Effiziente Implementierung von mehrzentrigen molekulardynamischen Potentialmodellen mit Cuda
—
Efficient Implementation of Multi-Center Potential Models in Molecular Dynamics with Cuda

Andreas Kirsch

October 21, 2011

This paper discusses the parallelization of the molecule interaction calculations in the molecular dynamics simulator MarDyn with Cuda. It describes the performed optimizations on the code and how support for more advanced potential models was added. It concludes with benchmarking results and suggestions for further work.

1 Introduction

Molecular dynamics deals with the simulation of atoms and molecules. It estimates the interaction between a high number of molecules and atoms using force fields which are modeled after quantum mechanics equations. Such simulations are important for many different areas in biology, chemistry and material sciences. For example you can examine the folding of proteins or the nanoscopic behavior of materials.

The simulations are sped up using sophisticated approximations and algorithms. I have used MarDyn, a molecular dynamics application, for my work. It has been co-developed by the Chair of Scientific Computing at the Technische Universität München and a description of it can be found in [Buc10].

1.1 Parallelization

Molecules only have strong interactions with molecules that are nearby. One of the first approximations is to take into account these strong interactions only and ignore weaker long-distance ones. This space locality of the calculations leads to the Linked Cell-algorithm. It divides the simulation space into a grid of cells. Only interactions between molecules inside each cell and with nearby cells have to be calculated. This reduces, assuming an upper density limit of the molecules, the complexity from $\mathcal{O}\left(N^2\right)$ to $\mathcal{O}\left(N\right)$. Thus different cells that are not neighbors show a data independency which can be used for parallelization.

One way of parallelizing the calculations is using clusters of processing units which is also discussed in [Buc10]. Another option is the parallelization using modern GPUs which offer multiple processing cores and high parallelism in a single computer.

There are two programming frameworks for GPUs: Cuda[1] and OpenCL[2]. [Ore10] describes how to parallelize MarDyn on a GPU using OpenCL. I am using Cuda.

1.2 Cuda

In Cuda 2.x the GPU is modeled as having multiple **streaming multiprocessors** (**SM**) which execute multiple threads of so-called kernels. A **kernel** is the term used in Cuda for a GPU program.

Each kernel can have many **threads** that are run in parallel. The threads are grouped in **thread blocks** of at most 1024 threads and launched in a **grid** of at most 65535^2 thread blocks.

Each multiprocessor has at most 1536 **resident threads** which can be resumed with almost no overhead and runs up to 3 thread blocks. That is all threads of one thread block run on the same multiprocessor. 32 threads (belonging to the same thread block and kernel) constitute a warp. A **warp** is executed in real parallel by the multiprocessor. Internally the multiprocessor uses a principle called **SIMT** which stands for *Single-Instruction Multiple-Thread*.

This means that the instructions of a GPU program run in lock-step for all threads of a warp. If there is a branch that is taken differently by different threads of a warp, both code paths are executed by the multiprocessor sequentially. Register and memory operations are only *masked out* for the inactive threads.

There are different types of memory: **global memory** and **constant memory** are shared between all kernels and thread blocks; **shared memory** is on-chip memory on a multiprocessor and only shared between threads of a single thread block; **local memory** and **registers** only belong to one thread.

There are 64 KB of constant memory available and 48 KB of shared memory available per multiprocessor. Each thread has access to 512 KB of local memory. There are 32768 registers that are divided up between the threads.

[1] `http://developer.nvidia.com/category/zone/cuda-zone`
[2] `http://www.khronos.org/opencl/`

Additionally there is a 16 KB big L1 cache in each multiprocessor for global memory. There is a second cache mode in which only 16 KB of shared memory are available and L1 cache is 48 KB big instead.

The register count used by one thread and the amount of shared memory used per thread block determine how many threads and thread blocks can be executed on the same multiprocessor concurrently.

Most importantly Cuda uses threads and thread blocks to hide the high latency of memory accesses. Thus it is imperative to keep the multiprocessors busy with a high amount of threads, so a multiprocessor can resume a different warp, while the current one is waiting for a memory access to finish.

For more information see [NVI11b] and [NVI11a].

2 Chronology of the IDP

My IDP was intended to extend on the work of Orendt, [Ore10]. The idea was to port his OpenCL code to Cuda and continue from there. Porting his code to Cuda was a straight-forward API change. However, I found his code to be impossible to read, unmaintainable, and not very optimized either. The logic behind the code was not clear or well explained.

Consequently we decided that I would rewrite everything in Cuda and optimize it from the beginning. This took longer than expected and while the resulting parallelism and the logic behind it were clear in the code, code complexity was an issue. The code consisted of three helper functions and two kernel functions in mainly two files and when I went and integrated support for multiple centers and multiple components, it became clear that the current design was not clear enough to be enduring further feature additions and lacked the flexibility for quick changes.

Treating the old implementation as a prototype and knowing about many possible traps and fallacies, I set out and rewrote everything again. This time the focus was on modularity and separation of concerns instead of performance. Code architecture was the most important thing this time around. I scaffolded the new version around the old code which was working correctly and already optimized. I embedded it into the new design step by step.

3 Code Design and Architecture

The code resides in `src/cuda/` and only minor changes have been made to the original MarDyn code. These changes will be described in detail later as well.

The Cuda code shifts the molecule interaction calculation to the GPU. It calculates the interactions between different molecules using different potential models as specified in the component description, see [Buc10]. A Cuda application performs basically three steps:

1. upload data to the GPU
2. process data on the GPU

3. download results from the GPU

Different information has to be uploaded in MarDyn. Information about the different molecule types (**component descriptions**) has to be uploaded only once. The position and orientation of every molecule has to be uploaded for each time step. Later the calculated forces and torque have to be downloaded, as well as the calculated potential and virial statistics.

The Linked Cells-algorithm is used in MarDyn. The simulation space is divided into a grid of cells and the interaction between molecules of the same and of neighboring cells have to be calculated. The length of the cells is chosen equal to the cutoff radius of the potential functions to make sure that only interactions of direct neighbors have to be calculated. This leads to a natural parallelization approach:

- for **intra-cell molecule interactions** each thread can process the molecules of one cell, and
- for **inter-cell molecule interactions** each thread can process a different pair of neighboring cells.

3.1 General Architecture

The Cuda code is made up of different components. Each component has exactly one responsibility. The intention is to make it clear where changes go and make it easier to navigate the code.

A **component** usually consists of CPU and Cuda code which interact in a certain way. For this there usually are `component.cpp` and `component.h` files, which contain the CPU-side code, and a `component.cum` file, which contains the Cuda code. `.cum` stands for **cu**da **m**odule.

Normally you pass parameters as kernel parameters to Cuda. Separation of responsibilities is the main goal of having different components. Thus passing all parameters as kernel parameters is not a good approach. The CPU code of each component needs a way to interact with the Cuda code and the 'glue' code should be part of the components and not be kept somewhere else[3].

The solution is simple: Cuda supports global variables and the CPU/Cuda interface in each component uses global variables to exchange data. Different namespaces can be used for each component to avoid name conflicts. See Figure 1.

Moreover, instead of global memory variables constant memory variables could be used. Constant memory has its own on-chip cache, so this is desirable. Since only little data is stored in constant memory, cache misses are unlikely.

3.2 Component Overview

Most components implement one of the two `CUDAComponent` interfaces:

- `CUDAInteractionCalculationComponent` is used by components that perform data exchange operations at each time step, and

[3] the first implementation used kernel parameters and the parameter count quickly became unacceptable

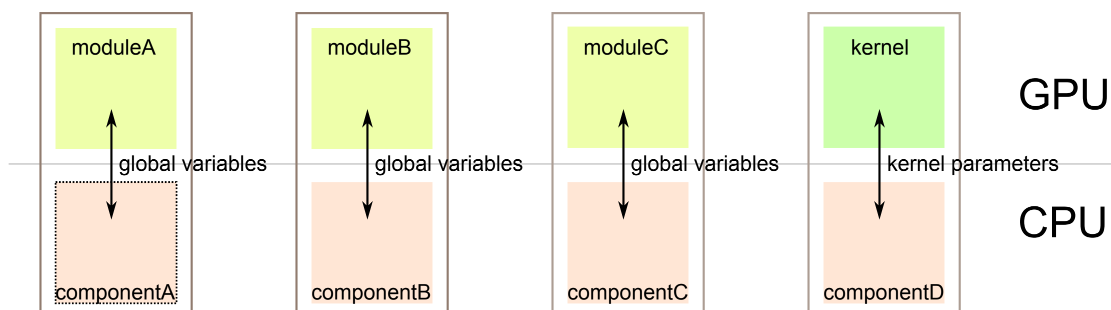


Figure 1: components and Cuda modules

- `CUDAStaticDataComponent` is used by components that only need to upload data once.

There are several components:

CellProcessor contains logic to efficiently loop through the molecules of a single cell or a cell pair,

ComponentDescriptor manages the molecule component descriptions (eg the number of Lennard-Jones centers, their positions, etc),

MoleculeInteraction links all components together and calculates the forces,

GlobalStats keeps track of potential and virial statistics,

MoleculePairHandler calculates the interactions between two given molecules,

MoleculeStorage manages all per-molecule data like positions, forces and orientations,

DomainTraverser enumerates all cell pairs in a way that allows for maximum parallelization, and

PotForce calculates the force between Lennard-Jones centers, dipoles, etc.

