NCHC Open Hackathon

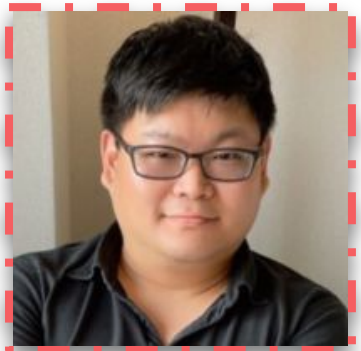
Dream Chaser Final

2024/12/04

Dream Chaser

Leader

Gao Quan-Ze
高銓澤
CGU 醫學



Member 1

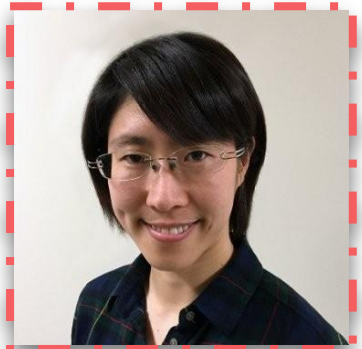
Cheng-Yu Ma
馬誠佑
CGU 人工智慧

Member 2

Chi-Ching Lee
李季青
CGU 資訊工程

Mentor 1

Anthony Chang
張安政
NVIDIA DevTech



Mentor 2

Ying-Ja Chen
陳映嘉
NVIDIA Solution Architect

OmegaDocker

Introduction

- OmegaDocker is an oligomer-supported and gpu-accelerated docking program under LGPL.
- The first (current) version of OmegaDocker is based on AutoDock-GPU v1.5.4 developed by the Forli lab at Scripps Research.
- AutoDock-GPU v1.5.4 is the GPU version of AutoDock v4.2.6.

Algorithmic motif

Genetic algorithm (GA), Adadelta

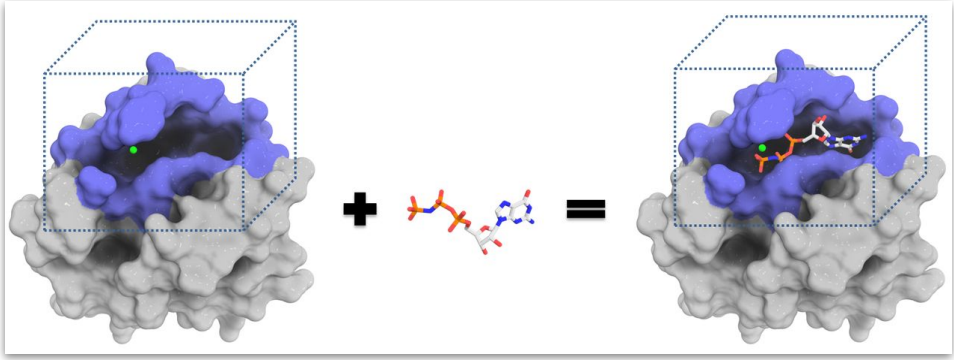
Libraries

CUDA, GMP

Programing Language

C/C++

Docking

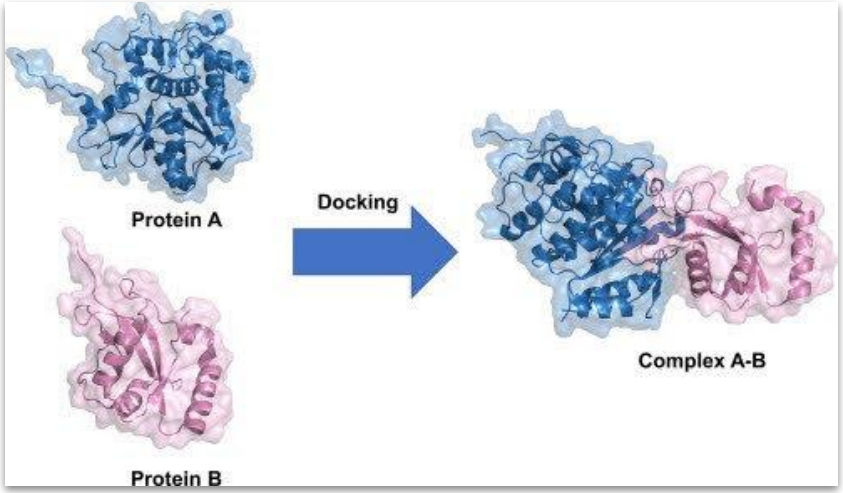


Due to the computation complexity, only the small compound is flexible.

