The handle http://hdl.handle.net/1887/37026 holds various files of this Leiden University dissertation.

**Author:** Schram, Raoul Diederick  
**Title:** Enumeration and simulation of lattice polymers as models for compact biological macromolecules  
**Issue Date:** 2015-12-15
Chapter 3

GPU acceleration of an efficient Monte Carlo polymer algorithm for melts

3.1 Introduction

Polymers have been studied for decades, both numerically and analytically. However, there is still much to learn. Due to the nature of polymer systems, which involve large length and time scales, relatively crude models, such as Monte Carlo lattice polymer models are still an important tool. In this chapter, we present a new implementation that uses the GPU (Graphics Processing Unit) to significantly shorten the time of simulations. The underlying simulation model is the elastic lattice polymer model, which has already proven its efficiency in a wide range of applications [24–26].

GPUs have been used in many other areas of statistical physics [27–29], for their efficiency in parallel computation. More specifically, it has been used for research in Molecular Dynamics simulations [30, 31] and Monte Carlo simulations [32]. The latter is an implementation of the bond fluctuation model. The elastic lattice polymer model has a number of advantages, including improved ergodicity and more efficient dynamics at high density.

\(^\text{0}\)R D Schram, GPU acceleration of an efficient Monte Carlo polymer algorithm for melts, \textit{to be submitted} (2015).
Our algorithm is most efficient for high density polymer melts. Melts of linear and ring polymers or a mix thereof are supported. The GPU implementation is many times faster than algorithms that use a CPU (Central Processing Unit). We also present a multicore CPU approach, but due to the higher theoretical compute power, the GPU algorithm manages to stay comfortably ahead.

The GPU algorithm is implemented in both the CUDA [33] and OpenCL [34] frameworks. In our description of the algorithm, we use the OpenCL terminology. For those that are familiar with GPU architecture or GPU programming in general, it should be easy to understand what the terms mean. Otherwise, the glossary of the OpenCL specification document provides an explanation of the terminology. OpenCL terms are presented in italic font. For our multicore implementation, we use the POSIX Threads (pthreads) library. This is a portable multithreading library, that is implemented on most major platforms.

This chapter is organized as follows. The next section introduces the elastic lattice polymer model, and notes some of the modifications that were made for the implementations described in this chapter. Sections 3.3 and 3.4 together describe the implementation of this model on the GPU. Section 3.5 describes a multithreaded implementation, which draws inspiration from the GPU implementation and provides a way to utilize CPU’s more efficiently. The following section shows the performance of the GPU implementation and compares it to both singlethreaded and multithreaded CPU implementations. The last part of this section takes a closer look at the scaling of the multithreaded CPU implementation.

Section 3.7 presents a set of simulations of linear polymer melts. They serve to show that the small number of modifications have no observable effect on the simulation results: we find results that are qualitatively equal to those found in previous studies. Finally, we summarize our findings in the last section.

### 3.2 Elastic lattice polymer model

The GPU implementation and the multicore CPU one are very similar at a basic level. They follow the lattice polymer model described in Ref. [35], with some minor modifications. Details of the specific algorithms are given in sections 3.4 and 3.5.
One ingredient of the elastic lattice polymer model is the notion of “stored length”. Normally, monomers are disallowed from sitting at the same place, the excluded volume principle. Here, there is one exception; if monomers are adjacent along the polymer chain they are allowed to sit on the same lattice site. A modification for our algorithms is that only two monomers are allowed to sit on the same site this way. Bonds that have zero length due to this are called units of “stored length”. In effect, this makes the chain shorter than the number of monomers in the polymer chain, and provides elasticity to polymers. The sequence of sites visited by the polymer is called the backbone of the polymer. A simple move is to move stored length along the backbone. In combination with a move that extends and retracts the tails of the monomer, the model provides reptation in an explicit way.

