure explained in previous section. The structure is the same in the whole volume of the matter and results in regular shape even of the crystal in macroscopic scale. The second type, more commonly found in nature, is polycrystal and basically consists of a great number of randomly oriented tiny monocrystals (called crystallites). While the overall symmetry is reduced, this state offers isotropic properties, which is one of the basis for powder diffraction, since powder is basically a polycrystalline matter.

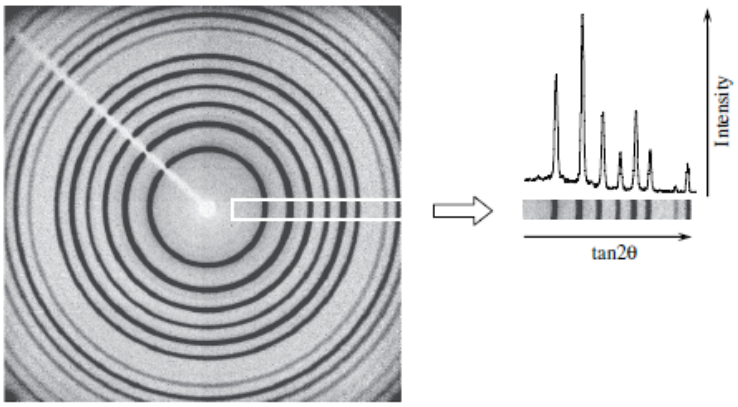

Fig. 2. X-ray diffraction pattern of polycrystalline LaB_6 (left) and corresponding intensity as a function of $\tan 2\theta$ (reprinted from [3]).

The second part of powder diffraction is the X-ray diffraction phenomenon. Similar to its optics equivalent using visible light, the use of X-rays enables the diffraction to occur when the distances of atoms in crystal lattice are involved. Thus when X-ray radiation passes through crystal, be it monocrystal or polycrystal, it forms certain patterns from which the internal structure of the used sample can be determined. The results while using monocrystal are of course different than those using polycrystal (powder sample). The polycrystals are generally more affordable, but the patterns obtained contain less information about their structure, which then has to be extrapolated. Le Bail fitting is used during this process. The data obtained from powder diffractometry are usually in the form of diffraction pattern (profile) as illustrated by Fig. 2.

The right side of Fig. 2 shows the usual form of powder diffraction pattern record. The most important parameters used for crystal structure determination are the peak positions (for unit cell parameters), peak intensities and shape. The relationship between unit cell parameters and peak positions can be expressed based on Bragg's law by equations:

$$2\Theta_{hkl} = 2\arcsin\frac{\lambda}{2d_{hkl}}, \text{ where } d_{hkl} = \sqrt{\frac{1}{Q}} \tag{1}$$

$$\begin{aligned} Q = \Bigg[& \frac{h^2}{a^2}\sin^2\alpha + \frac{2kl}{bc}(\cos\beta\cos\gamma - \cos\alpha) + \\ & \frac{k^2}{b^2}\sin^2\beta + \frac{2hl}{ac}(\cos\alpha\cos\gamma - \cos\beta) + \\ & \frac{l^2}{c^2}\sin^2\gamma + \frac{2hk}{ab}(\cos\alpha\cos\beta - \cos\gamma)\Bigg] / \\ & (1 - \cos^2\alpha - \cos^2\beta - \cos^2\gamma + 2\cos\alpha\cos\beta\cos\gamma) \end{aligned} \tag{2}$$

The interplanar distance d_{hkl} is used in structure determination as an intermediary result, but it has meaning of its own (distance between two closest parallel crystal planes), $2\Theta_{hkl}$ is the peak position and Q is used just for simplification. The equation 2 holds for all crystal systems, but it can be simplified for other systems than triclinic.

1.3. Crystal structure determination from powder diffraction data

Getting the powder diffraction pattern is arguably the easier part of crystal structure determination. The process of obtaining structure parameters from it can be rather complicated and varied, both in methods used and the precision of obtained results. However, the process usually consists of the following steps or a variation thereof.

1. *Collection of powder data* is the process of obtaining the diffraction image from prepared sample.
2. *Preliminary processing* then determines the positions and intensities of Bragg's peaks in data.

3. *Indexing* attempts to find the unit cell parameters, which best fit the observed pattern. It is performed either without any prior knowledge of the parameters or with approximate knowledge of parameters. However, both cases are usually computationally demanding.