DockThor
GMMSB
https://dockthor.lncc.br/v2/

But the demand for docking with miRNAs or aptamers is increasing.

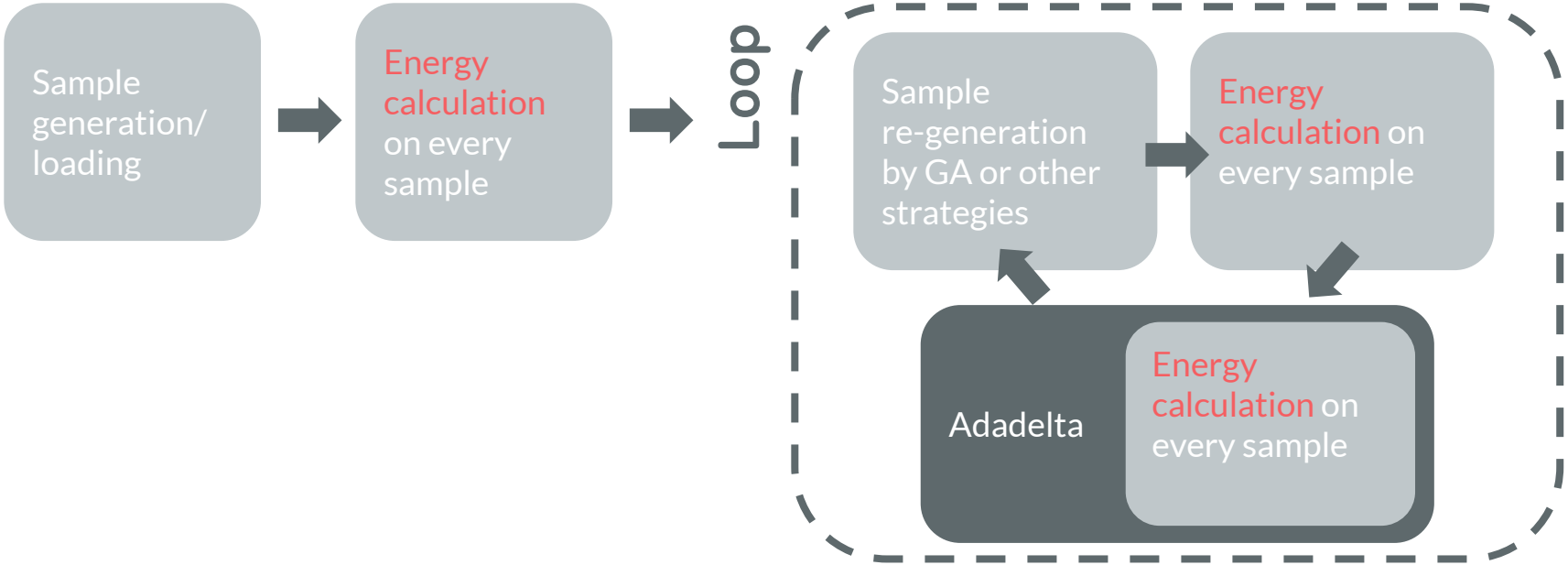
Computational Methods in Drug Discovery and Development
Sep. 2024, Sadettin Yavuz Ugurlu
https://chemrxiv.org/engage/chemrxiv/article-details/66f4016a12ff75c3a16b28c2



OmegaDocker

The most time-consuming function

Energy calculation function



Initial Goal

Acceleration test of tensor core applying

Accelerating a Molecular Docking Application by Leveraging Modern Heterogeneous Computing Systems

GABIN SCHIEFFER

Degree Programme in Computer Science and Engineering
Date: April 16, 2023

Supervisor: Ivy B. Peng
Examiner: Stefano Markidis
School of Electrical Engineering and Computer Science
Swedish title: Accelerering av en Molekylär Dockningsapplikation genom att Utnyttja Moderna Heterogena Datorsystem

Accelerating Drug Discovery in AutoDock-GPU with Tensor Cores

Gabin Schieffer and Ivy Peng(✉)