See Figure 2 for an overview how the different components interact with each other.

There are additional Cuda modules that contain helper and utility functions. All the `.cum` files are included in `kernel.cu`, which also contains the kernel entrance functions `processCellPair` and `processCell`, which are called by **MoleculeInteraction** to calculate the forces between all the molecules in a cell pair or in a single cell.

3.3 Cuda Helper Classes and Functions

`helpers.cpp/h` contains C++ wrappers for Cuda functionality that is used. It implements an internal DSL (see [Fow06] for a definition) to simplify working with the Cuda API. See Figure 3.

3.3.1 Device Buffers and Global Variables

Managing device memory is done by creating instances of `CUDA::DeviceBuffer<DataType>`. Methods are provided to resize the buffer and to copy data from and to the GPU. Global variables are mapped to `CUDA::Global<DataType>` objects. They provide a `set` method that is specialized for pointer global variables to take `CUDA::DeviceBuffer<DataType>` objects. Example:

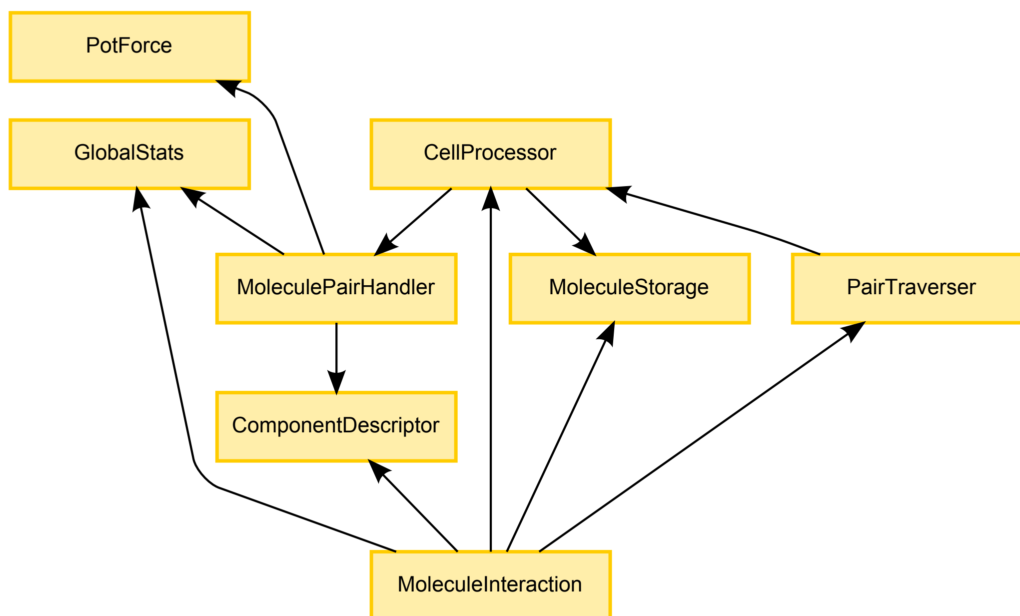


Figure 2: component dependencies (the arrows show which components each component uses, eg MoleculeInteraction uses GlobalStats)

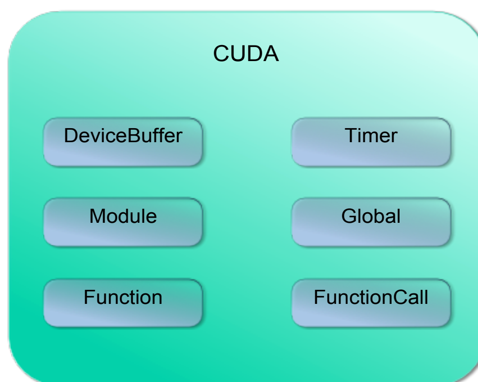


Figure 3: main Cuda helper classes

Listing 1: Cuda helper classes for device memory and globals

```
class GlobalStats : public CUDAInteractionCalculationComponent {
  CUDA::Global<CellStatsStorage *> _cellStats;
  CUDA::DeviceBuffer<CellStatsStorage> _cellStatsBuffer;

  GlobalStats( const CUDAComponent &component ) :
    _cellStats( _module.getGlobal<CellStatsStorage *>("cellStats") )
  /* ... */

    _cellStatsBuffer.resize( _linkedCells.getCells().size() );
    _cellStatsBuffer.zeroDevice();
    _cellStats.set( _cellStatsBuffer );
  /* ... */
};
```

3.3.2 Calling Cuda Functions

The `Function` class wraps Cuda functions. It has a `call` method that returns a `FunctionCall` object that can be used to specify the grid and block sizes and parameters before running the Cuda kernel by calling its `execute` method. Example:

Listing 2: Cuda helper classes for function calls

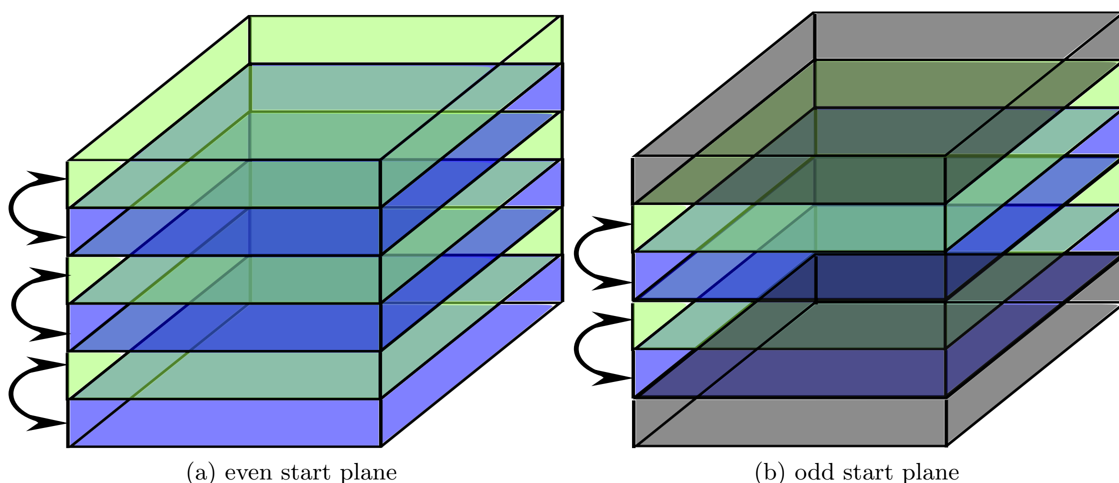
```
  CUDA::Function _convertQuaternionsToRotations;
/* ... */
  MoleculeStorage( const CUDAComponent &component ) :
    _convertQuaternionsToRotations( _module.getFunction( "
       convertQuaternionsToRotations" ) )
/* ... */
    _convertQuaternionsToRotations.call().
      parameter(quaternionBuffer.devicePtr()).
      parameter(currentIndex).
      setBlockShape(MAX_BLOCK_SIZE, 1, 1).
      execute(currentIndex / MAX_BLOCK_SIZE + 1, 1);
```

3.3.3 `cutil_double_math.h`

The Cuda SDK provides a helper file called `cutil_math.h` which adds overloaded operators and functions for most Cuda datatypes except `double2`, `double3` and `double4`. `cutil_double_math.h` adds support for these datatypes. This makes it possible to switch between single- and double-precision using a simple preprocessor #define.

3.3.4 Domain Traverser

The simulation space is divided into a grid of cells. The domain traverser iterates over all neighbor pairings in the volume while maximizing parallelism. Each cell inside the volume has 26 neighbors. There are 13 directions because of symmetry: it does not matter if you calculate the interactions between cell A and B or cells B and A. This



(a) even start plane

(b) odd start plane

Figure 4: schematic of the two steps the pair traverser performs

symmetry allows for a straight-forward parallelization. I am going to introduce it using x-axis as interaction direction:

1. calculate the interactions between all cells $(2*k, y, z)$ and $(2*k+1, y, z)$ in parallel (even case)
2. calculate the interactions between all cells $(2*k+1, y, z)$ and $(2*k+2, y, z)$ in parallel (odd case)

It is easy to see that this covers all neighbor pairings along the x-axis. The algorithm takes the even or odd layers parallel to the yz-plane and computes the interaction with the neighboring cells (along the x-axis).

All other directions can be treated like this. You only have to make sure that there are no out of boundary accesses.

The cells are flattened into a one-dimensional array and the CPU code calculates:

- the index of the first cell (`startIndex`)
- the size of the two-dimensional grid that constitutes the parallel layers (`dimension`)
- offsets to move in x-, y- and z-direction in the one-dimensional cell array, whereas this constitutes a local coordinate system, such that the x- and y- span the two-dimensional grid and z- moves along the layers (`gridOffsets`)
- offset to get the neighbor cell for a cell along the currently used direction(`neighborOffset`)

With these parameters, the `processCellPair` kernel can be launched in a grid of dimension$\times$ number of layers thread blocks, that is one thread block per cell pair.

3.4 Molecule Storage

The molecule storage component manages everything related to molecule data. The molecules of all the cells are flattened into a one-dimensional array. For each cell the start index into this array is stored in a `cellStartIndices` array. Since the start index

of one cell is the end +1 index of the cell before it, it is easy to extract all necessary information from this array.