4. *Space group symmetry selection* basically chooses the crystal system of sample based on information obtained in previous steps.

5. *Whole pattern decomposition* returns to the diffraction pattern to extract precise integrated intensities of observed peaks.

6. *Structure solution* then attempts to create a model of the crystal structure based on most of the previously obtained data.

7. *Initial model refinement* then completes the model and assesses the validity of obtained model, potentially restarting the process from symmetry selection.

8. *Rietveld refinement* or suitable alternative is the final step in structure determination and focuses on refining the crystal structure parameters by comparing experimentally obtained diffraction profile with one calculated from said parameters.

Le Bail refinement is usually employed in the final stages of *Indexing* to confirm the parameters, but can also serve as a part of final structure refinement. It can be performed on more sets of structure parameters estimates, which can be obtained from one studied sample by different methods or instrument and used to select the most accurate for further study.

1.4. Le Bail refinement

Now that the use of Le Bail refinement was explained, let's focus on the method itself. As mentioned previously, its purpose is to refine the parameters based on diffraction profile. The process can be summarized into the following steps, from which the second and third steps are done repeatedly.

- Firstly, the list of all possible observable reflections (represented by Miller indices hkl) has to be generated based on the crystal symmetry found in the crystal.
- Several iterations of Le Bail whole profile fitting (decomposition) are performed to obtain integrated intensities of peaks (reflections) from observed diffraction pattern.
- Least squares method is employed to adjust the structure parameters with the use of pattern reconstituted from peaks obtained in previous step.

The iteration of the second and third steps is stopped when the refinement has no positive effect on the quality of structure parameters given by their figures of merit R_p, M_{20} and F_N. These are usually based on difference between calculated and observed diffraction profile, more information can be found in [3, pp. 421-426,521-523].

The iterated steps of Le Bail refinement deserve more detailed explanation. The first step, Le Bail whole profile fitting, can be also called decomposition since its main purpose is to decompose the diffraction profile into separate peaks. More precisely, it has to determine their integrated intensities. Figure 3 illustrates the underlying principle and also shows the problem it has to solve – overlapping peaks.

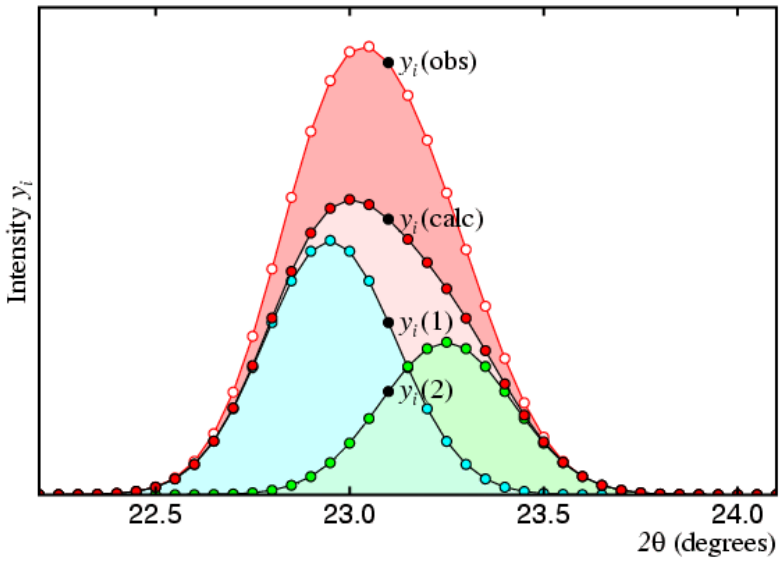

Fig. 3. The principle of diffraction pattern decomposition (adapted from [4]).

This principle can be applied to multiple overlapping peaks and on the whole pattern hence the name "whole profile fitting". Figure 3 also hints at the solution method. An approximation of the integrated intensity of a peak can be calculated using the following formula:

$$I_K(obs) = \sum_i \left(w_{i,K} \cdot S_K^2(obs) \cdot \frac{y_i(obs)}{y_i(calc)} \right), \tag{3}$$

where $I_K(obs)$ is the integrated intensity of Bragg's peak at position, Θ_K, $w_{i,K}$ is weight of the contribution of the peak to processed diffraction pattern point, $S_K^2(obs)$ is the integrated intensity from previous iteration (set to the same arbitrary value in the first iteration) and $y_i(obs)$, $y_i(calc)$ are the magnitudes of diffraction profile, observed is obtained from the diffraction experiment and calculated is based on current structure parameters estimates. The repeated application of the Equation 3 causes the integrated intensities to be refined and therefore the calculated pattern to come closer to the observed one.