The lattice of the model is the face centered cubic (FCC) lattice. A way to construct an FCC lattice is to take a Cartesian lattice \((x,y,z)\), and remove all sites for which \(x + y + z\) is odd, then divide all distances by \(\sqrt{2}\). There are in total 12 unit vectors that determine the neighbors of a site on the lattice. We use four of those as basic unit vectors \(t, u, v,\) and \(w\):

\[
\hat{t} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \hat{u} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \quad \hat{v} = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \quad \hat{w} = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 0 \\ -1 \end{bmatrix}. \tag{3.1}
\]

All other unit vectors can be written as a sum of basic unit vectors, where we add only one or zero times each basic unit vector. Of the possible 16 combinations, 4 are not unit vectors: \(t + u, v + w\) and \(t + u + v + w = 0\). The remaining 12 describe edges on the FCC lattice. The (non-cubic) box has periodic boundary conditions with a lattice that has a box size \(L\), with the edges of the box along the \(t, u\) and \(v\) unit vectors. Thus, each lattice site can be represented by coordinates \((t, u, v)\), with \(0 \leq t < L, 0 \leq u < L, 0 \leq v < L\). In total there are \(L^3\) unique lattice sites.

Before an attempted move a monomer \(i\) from polymer \(j\) is selected. The selection process depends on the specifics of the implementation and details will be given in their specific sections. The moves are shown in figure 3.1.

The diffusive move is the simplest move, in which stored length is moved along the backbone. The bonds \((i, i+1)\) and \((i+1, i+2)\) are swapped if either of them contain stored length.

The transverse move allows the polymer to move sideways, as well as extend or retract the tail monomer. First, consider the case where monomer \(i\) is not at the tail end of
Figure 3.1: A polymer on a 2-dimensional version of the elastic lattice polymer model. Monomers 1 and 2 are on the same lattice site, and their bond is a unit of stored length. Monomer 1 is able to move to site A as an extension move. In the same manner, monomer 5 can retract to the site of monomer 4 to perform a retraction move. This move is accepted with probability $1/8$, in order to maintain detailed balance. Stored length moves along the backbone of the polymer using a diffusive move, for example moving monomer 2 towards monomer 3. Monomer 2 can also move to the empty site A in a forward transverse move. The reverse move is also possible: moving monomer 3 to the site of monomer 4. In our implementation, monomer 3 cannot move towards monomers 1 and 2, because a maximum of two monomers are allowed at the same site.

The polymer. There are two moves that are inverse to each other, that we denote as a transverse move. The first is the forward transverse move, in which monomer $i$ co-occupies a lattice site with its neighbor (stored-length), and is subsequently moved to a neighboring empty site. The other is the backward transverse move, in which monomer $i$ moves to a neighboring site, leaving behind empty space.

A forward transverse move is possible if exactly one of the bonds $(i - 1, i)$ and $(i, i + 1)$ is stored length. Let $r$ be the non-stored-length bond unit vector. Then there are 4 combinations of unit vectors $(p, q)$ such that $p + q = r$ form a triangle. Thus, instead of the current “short-cut” $r$ that the polymer currently takes, we can reroute it along the unit vectors $p$ and $q$. One of the 4 possibilities is randomly chosen. If the new site is empty, the move is accepted. Otherwise the move is rejected.

In case a monomer is selected such that it is at the very tail end, an extension or retraction move is attempted. If the bond attached to the monomer $i$ is stored length an extension move is attempted. We select a neighboring site, each with probability $1/16$. Since there are only 12 neighboring sites on the FCC lattice, the move is guaranteed to be rejected with probability of at least $4/16$. If the site is empty, the monomer is moved there.
In the reverse case, where we select an end that is not connected to a stored length bond, we attempt a retraction move. Since there is only one possible way to retract a monomer, it is necessary to reject the move with probability 7/8, in order to satisfy detailed balance.

The backward transverse move is triggered when monomer $i$ is not at the end of the polymer and has non-stored-length bonds on both sides $(p, q)$, with the bonds defined in the same direction (from tail to head or the reverse). Then, if $r = p + q$ is a unit vector, either bond is randomly chosen, and an attempt is made to move the monomer to that side. Since only two monomers are allowed on the same site at once, the move is rejected if the destination already has two monomers.

### 3.3 GPU programming

The GPU as a compute device is particularly adapted towards executing many identical instructions on different data in parallel. This is due to the specific architecture in which modern GPUs are built. At the highest level a GPU encompasses one or more compute units. A high-end GPU has several dozens of these. The compute units themselves are comprised of several sub-units, some of them graphics specific (e.g. geometry shaders, pixel shaders), others can be used for general purpose computing (e.g. processing elements, load/store units, local memory). The computation is done on the processing elements (PE). These PE's always execute the same instruction. Branching is done by “disabling” individual PE's using a predicate bit. An array of PE's in a single compute unit is called a SIMD array (Single Instruction Multiple Data). Thus, to build an efficient GPU program, we need to ensure that many identical instructions can be executed in parallel.