Molecule data such as position, orientation, total force and torque, and molecule type, are stored non-interleaved into different buffers: `moleculePositions`, `moleculeRotations`, `moleculeForces`, `moleculeTorque` and `moleculeComponentTypes`.

The molecule orientation is stored and uploaded as quaternions but converted to conventional 3×3-matrices on the GPU.

The Cuda code wraps accesses to the data through a `MoleculeStorage` class. The `Molecule` class wraps accesses to it. This makes it possible to write the molecule interaction code without having to care for the actual implementation.

There is also a `MoleculeLocalStorage<blockSize>` class. It provides a cache for frequently used molecules. Its objects are supposed to be instantiated as `__shared__` objects and there are methods to load, merge and commit molecules from the cache to the regular storage.

3.5 Component Descriptors

Each molecule has a related `moleculeComponentTypes` attribute, which specifies which component the molecule belongs to.

The component descriptor specifies common properties of molecules:

- number of sites: Lennard-Jones centers, charges and dipoles
- position of a site relative to the molecule centers
- site-specific properties like ϵ and σ parameters for Lennard-Jones centers or $|\mu|$ for dipoles.

The properties of the different sites are stored in fixed-size arrays. Their sizes are set by the compile-time constants `MAX_NUM_LJCENTERS`, `MAX_NUM_CHARGES` and `MAX_NUM_DIPOLES` respectively. The number of supported components is set by the compile-time constant `MAX_NUM_COMPONENTS`.

The mix coefficients ξ and η are stored in two 2D arrays.

3.6 Global Statistics

The potential and virial are calculated per cell on the GPU. For this the GPU code creates shared arrays for one potential and one virial value per thread in a thread block, so the values can be updated by their threads without synchronization. The `CellStatsCollector` class has a `reduceAndStore` method which sum-reduces the two arrays into cell potential and cell virial values.

It does this by first summing all potentials inside a warp without any locking. Then it loops over all warps and reduces them sequentially.

The CPU code then reduces these per-cell values into per-domain values. Potential and virial values are calculated per molecule pair and added to the cells the molecules belong to. In the case of molecules from the same cell, the values are added to the same cell twice. Now the domain values, as sum of all cell values, count each potential and virial value twice, so the code halves the values.

There is also a difference compared to the regular MarDyn code: the force interaction code in MarDyn defines `distanceAtoB` as `positionA - positionB`, while the Cuda code defines it as `positionB - positionA`, which makes more sense semantically. Because of this sign change, the calculated GPU virial has to be sign inverted, too, to match the CPU virial value.

3.7 PotForce

This component mostly mirrors the corresponding `potforce.h` file in the CPU code except for sign changes as described in the previous paragraph. Only the dipole-dipole code has been refactored to use fewer operations.

3.8 Molecule Pair Handler

The molecule pair handler resolves the interactions of a molecule pair. All supported potential models are independent of the neighborhood of the two molecules, so only the two molecules are passed as parameters.

First the pair handler tests whether the molecule centers are within the cutoff radius. Otherwise no interactions are calculated.

After that the molecule pair handler loads the component descriptors for each molecule and iterates through all sites as necessary to calculate the interactions. It also adds the calculated potential and virials to the global statistics.

To calculate the interactions it needs to know the distance between the sites of the molecules. For this it transforms the relative position of each site as specified in the component descriptors using the `moleculeRotations` matrices that have been created from the quaternions. Once all needed values have been determined, it uses the methods in the PotForce component to determine the actual forces and torque.

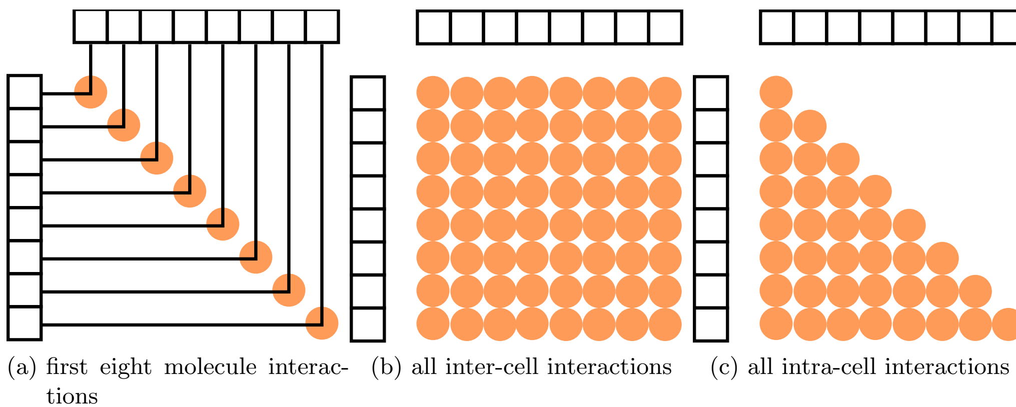
3.9 Cell Processor

The cell processor processes all molecules in the same cell (intra-cell) or between two cells (intra-cell). The reference approach is to iterate through all molecule pairs in both cells and calculate the interactions between them:

Listing 3: reference inter-cell block processing

```
__device__ void processCellPair( const int threadIndex, const CellInfo &
    cellA, const CellInfo &cellB ) {
  for( uint indexA = cellA.startIndex ; indexA < cellA.endIndex ; indexA++
      ) {
    StorageMolecule moleculeA(indexA);

    for( uint indexB = cellB.startIndex ; indexB < cellB.endIndex ; indexB
        ++ ) {
      StorageMolecule moleculeB(indexB);
      MoleculePairHandler::process( 0, moleculeA, moleculeB );
      moleculeB.store();
    }
```



(a) first eight molecule interactions (b) all inter-cell interactions (c) all intra-cell interactions

Figure 5: schematic of molecule interactions between two cells

```
    moleculeA.store();
  }
}
```

The intra-cell case is similar. The only difference is that only half of the interactions have to be calculated, because order[4] does not matter:

Listing 4: reference intra-cell block processing

```
__device__ void processCell( const int threadIndex, const CellInfo &cell )
    {
  for( uint indexA = cell.startIndex ; indexA < cell.endIndex ; indexA++ )
      {
    StorageMolecule moleculeA(indexA);

    for( uint indexB = cell.startIndex ; indexB < indexA; indexB++ ) {
      StorageMolecule moleculeB(indexB);
      MoleculePairHandler::process( 0, moleculeA, moleculeB );
      moleculeB.store();
    }
    moleculeA.store();
  }
}
```

Thread Block Cell Processor The main idea in the parallelized approach is to divide the cells into smaller blocks that can be kept in shared memory and process interactions inside between the different blocks. Parallelized matrix multiplication is based on the same idea. See [NVI11b].

Thus the cell processor divides the cells into blocks. Each block contains exactly one molecule for each thread in a thread block. The code then iterates over all block pairs

[4]molecule A with molecule B is the same as molecule B with molecule A

in both cells and calculates the interactions between molecules of the two blocks. The inter-cell case is straight-forward:

Listing 5: inter-cell block processing (thread block cell processor)

```
for( int blockIndexA = 0 ; blockIndexA < numBlocksA ; blockIndexA++ )
  for( int blockIndexB = 0 ; blockIndexB < numBlocksB ; blockIndexB++ )
    processBlock<BPT_UNRELATED>( /* ... */ );
```

For the intra-cell case only half of the block pairs have to be processed, because again it does not matter if block A with block B are processed of vice-versa:

Listing 6: intra-cell block processing (thread block cell processor)

```
for( int blockIndexA = 0 ; blockIndexA < numBlocksA ; blockIndexA++ ) {
  processBlock<BPT_SAME_BLOCK>( /* ... */ );

  for( int blockIndexB = 0 ; blockIndexB < blockIndexA; blockIndexB++ )
    processBlock<BPT_UNRELATED>( /* ... */ );
}
```

Likewise, the calculation of interactions inside the same block (ie intra-block) have to be treated as special case.

Since there are as many threads as molecules in a block, each thread can be assigned a fixed molecule of block A (in the outer loop). This molecule can be stored in registers or in local memory. The molecules of block B (of the inner loop) can be stored in shared memory, since they are only accessed by different threads of the same thread block.

A straight-forward algorithm would simply loop over all molecules in block B and each thread would calculate the interactions of its fixed molecule from block A with one molecule from block B. This is not the best solution however. Instead a block is divided into **warp blocks**. Each warp block contains 32 molecules (exactly the warp size). The aim here is not to improve cache usage, but to require less explicit synchronization. In the naive approach there would have to be a barrier after each interaction calculation to make sure that each thread is done with processing its molecule pair because another thread will access the molecule from block B in shared memory next. Threads in the same warp are synchronized by design, because they are executed in lock-step, so processing molecule interactions warp-block by warp-block improves parallelism. See 6 for an example. Again there is a special case when interactions of molecules of the same warp-block are calculated.

3.10 Warp Block Cell Processor

Another cell processor was added, because the performance of the thread block cell processor, which we have just looked at, does not scale well for sparsely populated cells.

The main problem is the sparse granularity of the thread block cell processor, which assigns the same amount of warps to each cell regardless of the number of molecules in it. The warp block cell processor fixes this.