KTH Royal Institute of Technology, Stockholm, Sweden
{gabins, ivybopeng}@kth.se

Abstract. In drug discovery, molecular docking aims at characterizing the binding of a drug-like molecule to a macromolecule. AutoDock-GPU, a state-of-the-art docking software, estimates the geometrical conformation of a docked ligand-protein complex by minimizing a scoring function. Our profiling results indicate that the current reduction operation that is heavily used in the scoring function is sub-optimal. Thus, we developed a method to accelerate the sum reduction of four-element vectors using matrix operations on NVIDIA Tensor Cores. We integrated the new reduction operation into AutoDock-GPU and evaluated it on multiple chemical complexes on three GPUs. Our results show that our method for reduction operation is 4-7 times faster than the AutoDock-GPU baseline. We also evaluated the impact of our method on the overall simulation time in the real-world docking simulation and achieved a 27% improvement on the average docking time.

Keywords: Molecular docking · AutoDock · GPU · Tensor Core · Drug Discovery

1 Introduction

The pharmacological effect of a drug is generally induced by the binding of a drug molecule to a specific protein target. Thus, characterizing the ability of binding is crucial for drug discovery. Once a target for a disease is identified, tens of millions of chemical compounds, or *ligands*, will go through high-throughput screening. For such vast search space, virtual screening that leverages computational approaches is becoming increasingly important for accelerating the process and reducing the high cost required in experimental screenings [12,5]. In particular, structure-based virtual screening software uses molecular docking tools to test a molecule drug candidate for binding a protein target (receptor). In recent COVID-19 research, high-performance virtual screening software has been used in combating the pandemic [5].

A typical molecular docking job consists of evaluating a large number of ligands, each as an independent docking task. Further distributing individual docking tasks onto high-performance computing (HPC) systems, with multi-core CPU or GPUs, can significantly accelerate docking, e.g., AutoDock-GPU reports 350-fold speedup over single-threaded implementation [8,12]. AutoDock is widely