The second step of Le Bail fitting is the least squares method. As the least squares is relatively extensive subject, its thorough explanation isn't included in this paper. The basic use of the least squares is to find best solution for an overdetermined system

of equations. In practice it can be used to minimize error in variable calculated from experiment results. In the case of Le Bail fitting, its main goal is to adjust the structure parameters in such way that the profile calculated from them better fits the profile obtained in the experiment. This can be done because each point in diffraction profile is basically a function of these structure parameters.

Since the function is not linear, the non-linear version of least squares has to be used. The most basic equation that can be used for solution of this problem follows:

$$\mathbf{\Delta x} = (\mathbf{A^T A})^{-1}\mathbf{A^T y}, \tag{4}$$

where $\mathbf{x}$ is a column vector containing adjustment that need be applied to structure parameters, $\mathbf{A}$ is a Jacobi matrix (of partial derivations) with one column per structure parameter considered and number of rows based on diffraction profile discretization and $\mathbf{y}$ is a column vector of differences between discretized calculated and observed profile points.

2. Algorithm design and implementation

Based on previous section, the sequential algorithm for Le Bail refinement procedure is relatively straightforward. Since it has to perform Le Bail refinement, it needs to be able to do multiple passes of Le Bail fitting followed by least squares application. Therefore it can be divided into two main parts realized by functions DoLeBail and DoLeastSquares.

The function realizing Le Bail fitting has to alter the integrated intensities of reflections in such a way that the calculated pattern better fit the observed one. In each cycle it has to plot the calculated pattern from the current reflections, then modify each reflection using calculated and observed profiles in range of the reflection based on its width. It does so utilizing following equation, which is based on Equation3.

$$\Delta newI = \frac{y_i(obs)}{y_i(calc)} \cdot I \cdot \exp\left(\frac{-(2\Theta_i - 2\Theta)^2}{peakWidth}\right), \tag{5}$$

where $\Delta newI$ is the contribution to currently calculated intensity, $y_i(obs)$ and $y_i(calc)$ are the values of points in observed and calculated profile respectively, I is an old reflection intensity, Θ_i is the position of point in profile, Θ is the position of reflection and $peakWidth$ is the width of reflection.

Technically, it is possible to involve the whole profile for the calculation for each peak, but in practice only the points in a certain neighborhood are worth considering, since the exponential function yields practically zero for far enough points.

Algorithm 1 DoLeBail(*calcPattern*, *obsPattern*, *calcReflection*)

```
1: for i ← 0 to maxLeBailIterations do
2:     diff ← 0
3:     PlotPattern(calcPattern, calcReflections)
4:     for all calcReflections[i] do
5:         newI ← 0
6:         for all calcPattern points in range of reflection do
7:             calculate ΔnewI
8:             newI ← newI + ΔnewI
9:         end for
10:        diff ← diff + abs(newI − calcReflections[i].I)
11:        calcReflections[i].I ← newI
12:    end for
13:    if diff < LeBailLimit then
14:        break
15:    end if
16: end for
17: PlotPattern(calcPattern, calcReflections)
18: return diff
```

The DoLeBail function uses PlotPattern function, which simply reassembles the reflections into the diffraction pattern based on equation:

$$\Delta y_i = I \cdot \exp\left(\frac{-(2\Theta_i - 2\Theta)^2}{peakWidth}\right). \tag{6}$$

Second important function is DoLeastSquares, which realizes the least squares method based on Equation (4). The basic algorithm executes the matrix operations step by step and adjusts the structure parameters based on the result. The derivatives in Jacobi matrix are calculated by basic numerical differentiation, which paired with the use of floating point arithmetic introduces the danger of rounding errors. This problem is somewhat avoided by experimentally selecting the parameter used. It is worth noting that since each matrix row represents one point in discretized diffraction profile, the matrix is initially quite large, however the multiplication between its transpose and itself drastically reduces the size of matrix. Therefore the matrix inversion operates on relatively small square matrix and therefore the simple Gauss-Jordan elimination is used to solve it. After the remaining matrix multiplications are performed, the resulting vector is used to alter the structure parameters and reflection positions are recalculated from them. Finally a new calculated pattern is plotted.