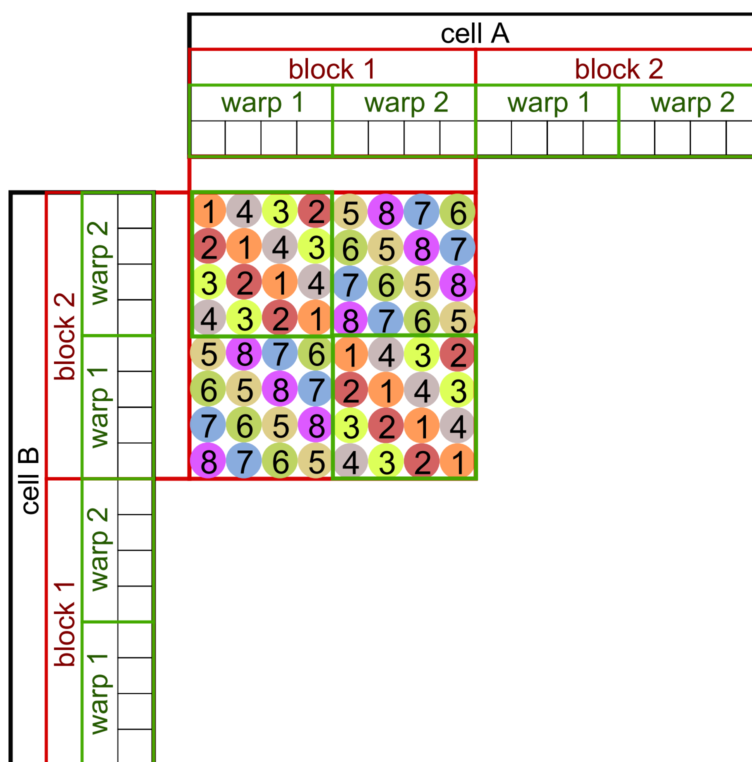


Figure 6: schematic of one block iteration (with a thread blocks of size 8 and warps of size 4, and all 8 parallel molecule pair iterations are shown for the block pair)

It works by using a dynamic scheduler and it processes cells on warp level. This means that a cell or cell pair is not processed by exactly one thread block, but by a varying number of warps inside possibly different thread blocks.

One thread block is running on each SM. A thread block can ask for jobs which are assigned to its different warps. A short jobs queue is used for every warp. Each warp processes a warp block inside the cell. For cell pairs the warp block inside one cell stays fixed while it iterates over all molecules of the other cell.

To synchronize all thread blocks and warps a global mutex is used. Idle warps try to acquire the lock and get new jobs. Multiple warps of the same thread block are joined together to reduce the amount of locking needed.

Because every warp calculates interactions on its own, storing back potential and virial values requires a per-cell lock to avoid race conditions.

There are two different implementations for calculating interactions. The simpler one calculates all interactions twice: between molecules in cells A and B and between molecules in cells B and A. The molecules of cell A are assigned to different warps and each warp iterates over all molecules in cell B and calculates the interactions and only stores them back into the molecules of cell A. When this is done, the cells are switched and now all interactions are calculated again and stored back into cell B. This avoids the need for additional locks and synchronization.

Listing 7: inter-cell block processing (warp block cell processor)

```
if( indexA >= warpBlockA.cell.endIndex ) {
  return;
}

StorageMolecule moleculeA;
moleculeA.init(indexA);

// now loop over all molecules in this cell
CellInfo cellB = warpBlockPairInfo.cellB;

for( int indexB = cellB.startIndex ; indexB < cellB.endIndex ; indexB++ )
    {
  StorageMolecule moleculeB;
  moleculeB.init(indexB);

  MoleculePairHandler::process( getThreadIndex(), moleculeA, moleculeB );
}

moleculeA.store();
```

The other implementation only calculates all interactions once and writes them to molecules from both cells. However, it needs additional locking because multiple warps might attempt to write into cell B at the same time. It uses shared memory to cache several molecules and iterate over them before writing them back. This reduces the amount of locking needed.

Listing 8: inter-cell block processing with shared memory cache(warp block cell processor)

```
StorageMolecule moleculeA;
moleculeA.init(indexA);

// now loop over all molecules in this cell
CellInfo cellB = warpBlockPairInfo.cellB;
for( int indexB = cellB.startIndex ; indexB < cellB.endIndex ; indexB +=
    WARP_SIZE ) {
  const uint threadIndex = getThreadIndex();
  resultLocalStorage.reset( threadIndex );

  if( indexA < cellA.endIndex ) {
    for( int shift = 0 ; shift < WARP_SIZE ; shift++ ) {
      const uint shiftedWarpThreadIdx = (warpThreadIdx + shift) %
          WARP_SIZE;
      const uint moleculeIndex = indexB + shiftedWarpThreadIdx;

      ResultStorageMolecule moleculeB;
      moleculeB.init( moleculeIndex );

      if( indexB + shiftedWarpThreadIdx < cellB.endIndex ) {
        MoleculePairHandler::process( threadIndex, moleculeA, moleculeB );

        const int targetIndex = warpIdx * WARP_SIZE + shiftedWarpThreadIdx
            ;
        moleculeB.store( targetIndex );
      }
    }
  }

  // store the processed molecules
  if( warpThreadIdx == 0 ) {
    cellLocks[cellIndexA].lock();
  }

  if( indexB + warpThreadIdx < cellB.endIndex ) {
    resultLocalStorage.commit( threadIndex, indexB + warpThreadIdx );
  }

  __threadfence();

  if( warpThreadIdx == 0 ) {
    cellLocks[cellIndexA].unlock();
  }
}

if( indexA < cellA.endIndex ) {
  moleculeA.store();
}
```

3.11 MarDyn Integration

The Cuda code is integrated into MarDyn by the `LinkedCellsCUDADecorator` class. It implements the `ParticleContainer` interface and decorates an internal `LinkedCells` object to calculate the molecule interactions using the GPU instead of the CPU. The main assumption of the code is that the cut-off radius is less than or equal to the size of a cell, so that only interactions between molecules of the same cell or between direct neighbors have to be calculated.

There are only few changes to MarDyn's code. The `MoleculeStorage` class is a friend of `Molecule`, a `setStats` method has been added to `ParticlePairs2PotForceAdapter` and additional accessor methods were needed in `LinkedCell`.

3.12 Debug and Testing Code

To verify that the Cuda implementation works correctly, there is code to run the CPU and Cuda interaction code in parallel and compare the calculated forces and torque. This option can be enabled by uncommenting `#define COMPARE_TO_CPU` in `cuda/config.h`.

Likewise there is code to disable the fast cell processor and use the reference implementation which is not parallelized at all and simple iterates over all molecules in the cells. It can be enabled by uncommenting `#define REFERENCE_IMPLEMENTATION` in `cude/config.h`.

To output the component descriptors as uploaded to the GPU, uncomment `#define DEBUG_COMPONENT_DESCRIPTORS` and to verify that the quaternion to matrix conversion works properly uncomment `#define TEST_QUATERNION_MATRIX_CONVERSION`.

3.13 `cuda/config.h`

The config header allows you to customize and choose different code paths. It is possible to switch between single-precision and double-precision calculations on the GPU and to configure the number of supported component sites such as the number of Lennard-Jones centers or dipoles.

4 Optimizations

4.1 Benchmarking Rigs

Two different systems were used:

- a workstation A with an Intel Core Duo E8500 (3.16 GHz) CPU, 4 GB RAM, and a Cuda 2.1 capable NVIDIA GTX 560 Ti graphics card with 1 GB video memory
- a workstation B with an Intel Xeon (2.53 GHz) CPU, 24 GB RAM, and an NVIDIA Tesla C2070 graphics card with 5 GB video memory

All charts are marked accordingly.

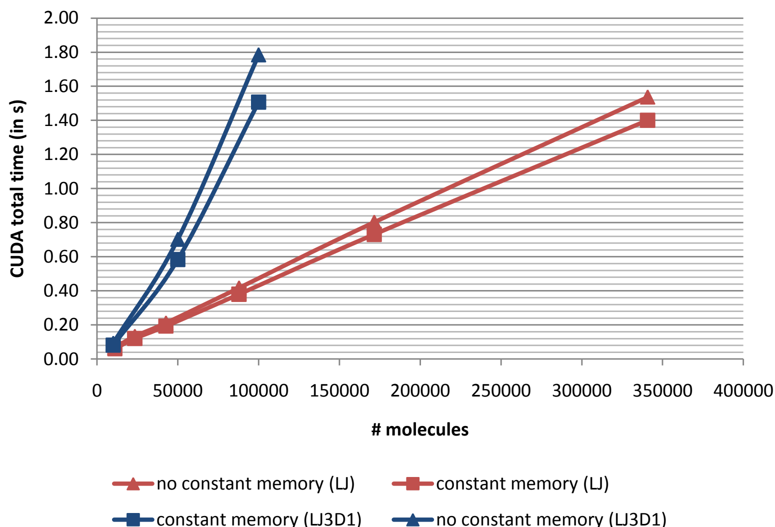


Chart 1: constant memory speed-up (on workstation A)

4.2 Using `__restricted__` Pointers and References

If all pointers and references in a function are marked as `__restricted__`, the compiler assumes that there is no aliasing between them, that is that all pointers and references point to different memory locations. Thus the compiler can better optimize the code and load more data directly into registers instead of having to use global memory accesses in sequence.