The most simple polymer MC simulation does not satisfy this requirement. Take for example the following structure for each step:

- Select a random monomer in the system.
- Attempt a Monte Carlo move for that monomer.
- If the move is valid, accept the move, otherwise keep the old configuration.

In this structure there is very little inherent parallelism. The obvious solution is to allow multiple monomers to be moved at once. We have to be careful not to move two
monomers that are close to each other simultaneously. Otherwise it might happen that two monomers are moved to the same place, or that two monomers next to each other along the chain become separated. To prevent this, it is required that two monomers moving at the same time are at least a distance of 3 bonds apart (on the lattice). This condition is sufficient with Monte Carlo moves that affect only neighboring sites, which is the case for the elastic lattice polymer model.

3.4 GPU implementation

As established earlier in the previous section, the most natural way to enable parallelism is to divide work spatially over the lattice. Each work-item is assigned its own small part of the lattice. These parts are assigned along the $t, u, v$ axes, and we’ll denote them as “cubes” although the edges of the boxes are not orthogonal. To allow for parallelism, the size of the work-item cubes should be small. Then, a large number of work-items can work together on a single system. On the other hand, the size cannot be too small either, because the moves of the work-items would interfere with each other.

To find the optimal size of a work-item cube, we first determine how the polymers are stored in the simulation. Data is stored per lattice site, to make sure that local moves are also local modifications in memory. The data for each lattice site is stored in the local memory of the GPU, which is on-chip memory that is about an order of magnitude faster than the global memory, and even greater benefits in terms of latency. At each Monte Carlo time step the data is loaded from the global memory into the local memory.

Each time a site is selected to attempt a move, the following information is needed to determine the outcome of a move:

1. Is there a monomer at the site?
2. At which site is the next monomer?
3. At which site is the previous monomer?
4. Is stored length present at the site?

Additionally, it is important to be able to determine which polymer is which, for observables that follow single polymers over a longer period of time such as the center-of-mass
diffusion. This information, however, does not need to be determined by the work-items during the simulation. Since we use the local memory for temporary storage, it is important to keep the data structure as compact as possible. Our implementation needs a total of 11 bits per lattice site. Of these, 8 are used to denote where the neighboring monomers in the chain are (relative to the current site), using two unit vectors ("prev" and "next"). Obviously, only one of the two is sufficient to uniquely define the polymers, but using both saves computing time and relieves (bandwidth) pressure on the local memory. In the case of an empty site, both these vectors are set to 0. If the polymer is a linear polymer, and the monomer at the site is the last monomer, the "next" unit vector will be 0 as well. The same goes for the "prev" vector of the first monomer. One bit is used to signify whether the site has stored length. This is why the amount of stored length is restricted to one per site. The last two bits are for storing a label, which keeps track of which polymer is which. All along the polymer, a label used for identification is coded with a unique string of bits, one per monomer, $N$ per polymer. Consequently, in case of a linear polymer, the number of polymers that we can track uniquely is limited to $2^N$. Unless our polymers are very small, this number is much greater than the number of polymers in our system. In case of ring polymers, we need to be careful that the label loops around, and might not be unique anymore if we start at a different monomer. Even then, in a homopolymer melt, the minimum ring polymer length is 21 (at $L = 96$, monomer density 1), which is quite small. It is necessary to use two bits per site, because two monomers might occupy the same site. During the Monte Carlo moves the label bits have to remain linked to the same monomer, and as such they are moved around in tandem.

The amount of 11 bits per site is not practical for storage. Of course, a possibility is to store it in a single 32-bit word, but this is quite wasteful. Storing two sites per word is already an improvement, but we use an even more efficient alternative. We group together $2 \times 2 \times 2$ lattice sites and store their data in the same part of memory. Two 32-bit words are used for the prev/next vectors, while one word contains both the stored length bits as well as the label. The layout is shown in detail in figure 3.2.

Since groups of $2 \times 2 \times 2$ blocks are stored together, we must ensure that neighboring work-items do not modify or read sites in the same group at the same step (otherwise atomic instructions are needed, which are much slower). This is achieved by assigning each work-item a block of size $4 \times 4 \times 4$. Each time a site is selected for a MC step, all work-items choose the same site relative to the $4 \times 4 \times 4$ block, with the appropriate offset. For example in figure 3.3, all work-items could select the bottom left sites of all the
colored work-item blocks. Careful examination shows that this indeed prevents situations in which work-items access/modify the same memory address at the same time.