It is worth noting that using this method of calculating least squares is not necessarily best one since it may operate with ill-conditioned matrices, however it proved to provide enough accuracy to be effective in Le Bail fitting, relatively simple to implement, and offered some parallelization potential.

Algorithm 2 DoLeastSquares(*calcPattern*, *obsPattern*, *calcReflection*)

```
1: for i ← 0 to maxLSQIterations do
2:     maxShift ← 0
3:     calculate dFx                          ▷ matrix of partial derivatives
4:     calculate Fx                           ▷ vector of differences
5:     result ←TRANSPOSE(dFx)·dFx             ▷ matrix multiplication
6:     result ←INVERT(result)                 ▷ matrix inversion
7:     result ← result·TRANSPOSE(dFx)         ▷ matrix multiplication
8:     result ← result · Fx                   ▷ matrix multiplication
9:     maxShift ← max(result)                 ▷ maximum of result vector
10:    AdjustUnitCell(result)
11: end for
12: for all calcReflections[i] do
13:    recalculate calcReflections[i].2Θ
14: end for
15: PlotPattern(calcPattern, calcReflections)
16: return maxShift
```

The AdjustUnitCell function only alters the structure parameters based on the result of least squares and takes into account the crystal system of currently calculated sample, since not all parameters have to be changed (*i.e.* angles in cubic system are always 90°).

It was also needed to choose the data representation of data on which the algorithm operates. The main issue was appropriate representation of the diffraction profiles, both calculated and observed, since they are represented by continuous graph and therefore need to be discretized for the use in algorithm. That meant choosing the equilibrium between precision and speed of execution. In our case we chose to define the step between two adjacent points and its chosen value lead to number of discrete points in order of thousands. The reflections present in any given crystal system is usually in hundreds and is naturally represented by a simple structure containing h, k, l triplet, d, Q, integrated intensity I and its position in diffraction profile Θ.

The implementation itself is realized in C++ language, since it offers good performance and is the most commonly used in GPU accelerated program.

2.1. Parallelization

The parallel implementation of code starts to be a standard way to get speed up. The computing power of the current graphic cards outperforms significantly the computing power of CPUs. For GPU implementation a well determined standard APIs like CUDA [5] and OpenCL were established. Current CPUs have several cores, each possibly having multiple execution threads. Parallel implementations developed for GPU acceleration can utilize multiple CPUs as well after modification or directly (when an OpenMP API is used). CPU or GPU computation acceleration was already implemented for several scientific problems related to crystallography: molecular dy-

namic simulations, quantum mechanics calculations of molecular structure or protein structure analysis (see [6, 7, 8]).

The data representation hints as to the parallelization approaches. There is quite a great amount of relatively independent data which allows to utilize to the potential of massively parallel platform such as general-purpose computing on graphics processing units (GPGPU).

First and probably most obvious data parallelism lies in the diffraction profile, which consist of points. For the purpose of Le Bail refinement, each point can be considered independent on any other in the profile (also one of the reasons of least squares method usage) and thus can be calculated by different threads without the need for much synchronization.

Second set of mostly independent data is in the number of reflection in which the given diffraction profile is decomposed. However, since the reflections in profile can overlap, the combination of parallel computation using both previously explained number of points in profile and reflection limits the data independence. Since each point can be composed from more than one reflection some form of synchronization has to be realized.

Third and the only truly independent data set is the number of different structure parameters estimates in the input of application. The only catch is the different number of iterations needed to refine each set, *i.e.* some estimates show no increase in quality after less iterations than others. Our GPU accelerated application uses all described data parallelism to some extent.

The task parallelism approach is much less applicable to the problem. As is obvious from previous section describing the sequential algorithm, the process and some of its parts are iterative and thus cannot be executed in parallel. Furthermore the dependence exists even between both main steps since least squares work with results from Le Bail fitting and vice versa.