https://arxiv.org/abs/2410.10447

Initial Goal

Acceleration test of tensor core applying

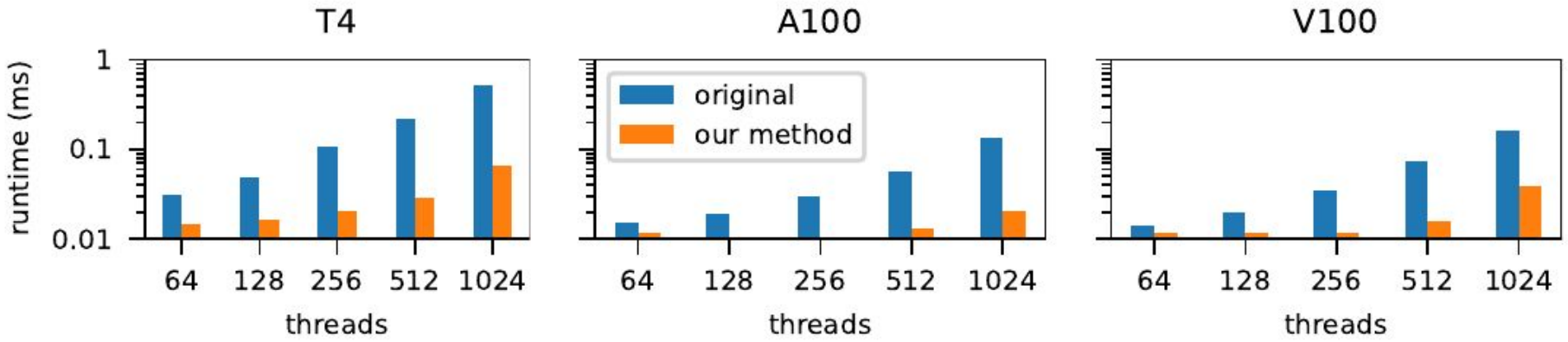
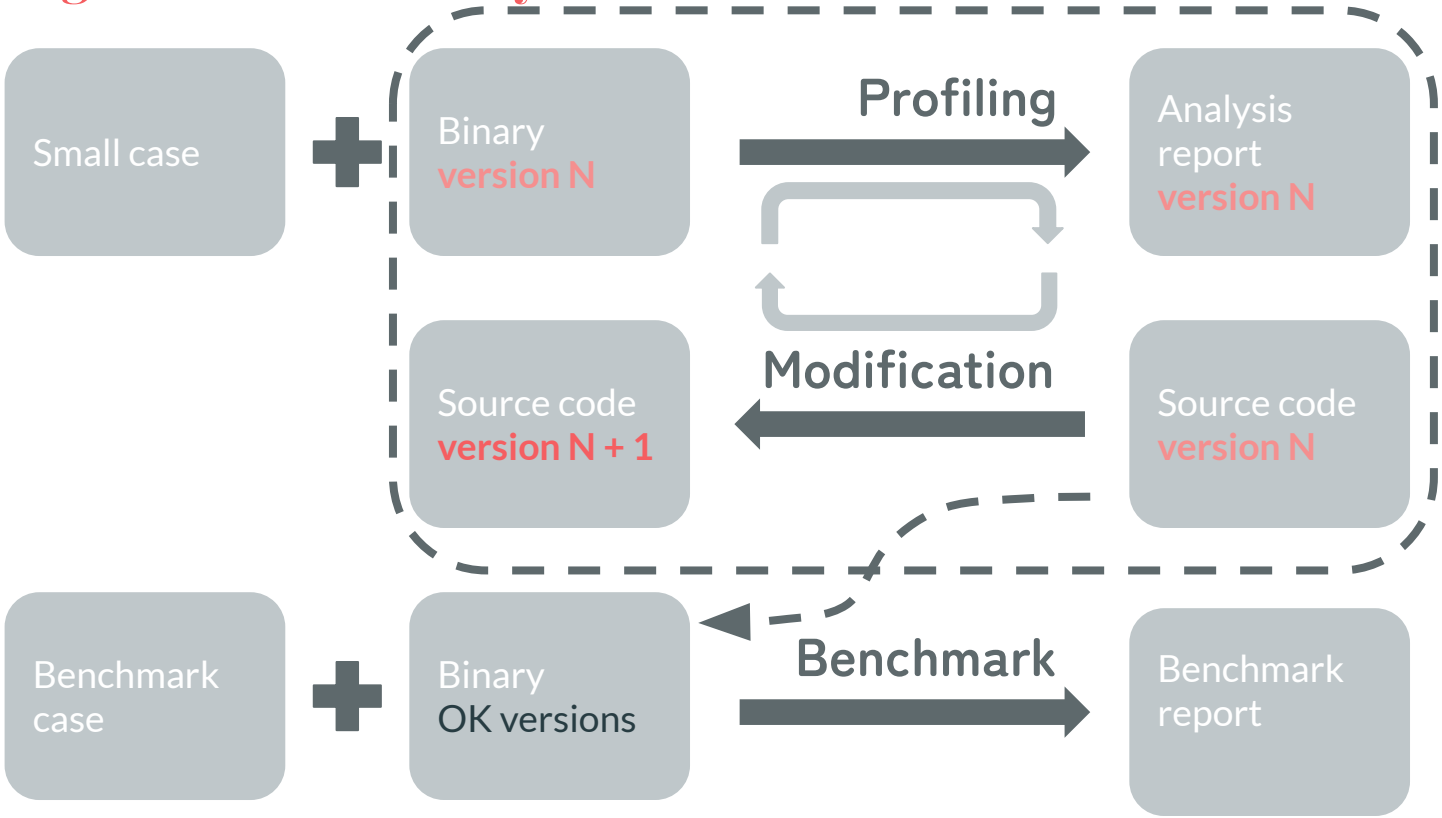


Fig. 5: Average runtime of the two versions of the test reduction kernel on three generations of NVIDIA GPUs: T4, A100, and V100.