Most modern day GPU’s are able to assign at least 48 KB of local memory to a single work-group. Thus, we use the largest possible work-group of $8 \times 8 \times 8 = 512$ work-items. The reason to take a work-group size as large as possible is that at the edges of the work-group block bonds connecting to other work-group blocks must be “frozen”, i.e. they must not be moved to maintain the integrity of the system (see figure 3.3 for an example). Having more work-items in a work-group, results in a larger work-group block, which has a lower fraction of frozen bonds. This improves the efficiency, and decreases the chance of artifacts due to frozen bonds.

It is clear that the boundaries between work-groups cannot always be at the same place on the lattice, because some bonds would be frozen indefinitely. This is prevented by choosing the position of the edges randomly after each MC time step ($t = 0, t = 1, \ldots$). The complete lattice is stored in the global memory, and is read by the work-items at the start of each time step, and is written back after each time step. In between the time steps the work-groups are synchronized.

The number of work-groups is a variable that is adjustable depending on the needs of the simulation. In the given implementation, the lattice dimensions are a direct result of
the number (and arrangement) of \textit{work-groups}. We chose a \textit{work-group} grid of $3 \times 3 \times 3$, which results in a lattice of size $96 \times 96 \times 96$.

At the start of the program, the polymer configuration is either read from a file or constructed from scratch, and loaded into the GPU buffers. After the initialization, the flow of the GPU program is as follows:

- Choose an offset $(t, u, v)$ that determines the boundaries of the \textit{work-groups}.

- Give control to the GPU with the given offset.

- The \textit{work-groups} read their part of the lattice as determined by the offset and their \textit{work-group} id, into the \textit{local memory}.

- Each \textit{work-item} attempts $4 \times 4 \times 4$ Monte Carlo moves (one diffusion and one transverse).

- The updated lattice is written back to the global memory of the GPU.

- The GPU as a whole is synchronized and a new offset is chosen.
• After a chosen number of time steps (usually $10^5$ for long polymers), the GPU buffers are read into the system memory and the polymer configuration is written to a file.

### 3.5 CPU implementation

Most computing systems these days have multiple cores. In cases where we have few simulations to run, or when access to compute power is relatively easy, some potential is wasted by only using one core. Starting from the GPU algorithm, it is not hard to derive an algorithm that also works on the CPU. There are some adjustments that are made of course, because the optimal number of concurrent threads is obviously much lower for a set of CPU cores.

We divide the lattice into small blocks of $k = 3 \times 3 \times 3$ lattice points. These blocks are similar to the blocks for each processing elements in the GPU program. Only in this case, the blocks are divided over all assigned CPU cores. The number of blocks does not have to be dividable by the number of CPUs, because one CPU doing one extra bit of work is not really a problem.

At each step, all CPUs choose the same site relative to the $k = 3 \times 3 \times 3$ block, and then attempt a Monte Carlo move (on odd steps a transverse move, otherwise a diffusion move), for all sites on their list. After a thread is done going through the list, it waits at a barrier for all threads to synchronise. Then a new site is taken randomly, and the process is repeated. After $2^k$ steps, one Monte Carlo time step has passed.

The selection of sites is less random than that of the GPU program (more sites are done “simultaneously”). This is to ensure that the synchronization costs do not become prohibitive. Synchronization of CPU threads is relatively costly (especially as compared to the SIMD structure of the work-groups), thus small blocks are chosen to reduce the number of barriers per Monte Carlo time step. Another way to reduce the relative penalty of the synchronization is to increase the lattice size, because there is more work to be done between each synchronization.

Multithreading is implemented using the pthreads library. Synchronization is achieved with a custom implementation that uses the `signal` and `conditional wait` functions of the pthreads library. The `barrier` functions in the pthreads library proved to be slower, and are also not implemented in OS X.
Chapter 3. *GPU acceleration of polymers in a melt*  

3.6 Benchmarks

3.6.1 Test setup and methodology

Benchmarks are done on three different systems. The first is a MacBook Pro Retina (2012), the other two are nodes on the supercomputer Cartesius in Amsterdam. The (relevant) specifications are given in Table 3.1.