4.3 Molecule Proxies

The original code used a `Molecule` structure to collection molecule data early on before calling the MoleculePairHandler. This was done to be independent of the actual molecule storage structures. However, the `Molecule` structure used a lot of memory and this resulted in big stack frames. I rewrote it to use method accessor functions and act like a proxy to the molecule storage. It still wraps accesses and allows the molecule code to work without knowledge of the underlying storage but also uses less registers and local memory.

4.4 Constant Memory

Constant memory has its own 8 KB cache on each multiprocessor. Thus by moving constant data like the component descriptors, molecule interaction parameters and the storage buffer pointers into constant memory, global cache misses will be avoided and since this data is less than the size of the constant memory cache in total, it is likely that there will not be many constant memory cache misses. Using constant memory results in a speed-up between 7 and 20 percent. See Chart 1.

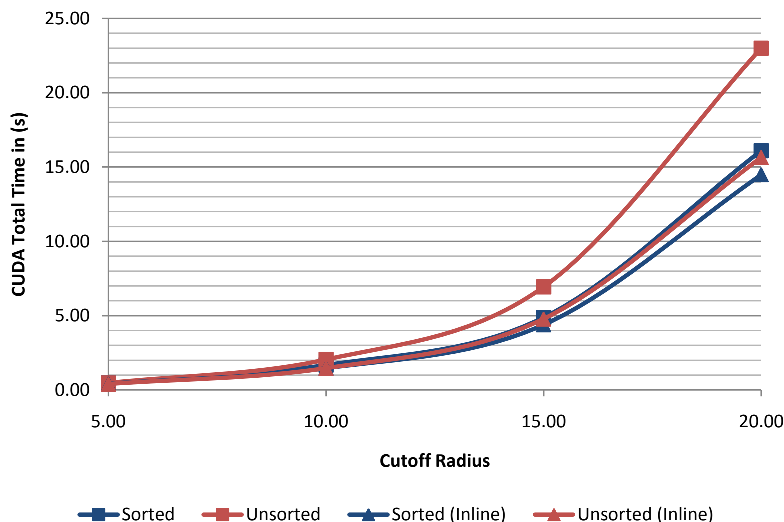


Chart 2: cells sorted by component type speed-up (on workstation A). inline means that the pair handler function is being inlined (it usually is)

4.5 Cells Sorted by Component Type

Cuda's performance decreases if threads inside the same warp take different branches because the SM has to serialize both code paths. A simple but efficient optimization is to sort the molecules inside of each cell before calculating the interactions. The current implementation simply loops over all molecules inside a cell once for each component type and writes them out in component type order. This is inefficient but total execution time still improves visibly. See Chart 2.

4.6 Without Effect: Non-Interleaved Molecule Data

The [NVI11a] makes a strong point about the use of coalesced memory access. For additional insight on this, see [Vol10]. The initial and currently used molecule storage code keeps the molecules stored in a semi-interleaved format:

Listing 9: Molecule Storage Declarations

```
__constant__ __device__ floatType3 *moleculePositions;
__constant__ __device__ Quaternion *moleculeQuaternions;
__constant__ __device__ Matrix3x3 *moleculeRotations;

__constant__ __device__ floatType3 *moleculeForces;
__constant__ __device__ floatType3 *moleculeTorque;

__constant__ __device__ ComponentType *moleculeComponentTypes;

__constant__ __device__ uint *cellStartIndices;
```

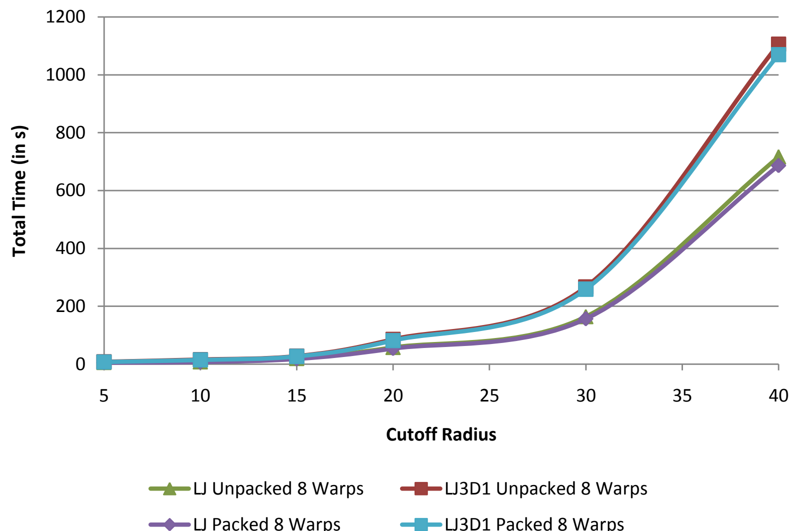


Chart 3: semi interleaved (packed) vs non-interleaved (unpacked) molecule storage (on workstation B)

The different fields (or streams) are stored non-interleaved but vector types are stored interleaved internally, that is the components are stored in sequence. I implemented fully non-interleaved version[5] and tested it.

It actually runs slower. See Chart 3[6]. My explanation for this is that the semi-interleaved version has better cache-locality. Even though each access on its own is not fully coalesced and has a stride, the next access can mostly use the same cache lines. The non-interleaved version does not have this benefit.

5 Benchmark Results

5.1 Cell Processor Comparison

See Figure 7 and Figure 8 for the results[7]. The specified warp count always refers to the used warps per thread block, that is how many threads each thread block runs ($32 \times$ warp count).

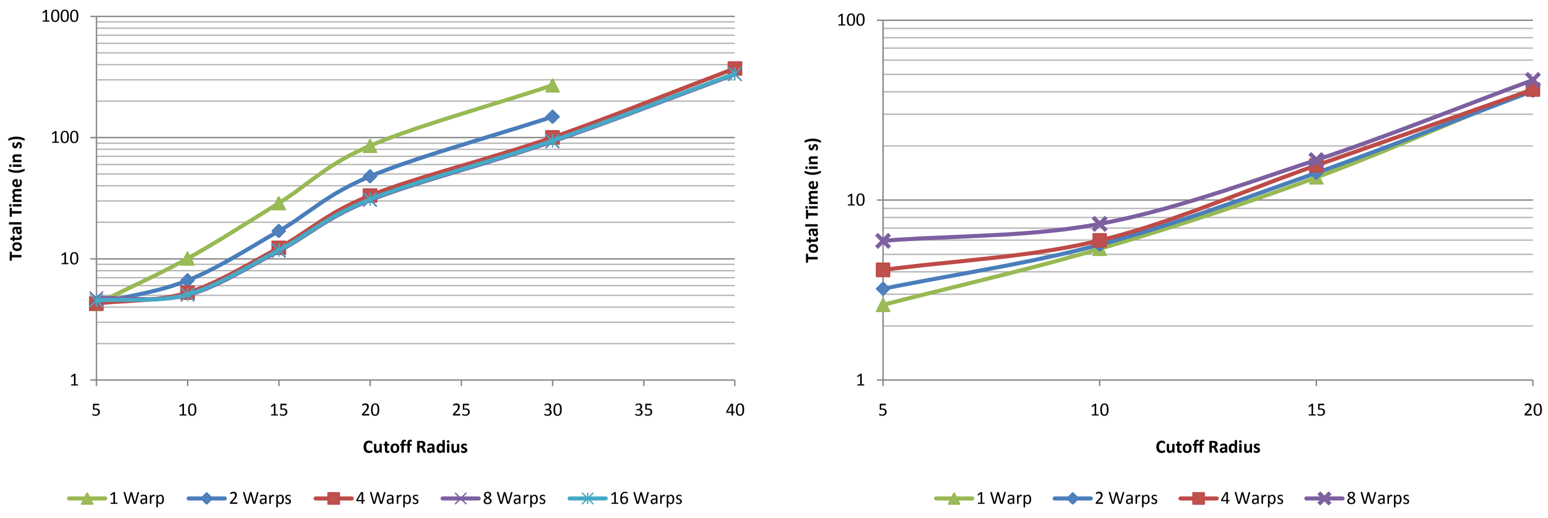
The `lj_80000` dataset consists of that many molecules with a single Lennard-Jones center. The `lj3d1_lj2d1_50000` dataset consists of a mix of molecules with 3 Lennard-Jones centers and 1 dipole and molecules with 2 Lennard-Jones centers and 1 dipole. The `lj3d1_50000` dataset only consists of molecule of the former kind.

The charts are plotted against varying cutoff radii. This means that the molecule

[5] every component of a vector, matrix or quaternion is stored in its own array

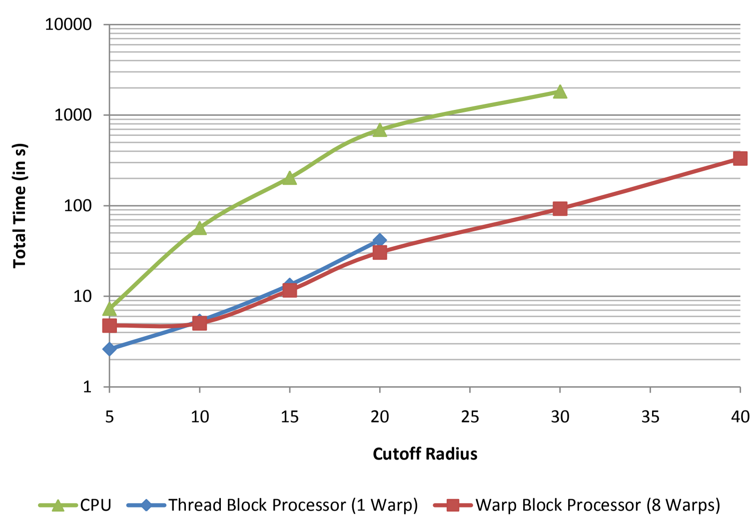
[6] build configuration: double precision, shared memory (`CONFIG_CUDA_DOUBLE_UNSORTED`)

[7] All benchmarks were run with double precision. The thread block cell processor was run with dedicated shared memory, the warp block cell processor without.