2.1.1. CUDA

The most common technology used for parallel implementation on GPU is Compute Unified Device Architecture (CUDA). It is NVIDIA proprietary massively parallel computing platform and programming model allowing the use of GPU for general purpose computing. The GPU parallel execution differs in some ways from the conventional CPU parallel computing. Whereas the CPU parallel application usually uses at most tens of concurrent threads because there are corresponding number of cores on processing unit, the GPU parallelism uses hundreds or even thousands of threads. This relates to its architecture, since GPU comprises of hundreds of much simpler cores which share instruction scheduler, cache and memory. More precisely, the cores are organized into bigger units which share cache and scheduler, called streaming multiprocessors (SM), several of which then combine into a GPU. The information presented here is basic, more can be found for example in [9] or on NVIDIA websites.

The execution model conforms to the hardware architecture, it is basically a SIMD (single instruction multiple data) model with several changes in memory access requirements and thread divergence handling, called SIMT (single instruction multiple

threads). Therefore even though the SMs are basically independent units to certain extent comparable to CPU cores, the main strength lies within the SM, where tens of cores are capable of performing operations on different data. However this is also the main limitation, since cores in SM share one scheduler, all have to perform the same instruction.

The programming model is based on the architecture in such way, that the concurrently running threads are organized into blocks. One or more block of threads is then active (and executed) on each SM, the number depends mostly on properties of the blocks such as how much memory the threads require. Among others, registers and shared memory of SM is divided between all threads on SM to allow for fast context switching. The main thing to note is that the number of threads in blocks running on SM should be greater than number of cores, which allows effective hiding of memory access latency by executing different set of threads. For this reasons, running threads are divided into warps, which are sets of threads executed concurrently on cores in SM. Therefore if one warp has to wait for memory operation, another warp might be ready to execute and the fast context switching allows for it to run. This model also leads to optimization problem called occupancy, which has to find balance between complexity of threads (memory requirements) and number of threads executed, which helps hide memory latency.

The memory model referred to in previous paragraph is also more complex than usual. The fastest but smallest available memory are registers and shared memory in each SM. Then there is global memory and its specialized variants which are the much bigger in size (GDDR of a graphics card) but have significantly higher latency.

The main part of CUDA important for our work is the API allowing the use of specialized C/C++ language toolkit for the programming, compiling and running code executing on most of NVIDIA graphics cards. It also offers extensive debugging and optimization tools, which were among the reasons for selecting CUDA instead of one of its alternatives (*i.e.* OpenCL).

From the implementation standpoint, executing code on GPU requires several changes. Firstly, code to be run on GPU has to be organized into kernels, which are basically functions with some limitations. The biggest difference is that these function can access only memory on the device, thus all data has to be explicitly copied from computer (or host) memory into device memory and pointers to it passed to the kernels. Also, for each kernel has to be specified in how many blocks and threads per block it should run. One of the main focus in optimization is to correctly implement the kernels, since the implementation translates into memory requirements and then how many threads are in a block. Basically, the less fast memory (registers, shared) the kernel needs, the more warps can reside on SM and be quickly switched to hide memory latency. The following section focuses on this and other optimization problems more closely.

Similarly to the sequential implementation, the main kernels in final parallel implementation are LeBailKernel, diffKernel and lsqKernel. The two latter kernels performing the work of original DoLeastSquares function in sequential version and leBailKernel doing the work of DoLeBail.

2.2. Optimization for GPU execution

Program optimization is arguably more important in CUDA applications than in sequential implementations. The price of significant performance boost it offers is the need to adhere to certain rules and circumvent the limitations of GPU accelerated computing. Aside from design and implementation, the way to further improve the CUDA application is optimization. While the nvcc compiler is able to perform some basic optimizations, most of the work has to be done by the programmer. It is also worth noting that due to the fast evolution of GPUs, optimization is done on GPU type basis. This doesn't mean that application optimized for one graphics card won't perform well on newer or more powerful devices, only that application may not be able to fully utilize their full potential. Our application was optimized using NVIDIA GeForce 560Ti graphics card.

The optimization is an iterative process. First step is the analysis of the application performance and identification of possible issues and bottlenecks.

Without delving too deep into implementation details, the optimization of the application includes several methods. One relatively general rule is to perform as much computation as possible on GPU. Even though a single thread on GPU is usually slower than one executed by CPU, if there are memory transfers between host and device needed during the computation, it is more effective to perform this computation completely on the device. Using this information, in the final version of our application is almost entire Le Bail fitting performed on GPU with minimal memory transfers. Another changes included usage of constant memory type instead of registers, since constant memory in certain circumstances offers comparable performance to registers. As was mentioned previously, registers are somewhat limited resource and their lower usage may lead to better performance via increased occupancy of SMs. Beside this change, some changes in mathematical functions used by the application also improved performance slightly. Finally, sequential code optimization performed by the compiler also contributed to the improvements.