<table>
<thead>
<tr>
<th>CPU Type</th>
<th>MacBook Pro</th>
<th>Cartesius 1</th>
<th>Cartesius 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Architecture</td>
<td>Intel Core i7</td>
<td>Intel Xeon</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>2.3 GHz</td>
<td>2.6 GHz</td>
<td>2.5 GHz</td>
</tr>
<tr>
<td># Cores/Threads</td>
<td>4/8</td>
<td>12/24</td>
<td>8/16</td>
</tr>
<tr>
<td># CPU</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>GPU</td>
<td>NVidia 650M</td>
<td>-</td>
<td>NVidia K40m</td>
</tr>
<tr>
<td>GPU Architecture</td>
<td>Kepler</td>
<td>-</td>
<td>Kepler</td>
</tr>
<tr>
<td># GPU</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Cuda version</td>
<td>6.5.51</td>
<td>-</td>
<td>7.0.27</td>
</tr>
<tr>
<td>Compiler</td>
<td>LLVM 6.1.0</td>
<td>gcc 4.7.7</td>
<td>gcc 4.7.7</td>
</tr>
</tbody>
</table>

The performance is measured by performing multiple runs with identical starting conditions: a melt with monomer density 1 of ring polymers with length 1000. Due to stored length, an average of 29% of the sites are empty. We perform 10 runs of 10 MC timesteps (= 10 \cdot 96^3 moves), and 10 runs of 20 MC timesteps. By comparing the average time difference between the two sets, we find an accurate estimate of how many MC steps are done per second. However, sometimes runs take significantly longer than expected (possibly due to hard drive spin-ups or interfering processes). To make the benchmarks more consistent, we drop the two slowest runs for each set. For the GPU runs, we increased the number of MC timesteps, because otherwise the runs are too short to be accurate, with a factor 20 for the MacBook Pro, and a factor 100 in case of the Cartesius supercomputer.

3.6.2 Performance

Performance is measured in the number of Monte Carlo moves (1 diffusion move + 1 transverse move). Performance of the different systems is shown in Table 3.2.
Chapter 3. *GPU acceleration of polymers in a melt* 39

**Table 3.2:** This table shows the performance of the different test systems. The simulation is done using a dense ring polymer melt. Detailed descriptions of the systems are given in Table 3.1. Performance of melts which include linear polymers is slightly worse for the GPU program. This small deficiency is in the range $0 - 5\%$. The second row shows the performance of an optimized version single core CPU version of the elastic lattice polymer model $[35]$.

<table>
<thead>
<tr>
<th>System</th>
<th>Compute type</th>
<th># units</th>
<th>moves per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacBook (Ivy)</td>
<td>CPU</td>
<td>1 thread</td>
<td>$9.9 \cdot 10^6$</td>
</tr>
<tr>
<td>MacBook (Ivy)</td>
<td>CPU (Barkema$[35]$)</td>
<td>1 thread</td>
<td>$22.4 \cdot 10^6$</td>
</tr>
<tr>
<td>Cartesius (Has)</td>
<td>CPU</td>
<td>1 thread</td>
<td>$12.5 \cdot 10^6$</td>
</tr>
<tr>
<td>Cartesius (Ivy)</td>
<td>CPU</td>
<td>1 thread</td>
<td>$9.3 \cdot 10^6$</td>
</tr>
<tr>
<td>MacBook (Ivy)</td>
<td>CPU</td>
<td>8 threads</td>
<td>$41.6 \cdot 10^6$</td>
</tr>
<tr>
<td>Cartesius (Has)</td>
<td>CPU</td>
<td>23 threads</td>
<td>$80.0 \cdot 10^6$</td>
</tr>
<tr>
<td>Cartesius (Ivy)</td>
<td>CPU</td>
<td>24 threads</td>
<td>$37.8 \cdot 10^6$</td>
</tr>
<tr>
<td>MacBook (650M)</td>
<td>GPU/OCL</td>
<td>1 GPU</td>
<td>$91 \cdot 10^6$</td>
</tr>
<tr>
<td>MacBook (650M)</td>
<td>GPU/Cuda</td>
<td>1 GPU</td>
<td>$105 \cdot 10^6$</td>
</tr>
<tr>
<td>Cartesius (K40m)</td>
<td>GPU/Cuda</td>
<td>1 GPU</td>
<td>$805 \cdot 10^6$</td>
</tr>
</tbody>
</table>

Table 3.2 compares the performance of the CPU programs and the GPU program. An alternative single CPU core implementation $[35]$ is included for comparison. One has to be careful to compare the number of moves per second of different algorithms, because the (effective) density, entanglement length, move acceptance probabilities may all benefit or adversely affect this metric, making comparisons difficult at best. Comparisons between the algorithms described here should be much cleaner, given that the model is practically equal for them.