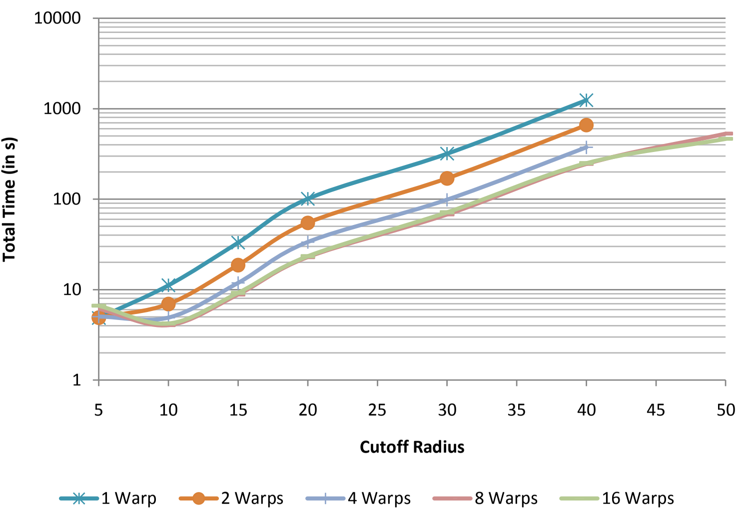
(a) warp block processor

(b) thread block processor

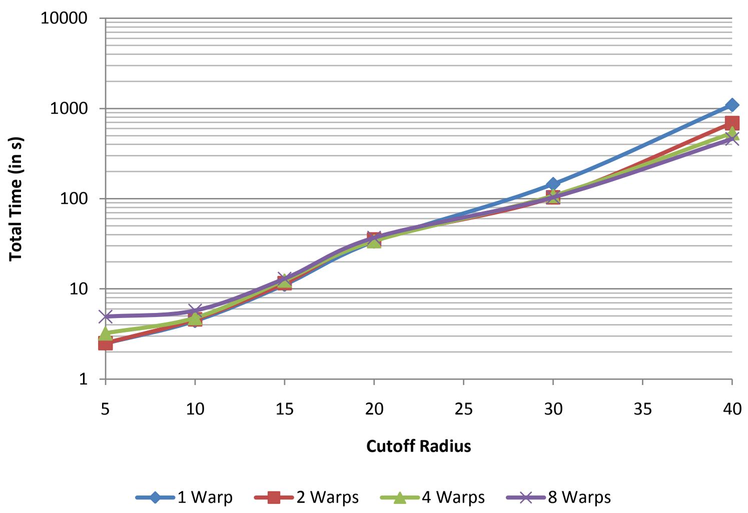


(c) summary chart

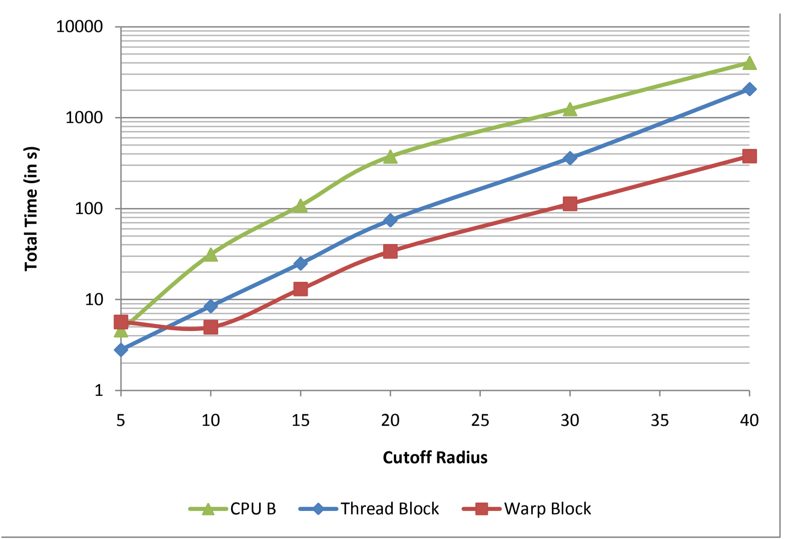
Figure 7: benchmark results for different cutoff radii on workstation A (dataset: `lj_80000`)



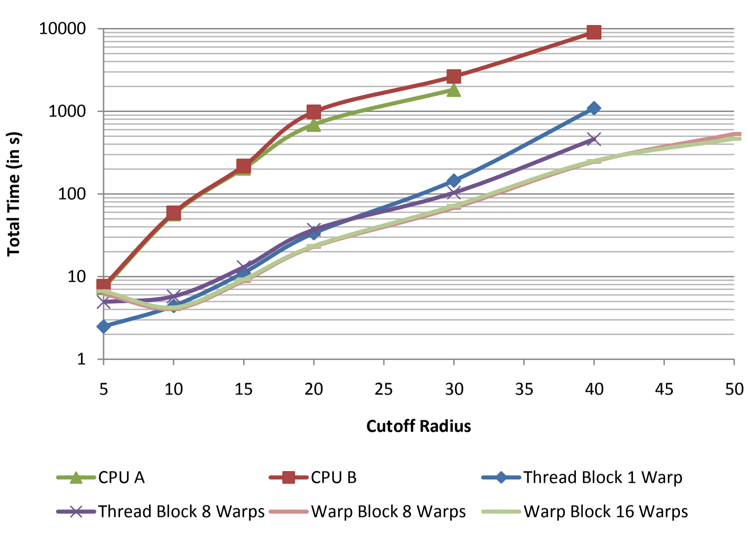
(a) warp block processor (dataset: `lj_80000`)



(b) thread block processor (dataset: `lj_80000`)

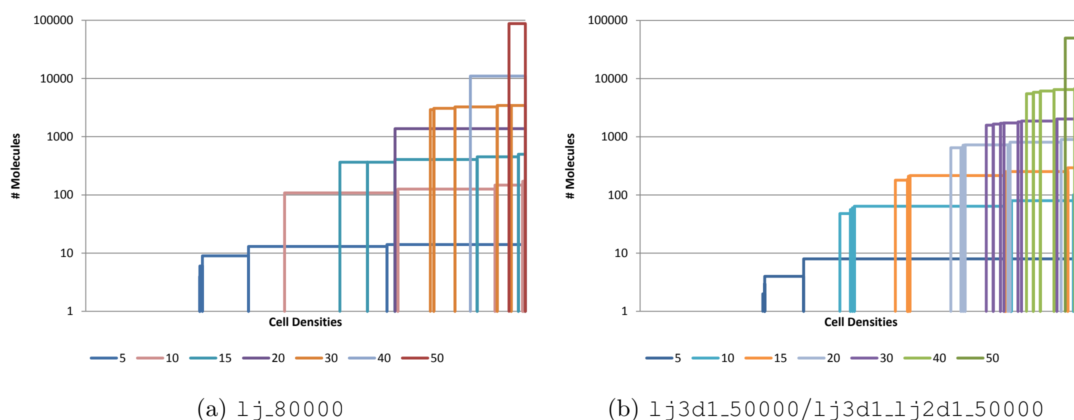


(c) summary chart for `lj3d1_lj2d1_50000`



(d) summary chart for `lj_80000`

Figure 8: benchmark results for different cutoff radii on workstation B



(a) `lj_80000` (b) `lj3d1_50000/lj3d1_lj2d1_50000`

Figure 9: cell molecule count histogram (for varying cutoff radii)

material density remains constant and only the varying cutoff radius is used to increase the number of molecules in the cells (the length of cell is exactly the cutoff radius).

Figure 9 shows the molecule counts per cell for the different datasets and different cutoff radii. The y-axis shows the molecule count. The width of the bars represents the ratio of the number of cells with a given molecule count to the total number of cells.

You should keep in mind that number of cells decreases with increasing cutoff radius: for a cutoff radius of 50 there are only 27 cells (1 inner cell and 26 halo cells), while for a cutoff radius of 5 there are many cells.

As you can see, the higher the cutoff radius, the higher the average number of molecules and the smaller the variance in the molecule counts per cell.

Thread Block Cell Processor Only for big cutoff radii the thread block cell processor starts scaling at all. For small radii $<= 20$ using only 1 warp per thread block always is fastest. Actually using many warps per thread block is significantly slower than just using 1 warp. The reason for this negative scaling is the low cell density for small cutoff radii. If many cells have less molecules than the warp size, even some threads of only one warp are already idling. Adding more warps just adds more idle threads that need to be scheduled. Since there is a limit for resident threads on an SM, this clogs the SMs. However, for big cutoff radii these threads will actually be put to use.