The process of kernel optimization is more complex and there are usually multiple solutions to issues found via profiling. For this kind of optimization, the Nsight profiling is an invaluable tool. The information obtained for each kernel is for example occupancy of each SM, the scheduler utilization by operation type, thread divergence and memory usage for each memory type. Based on this, most effective optimizations implemented in our application included changes in memory access patterns to use the properties of CUDA memory access, use of different memory type to lower latency or to increase occupancy and therefore allow better hiding of mentioned latency. Several changes were also done to the design of the algorithm, most notably substituting several critical sections with parallel reduction variant. Several other optimizations and their combinations were attempted but discarded in the end since they didn't have any positive effect.

The final improvement obtained by optimization is depicted in Table 1. It is apparent that the majority of the speed-up is obtained from the "global" optimizations since kernel improvement is less pronounced. The total speed-up may not seem worth the additional effort, but 60% increase in execution speed is not negligible.

Table 1. Achieved execution time of each kernel and total elapsed time on GTX 560 Ti

Parameter	Initial time [ms]	Optimized time [ms]	Speed-up [-]
leBailKernel	45.744	37.878	1.208
diffKernel	9.951	8.936	1.114
lsqKernel	24.144	20.678	1.167
Application	2545	1583	1.608

This result also cannot be generalized, since kernel profiling in our application didn't show any major issues which could be solved by optimization. Other applications or implementations may benefit more and optimization therefore should be always considered.

3. Achieved results

The main objective of this section is the assessment of the final application in terms of both achieved speed and precision. Usually a comparison to programs and algorithms performing similar task is done, however in case of Le Bail fitting this is somewhat complicated. The first issue is that Le Bail fitting is usually integrated into more complex applications, effectively eliminating the option to measure its performance. Second problem is that most of currently available programs and algorithms for crystal structure determination are not freely available and probably none is open source. Therefore the performance results of our CUDA accelerated application are measured to the sequential version of this program. To better illustrate the advantage of using GPU parallel computing, a reference version using CPU parallel solution with OpenMP implementation was created and included in comparison.

First and most important objective of our application was the faster processing speed. The achieved performance is summarized in Table 1 and speed-up plotted in graph depicted in Fig. 4. The sequential and CPU parallel versions were run on processor Intel Core i5-2500K with 4 cores at 4 GHz, 8 GB 1333 Mhz memory. The GPU parallel versions were run on similarly equipped machines using various graphics cards.

Table 2. The total execution time of sequential, OpenMP and CUDA program versions on different GPU

Data sample size	40	80	120	160	200
Sequential code	3.39 s	7.11 s	10.04 s	13.71 s	16.89 s
OpenMP	1.23 s	2.37 s	3.30 s	4.16 s	4.99 s
GTX 560 Ti	0.59 s	0.94 s	1.15 s	1.40 s	1.88 s
Tesla K40	0.27 s	0.37 s	0.43 s	0.50 s	0.61 s

As can be seen, the performance increase is significant. Even the arguably least powerful GPU, comparable in price with the used CPU, offers more than twice the

speed-up of the CPU parallel version in samples of sufficient size. That also shows one of the caveats of massively parallel computing, the need for enough data for processing. While this is true even for conventional parallelism, it is much more pronounced in GPU accelerated computing as there has to be enough work for hundreds of cores to really use all the potential they have to offer.