Despite the relative weak GPU in the MacBook, it outperforms the fastest tested machine using the multithreaded CPU program. This shows that the GPU program is much more efficient than a CPU in these calculations. Comparing the fastest single CPU core (Haswell) to the fastest single GPU (K40m), we find that the GPU has the advantage by a factor of approximately 64. Alternatively, comparing the optimized single core program $[35]$ to the GPU, we find a gap that is a factor of 35. Using multiple threads decreases this gap to about a factor of 10, still a significant margin.

We have plotted the performance of the mutithreaded program as a function of the number of threads in figure 3.4. It is apparent that the scaling of the CPU program is not perfect: especially in the multi CPU case, using 10 threads is far from 10 times faster than using only one. The main source of this phenomenon are synchronization costs. Simply looking at the CPU usage during a run reveals that the cores are idle at least
some of the time, presumably waiting for other threads to finish their work, or send the signal to continue. Another interesting bit of information is that the multi CPU setup is slower than initially expected at low (but larger than one) numbers of threads. The most likely explanation is that the (Linux) kernel spreads the threads over as many CPU’s as possible. In cases that are less communication intensive, this is the desired behavior, but in our case it just makes synchronization significantly slower. This can be improved, by setting all threads to a single CPU (for low thread counts), but peak performance is unlikely to be (significantly affected), because only one of the CPUs can be used this way.

3.7 Test case: a melt of linear polymers

In Ref. [36] we discussed the application of the GPU model to a melt of ring polymers. Here, we present results for a melt of linear polymers. The results are meant to illustrate that our modifications to the Monte Carlo model facilitating the strengths of the GPU do not alter the physics significantly. The data presented in this section is not sufficient however to provide new insights, simply because the invested computing time was much lower compared to other studies. The polymer lengths and simulation times of the runs that were used in this chapter are given in Table 3.3.

We first look at the static properties of linear polymers in a melt. An indicative observable is the mean-squared distance $r^2(g)$, where $g = |i - j|$ is the number of bonds between


Table 3.3: A summary of all MC simulations done for this chapter. All simulations have a lattice size of $L^3 = 96^3$. The first column shows the number of monomers of the ring polymers in the melt. In one unit of time, each monomer attempts each type of move statistically once. The number of time units is shown in the second column. The column “# sys” denotes the number of systems run in parallel (for multi-GPU setups). “Eq. time” is the amount of MC time steps that are done before measurements are done. The last column shows the number of hours it would take a single reference GPU (NVidia K40m Tesla).

<table>
<thead>
<tr>
<th>$N$</th>
<th>MC timesteps</th>
<th># sys</th>
<th>Eq. time</th>
<th>GPU hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>$100 \cdot 10^3$</td>
<td>1</td>
<td>$15 \cdot 10^3$</td>
<td>$&lt; 0.1$</td>
</tr>
<tr>
<td>79</td>
<td>$300 \cdot 10^3$</td>
<td>1</td>
<td>$74 \cdot 10^3$</td>
<td>$&lt; 0.1$</td>
</tr>
<tr>
<td>99</td>
<td>$500 \cdot 10^3$</td>
<td>1</td>
<td>$155 \cdot 10^3$</td>
<td>$&lt; 0.1$</td>
</tr>
<tr>
<td>149</td>
<td>$2000 \cdot 10^3$</td>
<td>1</td>
<td>$591 \cdot 10^3$</td>
<td>0.6</td>
</tr>
<tr>
<td>249</td>
<td>$1000 \cdot 10^4$</td>
<td>2</td>
<td>$319 \cdot 10^4$</td>
<td>6.1</td>
</tr>
<tr>
<td>499</td>
<td>$5000 \cdot 10^4$</td>
<td>2</td>
<td>$1466 \cdot 10^4$</td>
<td>30.5</td>
</tr>
<tr>
<td>999</td>
<td>$7000 \cdot 10^5$</td>
<td>2</td>
<td>$1257 \cdot 10^5$</td>
<td>427.4</td>
</tr>
</tbody>
</table>

monomer $i$ and monomer $j$. We expect a smooth transition from a swollen polymer $r^2(g) \sim g^{2\nu}$, with $\nu \approx 0.5876$ the growth exponent for a swollen polymer. For larger chemical distances the excluded volume is screened and we expect Gaussian polymer statistics: $r^2(g) \sim g$. We have plotted the results in figure 3.5. The mean-squared distance indeed behaves as expected.