The thread block cell processor is inflexible in this regard because it always uses a fixed amount of warps per thread block for each cell or cell pair. Thus high warp counts only really work well for domains with homogeneously high molecule counts per cell, whereas a warp count of 1 works best for low molecule counts per cell.

Warp Block Cell Processor This cell processor scales as expected with increasing warp counts per thread block. However, for all but the highest cutoff radii, it runs faster with 8 warps than with 16. 16 warps win with very high molecule counts per cell though.

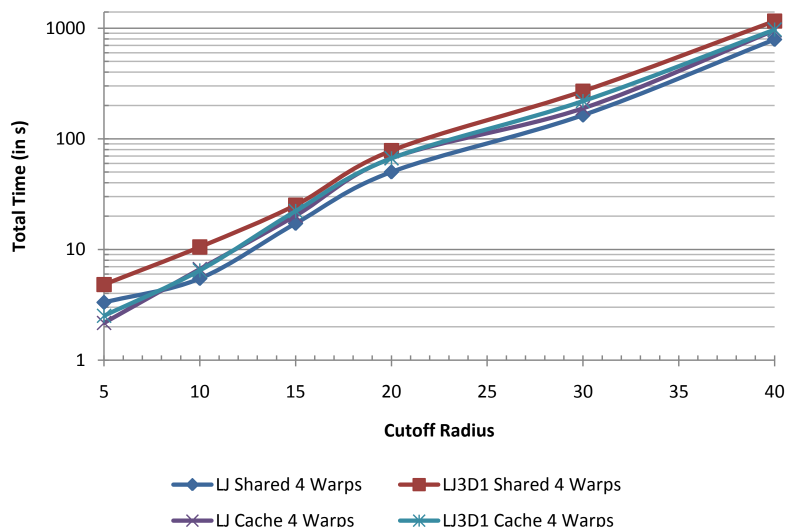


Chart 4: shared memory vs HW cache in the thread block block cell processor (on workstation B)

For low cutoff radii, the additional overhead in the scheduler negates the benefits of having additional warps available for processing.

Summary The thread block processor runs best for low to mid-range cutoff radii with only 1 warp. For high cutoff radii one can use more warps. Compared to the CPU the thread block cell processor always is faster—for mid-range cut off radii by one order of magnitude.

The warp block cell processor scales better. For low densities, it does not run that well compared to a thread block cell processor with only 1 warp, but already for a cutoff radius of 10 it runs faster than the thread block cell processor and it stays this way.

Compared to the CPU and the thread block cell processor this cell processor runs fastest and scales best except for very low molecule per cell counts.

5.2 Shared Memory versus HW Cache (Thread Block Cell Processor)

It is possible to reduce the shared memory size from 48 to 16 KB and increase the available L1 cache per SM to 48 KB.

The thread block processor normally uses shared memory to cache molecule data that is being used. I have tried using a bigger L1 cache and disabling the shared memory storage. The results are mixed as can be seen in Chart 4.

Using the bigger L1 cache is faster for low cutoff radii. As the molecule count per cell increases the results become mixed: using the cache is faster for the simple `lj_80000` dataset but slower for the more complex `lj3d1_50000` or `lj3d1_lj2d1_50000` datasets.

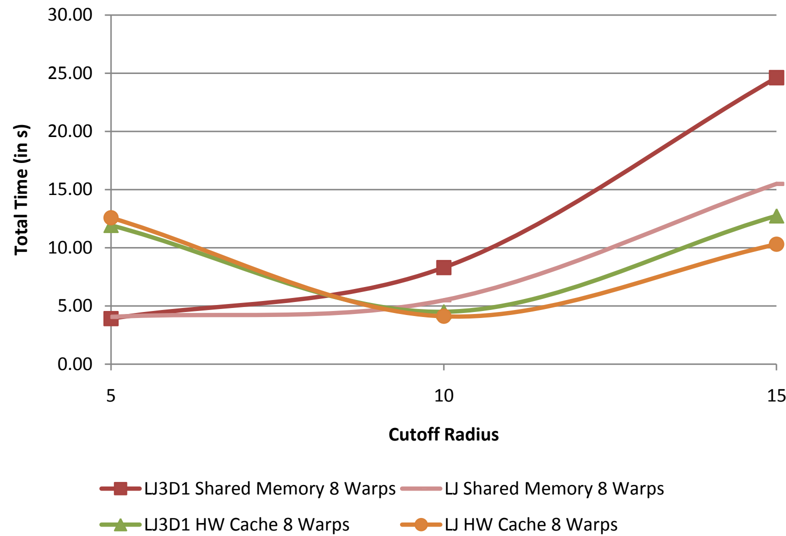


Chart 5: shared memory vs HW cache in the warp block cell processor (workstation B)

5.3 Shared Memory versus HW Cache (Warp Block Cell Processor)

As with the thread block cell processor, there are two modes in the warp block cell processor. One uses shared memory to cache the calculated results and the other only uses an enlarged L1 cache. As you can see in Chart 5 there are very mixed results, too. Only results for small cutoff radii are shown because performance for the shared memory version degrades very much with higher cutoff radii.

For the warp block cell processor, everything is inverse: shared memory performs better than the HW cache for the smallest cutoff radius and worse for bigger cutoff radii.

The problem is that the additional synchronization due to locking outweighs any benefit of shared memory. When the cutoff radius is 5, only one warp is needed per cell, so no locking occurs. For bigger cutoff radii locking has a huge impact on performance. The current implementation worsens this because it tries to execute all warp blocks of a cell at once, so different warps processing the cell will almost always have to wait for each other.

5.4 Single Precision Floating Point Mode

All the observations hold for single precision mode. Using single instead of double precision doubles the speed as you can see in Chart 6.

5.5 Local and Global Error

See Figure 10 for charts that show the local and global errors. GPU potential and virial values are compared to CPU ones and show the global error, whereas the RMS values

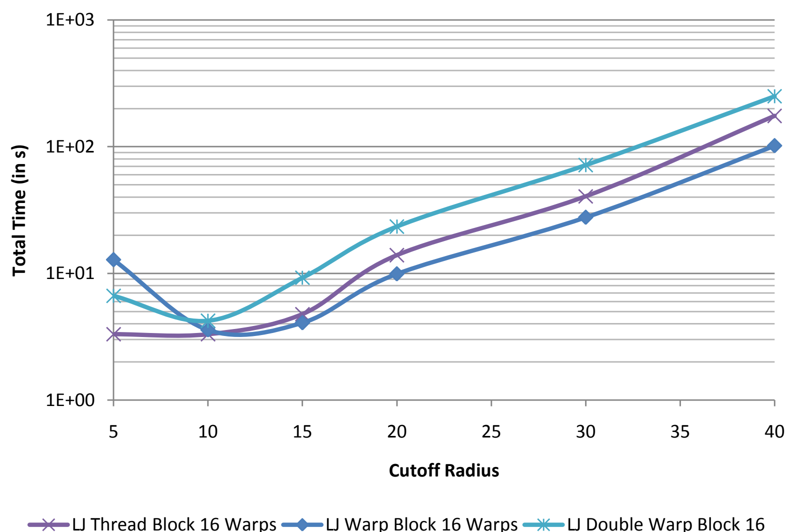


Chart 6: single precision floating point results

show the local error between CPU and GPU calculations.

The error chart for the simple `lj_80000` dataset looks very good, but the global error is off for the more complex datasets. For one the GPU only has a precision of 64 bits, while the FPU uses 80 bits internally. Secondly, the dipole force and torque calculation code on the GPU is not the same as the one on the CPU in contrast to the Lennard-Jones potential code which is exactly the same. This is due to me optimizing the code to use less variables and registers.

6 Further Work

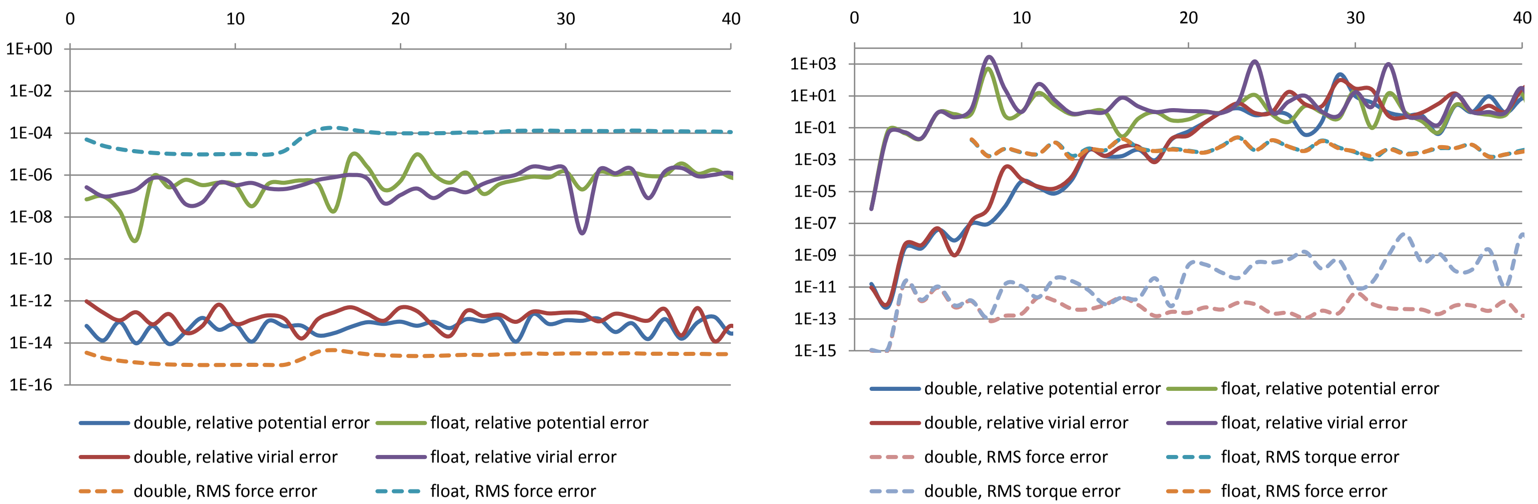
There are many areas where further work is possible:

6.1 Molecule Storage

I have only compared fully non-interleaved storage with the current one. It seems that cache hits are more important than fully coalesced accesses, so it could well be possible that storing the molecule data fully interleaved could outperform the current implementation.