Actual Strategy

Profiling – Modification Cycle



Overall Improvement

Overall Improvement

7.79

Size of blockDim

1.38

Algorithm modification

1.18

Conditional statements

1.02

printf() in kernel

1.01

Prefetch

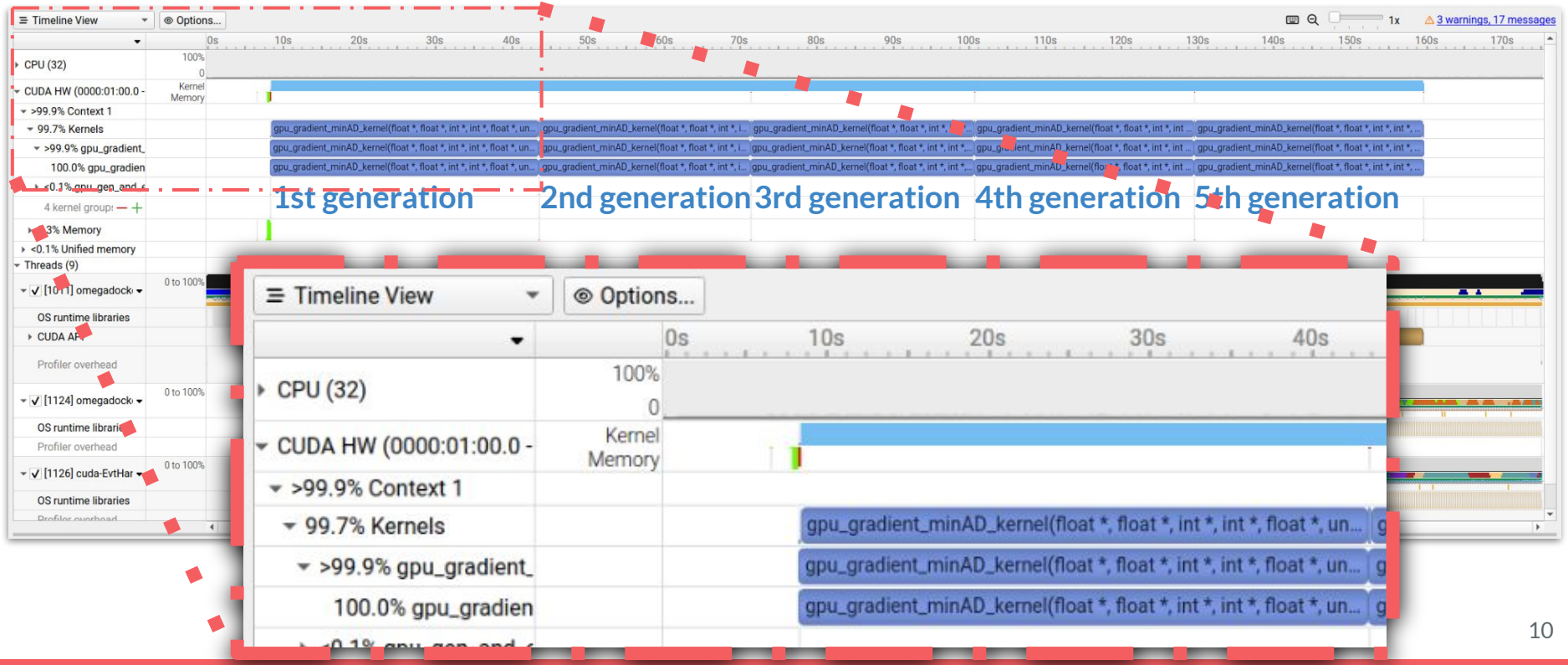
N/A

Remove unnecessary atomic operations

4.68

Profiling of OmegaDocker

Nsight Systems 2024.5.1



Profiling of OmegaDocker

Nsight Compute 2024.5.1

Run with root permission

Welcome × base_NsightComputeReport2.ncu-rep ×

	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	728 - gpu_gradient_minAD_kernel	(160, 1, 1)x(128, 1, 1)	15.83 s	22,323,384,802	1 - NVIDIA RTX A6000	1.41 Ghz	[4084] omegadocker_gpu_128wi	

Summary | Details | Source | Context | Comments | Raw | Session

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

ID	Estimated Speedup	Function Name	Demangled Name	Duration (2.92062e+10)	Runtime Improvement (2.43521e+10)	Compute Throughput	Memory Throughput	# Registers	Grid Size	Block Size
0	83.15	gpu_gradient_min...	gpu_gradient_min...	15.83	13.16	1.86	5.35	64	160, 1, ...	128, 1, ...
1	83.66	gpu_gradient_minA...	gpu_gradient_minA...	13.37	11.19	2.38	6.87	64	160, 1, ...	128, 1, ...

Profiling of OmegaDocker

Nsight Compute 2024.5.1

Run with root permission

Welcome ✕ | base_NsightComputeReport2.ncu-rep ✕

	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	728 - gpu_gradient_minAD_kernel	(160, 1, 1)x(128, 1, 1)	15.83 s	22,323,384,802	1 - NVIDIA RTX A6000	1.41 Ghz	[4084] omegadocker_gpu_128wi	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

The report contains imported source files.