Next we explore the dynamics as produced by the GPU algorithm. To keep this section as compact as possible, we concentrate on the long time and length scales. We probe this by studying the end-to-end vector $R_{ee}$. Since we know that $r^2(g) \sim g^1$, it must also be true that $R_{ee}^2 \sim N$. The dynamics at the largest scale are expressed in the auto-correlation function:

$$\langle R_{ee}(t) \cdot R_{ee}(t+dt) \rangle \sim N \exp(-dt/\tau_d), \quad (3.2)$$

where $\tau_d$ is the disentanglement time of the polymer. From other simulations, Monte Carlo and molecular dynamics, this is found to scale with length roughly as $\tau_d \sim N^{3.4}$. If our simulations are long enough, we expect to see the same exponent. We have plotted the auto-correlation function in figure 3.6. The observed behavior is indeed as expected for $N \geq 500$, and we find $\tau_d \sim N^{3.35}$.

A dynamic quantity that shows aspects of both the short and long time regimes, is the squared displacement of the center of mass $r_{cns}^2(t)$. At short times $t < \tau_E$, with $\tau_E$ the entanglement time, subdiffusive behavior is observed in simulations and experiments, which is in contrast with the tube model, that predicts diffusion on this time scale. We
Chapter 3. GPU acceleration of polymers in a melt

Figure 3.5: The average mean-squared distance $r^2(g)$ as a function of chemical distance $g = |i - j|$, with monomer numbers $i$ and $j$. It shows a simple crossover between $g^{2\nu}$ and $g^\lambda$. The lengths of the polymers are given in Table 3.3.

Figure 3.6: On the $y$-axis, the auto-correlation function of the end-to-end vector is rescaled by the length of the polymer. On the $x$-axis the time is divided by the $N^{3.35}$ to obtain a collapse. Thus we find numerically that the disentanglement time $\tau_D \sim N^{3.35}$. The longest two lengths are $N = 999$ and $N = 499$. All used lengths in this plot are given in Table 3.3.
find approximately \( \langle r_{\text{cms}}^2(t) \rangle \sim t^{0.84} \). Then for very long lengths and \( \tau_E < t < \tau_D \), we expect to find \( \langle r_{\text{cms}}^2(t) \rangle \sim t^{0.5} \). Unfortunately, our polymers are not yet long enough to show this asymptotic behavior. For \( t > \tau_D \), we find simple diffusive behavior as expected. The resulting curves are shown in figure 3.7.

Finally, the diffusion coefficient \( D = \lim_{t \to \infty} \langle r_{\text{cms}}^2(t) \rangle / (6t) \) is measured. The result is plotted in figure 3.8. A crossover is seen from approximately \( D \sim N^{-1.2} \) to \( D \sim N^{-2.4} \). This is in agreement with experiments [37–39], Molecular Dynamics studies [40] and previous Monte Carlo studies [39]. However, this is not in agreement with standard reptation theory [1, 41–45], which predicts \( D \sim N^{-2} \).

### 3.8 Conclusion and discussion

It is clear that with the right implementation, a GPU is very efficient in performing polymer simulations. We implemented the elastic lattice polymer model, which has been used before and proven its efficiency. Our implementation on the GPU is about 35 times faster than any previous works on the CPU. We have also implemented a multithreaded
algorithm. This implementation showed that scaling on the CPU is a problem that impedes efficient use. This is mainly due to synchronization costs. Nonetheless, a sizable speedup over single core implementations is achieved of 3.5.

We applied the GPU algorithm to a test-case of linear polymers in a melt. Both the static and dynamic properties are in line with previous studies involving CPU based Monte Carlo simulations. This assures us that it is very unlikely that the minor modifications made have an effect on the physical results.

Finally, there might be the question of wider application of the algorithm to more complex problems. It is our belief that our approach is general enough that it should be extensible, on the condition that there are only short range interactions between polymers. In case the problem demands the storage of more than 1 bit extra per site, the amount of local memory assigned per site has to be increased, and consequently the number of work-items per work-group has to be decreased.

The code for the GPU simulations is available on request by emailing the author of this thesis.