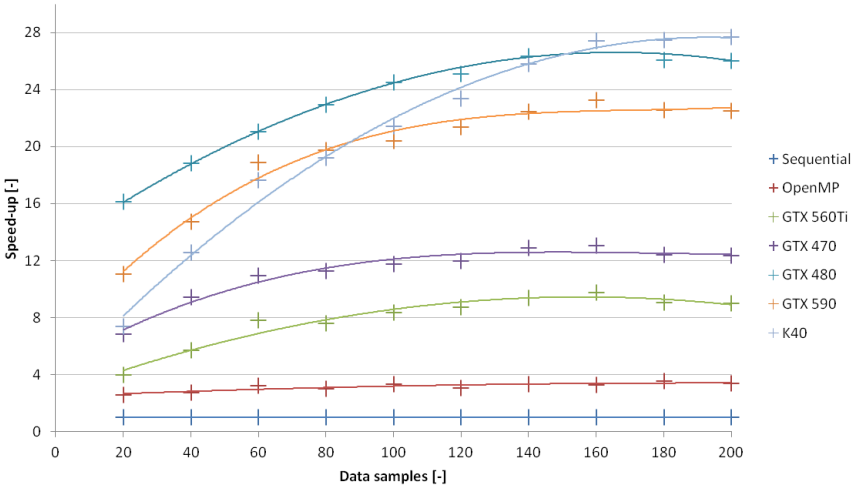

Fig. 4. The speed-up achieved by different program versions for moderate data size.

As for the precision standpoint, the goal was to preserve or ideally increase the quality of obtained results, measured by the previously mentioned figures of merits. The comparison was also made in respect to sequential version and the results were acceptable, even though several factors influenced the comparison. Most notably, each run of the parallel application offers slightly different results. This is caused mainly by the floating point operation errors, which are the unfortunate consequence of combination of parallel processing and single-precision format used in our program. However, the effect of these errors is not serious enough to warrant use of double-precision format, since this would cause major performance reduction.

The final application exhibits comparable precision to a sequential version, the mean change in figures of merits is several percent and is positive for most of the test samples.

4. Conclusion

The main objective of this article is to illustrate the advantages of massively parallel computing using CUDA in practical application, in this case represented Le Bail refinement. It also showed some of the limitations and specifics of the employment of CUDA. As is apparent from previous text, a significant effort may be required to fully utilize the performance of graphics accelerator and in some cases in may not even be possible due to the nature of solved problem. Nevertheless when the situation

permits, it offers far better performance than conventional parallel computing using CPU. It also can be more affordable than other massively parallel platforms, which is nowadays also an important feature. A long time has passed since the beginning of general-purpose computation on GPU and CUDA and the development of applications using it can benefit from a set of mature and useful tools as much as from the existing solutions and community behind it.

In conclusion there are many reasons to use the GPU accelerated computing to enhance and speed-up your application if there is a need to process a great amount of data. Otherwise the benefits may be outweighed by the amount of design and implementation changes, not mentioning the need to better understand the specifics of graphics processing units.

Acknowledgements. This work was supported by internal CTU grant No. SGS16 /122/OHK3/1T/18 of the Czech Technical University in Prague. This work was also supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/ 02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).

References

[1] MAŘÍK O., ŠIMEČEK I., *Acceleration of Le Bail Fitting Method on Parallel Platforms*, Proceedings of PANM17 conference, PANM'2014, 2014, pp. 136–141 (`http://panm17.math.cas.cz/PANM17_proceedings.pdf`).

[2] BORCHARDT-OTT W., *Crystallography: An Introduction*, 3rd Edition, Springer, Berlin, 2011.

[3] PECHARSKY V. K., ZAVALIJ P. Y., *Fundamentals of Powder Diffraction and Structural Characterization of Materials*, 2nd Edition, Springer Science, New York, 2009.

[4] Birkbeck College, University of London, *Advanced certificate in powder diffraction*, online (2006) (`http://pd.chem.ucl.ac.uk/pd/indexnn.htm`).

[5] NVIDIA Corporation, Cuda c programming guide, online (2014) (`http://docs.nvidia.com/cuda/cuda-c-programming-guide/`).

[6] ŠIMEČEK I., *A new approach for indexing powder diffraction data suitable for gpgpu execution*, Advances in Intelligent Systems and Computing 188 AISC (2013) 409–416.

[7] ŠIMEČEK I., *A new approach for indexing powder diffraction data based on dichotomy method*, Computational Science and Engineering (CSE), 2012, pp. 124–129.

[8] ŠIMEČEK I., ROHLÍČEK J., ZAHRADNICKÝ T., Langr D., *A new parallel and GPU version of a treor-based algorithm for indexing powder diffraction data*, Journal of Applied Crystallography **48** (1) (2015) pp. 166–170 (`http://dx.doi.org/10.1107/S1600576714026466`).

[9] SANDERS J., KANDROT E., *CUDA by example : An Introduction to General-purpose GPU programming*, Addison-Wesley, Upper Saddle River, NJ, 2010.