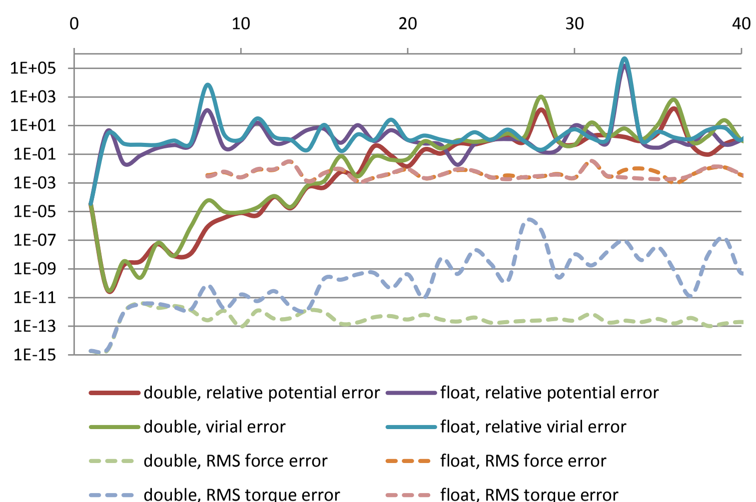
6.2 Cell Processor

The warp block cell processor is better than the thread block cell processor for all but the most basic case. For this simplest case, which is a cutoff radius of 5 in my datasets or, more generally, a molecule count per cell smaller or equal to the warp size, the simplest



(a) `lj_80000`

(b) `lj3d1_50000`



(c) `lj3d1_lj2d1_50000`

Figure 10: local and global error measurements on workstation B

possible implementation is the most effective: a simple loop over all molecule pairs. This is exactly what the thread block cell processor with only 1 warp degrades to.

Even though using a shared memory cache in the warp block cell processor speeds it up, the additional locks in the code erase all benefits.

A possible remedy would be using the cache while still calculating all interactions twice. This would remove the need for any additional locks.

Another idea would be to dynamically choose the cell processor depending on the maximum molecule count per cell.

6.3 Halo Handling and Global Domain Statistics

Support for domain decomposition has not been implemented. The code assumes that the halo cells are simply copies of the border cells and uses this assumption when calculating the global potential and virial values.

It does not implement an `isFirstParticle`-check, because the additional branching in the molecule pair handler is inefficient.

I would rather propose to change the halo global statistics handling to a cell-based approach, ie it should be possible to decide on the cell level whether to take the potential and virial into account or not. This is possible by eg simply extending 3 of the 6 planes of the domain cuboid and using them to split the halo cells accordingly into "owned" and "ghost" cells. This would add the necessity of calculating molecule interactions between some halo cells, but the design could be quite clean otherwise. Especially potential and virial calculation would be very simple: the potential and virial values of ghost halo cells would have to be subtracted of the total values of the inner and border cells, and those of owned halo cells added.

6.4 Domain Traversal

Especially considering the possible changes to the halo handling, the current domain traversal code needs to be changed, too. Its main idea is the construction of a virtual coordinate system for the cell grid, that can be mapped effectively to real cell indices. In this virtual coordinate system all cells or rather cell pairs can be visited in any order and there is no overlap between any cell pairs.

The current code processes more cell pairs than necessary compared to the CPU code. The best solution for this would be to rewrite the domain traverser to output an array of cell pairs similar to what [Ore10]. It could take into account empty cells and reduce the number of distinct kernel calls with the added information.

6.5 Less Synchronization/Fewer Locks

The spin-locks are the only reason the warp block cell processor is a lot slower when run in the shared memory mode compared to the bigger L1 cache. Finer lock granularity could really help avoid unnecessary waiting between warps.

If the domain traversal code was changed to output a cell pair array, one could directly make it create a warp block pair interaction array between all warp blocks of a cell. The

scheduler could then be reduced to an atomic increment on a global warp block pair counter.

Creating this cell/warp block pair array could be done on the GPU, too, given the information already available.

6.6 Memory Usage

My code uses way too little memory. The footprint is very small compared to the amount of available video RAM in today's workstation graphics cards. Thus finding a design that uses lots of scratch memory could well lead to a faster implementation.

From benchmarking and profiling I suspect that my code is instruction limited instead of memory limited because most changes regarding memory access patterns had very little effect. The alternative design consists of having a scratch buffer for every warp/thread and reducing all the interactions later on.

Maybe it could even be possible to sort the interactions by magnitude to reduce them in a way that minimizes the floating-point error.

The current code also does not make use of pinned or mapped memory. The amount of time spent copying data around is negligible for the datasets I have used, but this could change.

6.7 Cuda Interface

I have intentionally used the driver API, because I prefer its cleaner code separation between GPU and CPU code. Moreover, it could be possible to write code that dumps all parameters and calls to the API to replay kernel calls on different systems for debugging[8]. I have not had time to write this.

There is a good reason to switch back to the runtime CUDA API: most third-party tools support only it. Ocelot[9] is a good example and certainly a reason to switch back.

I think that switching back to the runtime API is less work than the other way around because the code is cleanly separated.

6.8 Quaternions

Currently quaternions are converted to 3×3 matrices and these matrices are used for transformations. I have not researched whether rotating points directly by a quaternion might not be faster on the GPU.

6.9 Warp-Centered Design

[BCK11] and [WPSAM10] explain that code divergence on a warp-level has no negative effect but rather can be used for performance optimizations. You could come up with a design that makes use of this even more than the warp block cell processor.

[8] a prime example would be a Windows workstation with NVIDIA's Parallel Nsight—`http://developer.nvidia.com/nvidia-parallel-nsight`

[9] `http://code.google.com/p/gpuocelot/`

6.10 More Stages

I have encountered a compiler bug that was caused by the size or rather depth of some nested loops inside various inlined functions. I doubt that it is possible to reduce the used register count (without sacrificing performance), but the register count is currently the reason that only 512 threads can be resident at any time in an SM, which is only a third of the maximum number.

Maybe it could be possible to split the current molecule pair handler into multiple smaller kernels that can run independently and use less registers each to achieve better parallelization.

6.11 Low Molecule Counts Per Cell

An open question for me is how to deal with sparsely populated cells effectively. One warp would have to process multiple cells to keep all threads in a warp busy.

6.12 Single Precision Mode

The current code is only optimized for double precision. The current Cuda version does not support atomic operations or reductions for doubles like it does for floats. One could optimize the code for float support and make use of reductions or atomic operations to avoid synchronization and simplify the code.

7 Summary

This paper describes the architecture of a Cuda kernel that supports different molecule potential types and explains optimizations and implementation details. We took a hard look at different approaches for parallelizing molecule interaction calculations and compared them.

It is evident that the warp block cell processor is most effective for higher molecule counts per cell and the simplest solution works best for low molecule counts per cell.

Lots of further optimizations are possible and the Cuda implementation does not yet support all features of MarDyn like domain decomposition and halo cell handling.

References

[BCK11] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. Presented at the International Conference on Super Computing 2011, 2011. Available from: `www.stanford.edu/~mebauer/pdfs/cudadma-sc11.pdf`.

[Buc10] Martin Buchholz. *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen.* Dissertation, Institut für Informatik, Technische Universität München, München, August

2010. Available from: http://www5.in.tum.de/pub/buchholz_dissertation10.pdf.

[Fow06] Martin Fowler. Internaldslstyle [online]. 2006. Available from: http://martinfowler.com/bliki/InternalDslStyle.html.

[NVI11a] NVIDIA Corporation. *NVIDIA CUDA C Best Practices Guide*, 2011. Version 4.0. Available from: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf.

[NVI11b] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, 2011. Version 4.0. Available from: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[Ore10] Kai Orend. Numerische Aspekte bei Molekulardynamik-Simulationen - Beschleunigung durch Graphikkarten. Studienarbeit/sep/idp, Institut für Informatik, Technische Universität München, September 2010. Available from: http://www5.in.tum.de/pub/orend2010.pdf.

[Vol10] V. Volkov. Better performance at lower occupancy. Presented at the GPU Technology Conference 2010 (GTC 2010), 2010. Available from: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[WPSAM10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246. IEEE Computer Society, 2010. Available from: http://www.stuffedcow.net/research/cudabmk.