GPU Speed Of Light Throughput — GPU Throughput Chart

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	1.86	Duration [s]	15.83
Memory Throughput [%]	5.35	Elapsed Cycles [cycle]	22,323,384,802
L1/TEX Cache Throughput [%]	37.93	SM Active Cycles [cycle]	1,704,260,722.87
L2 Cache Throughput [%]	5.35	SM Frequency [Ghz]	1.41
DRAM Throughput [%]	0.38	DRAM Frequency [Ghz]	7.59

Small Grid This kernel grid is too small to fill the available resources on this device, resulting in only 0.2 full waves across all SMs. Look at Launch Statistics for more details.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved close to 0% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

PM Sampling

Timeline view of PM metrics sampled periodically over the workload duration. Data is collected across multiple passes. Use this section to understand how workload behavior changes over its runtime.

Maximum Sampling Interval [ms]	131.07	# Pass Groups	2
Maximum Buffer Size [Mbytes]	39.69	Dropped Samples [sample]	0

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	0.03	SM Busy [%]	12.20
Executed Ipc Active [inst/cycle]	0.39	Issue Slots Busy [%]	9.64
Issued Ipc Active [inst/cycle]	0.39		

Low Utilization
Est. Local Speedup: 87.80% All compute pipelines are under-utilized. Either this workload is very small or it doesn't issue enough warps per scheduler. Check the Launch Statistics and Scheduler Statistics sections for further details.

Memory Workload Analysis — Memory Chart

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/s]	2.74	Mem Busy [%]	5.07
L1/TEX Hit Rate [%]	43.08	Max Bandwidth [%]	5.35
L2 Hit Rate [%]	97.45	Mem Pipes Busy [%]	1.86
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L1TEX Global Load Access Pattern
Est. Speedup: 3.89% The memory access pattern for global loads from L1TEX might not be optimal. On average, only 10.5 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the Source Counters section for uncoalesced global loads.

L1TEX Global Store Access Pattern
Est. Speedup: 4.56% The memory access pattern for global stores to L1TEX might not be optimal. On average, only 6.8 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the Source Counters section for uncoalesced global stores.

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	1.36	No Eligible [%]	90.33
Eligible Warps Per Scheduler [warp]	0.10	One or More Eligible [%]	9.67
Issued Warp Per Scheduler	0.10		

Issue Slot Utilization
Est. Local Speedup: 90.33% Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 10.3 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 12 warps per scheduler, this kernel allocates an average of 1.36 active warps per scheduler, but only an average of 0.10 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, reduce the time the active warps are stalled by inspecting the top stall reasons on the Warp State Statistics and Source Counters sections.

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	14.02	Avg. Active Threads Per Warp	29.12
Warp Cycles Per Executed Instruction [cycle]	14.04	Avg. Not Predicated Off Threads Per Warp	27.70

Long Scoreboard Stalls
Est. Speedup: 60.41% On average, each warp of this kernel spends 8.5 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. Find the instruction producing the data being waited upon to identify the culprit. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality (coalescing), or by changing the cache configuration. Consider moving frequently used data to shared memory. This stall type represents

Potential Performance Bottlenecks

Size of blockDim

- Original: 128 (Original source code up limit: 256)
- Tested: 256, 512, 1024 (Source code modified)

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active war
Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to h
achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	66.67
Theoretical Active Warps per SM [warp]	32
Achieved Occupancy [%]	11.24
Achieved Active Warps Per SM [warp]	5.39

ctive war
ility to h

66.67
32
65.52
31.45

Performance ratio

- 1.38x

Potential Performance Bottlenecks

printf() in kernel

- Original: For an important function in other docking stage
- Modified: Create a new kernel for it.

```
if(threadIdx.x == 0 && blockIdx.x == 0)
{
  printf("COOR1 %d\n", iteration_cnt);
  for(uint i = 0; i < cData.dockpars.num_of_atoms; i++)
  {
    // printf("ATOM1  %12u %12.6f %12.6f %12.6f\n", i, calc_coords[i].x, calc_coords[i].y, calc_
    printf("ATOM1  %12u %12.6f %12.6f %12.6f\n", i, cData.pKerconst_conform_ref_coords_const[3*
  }
  printf("DONE1 %d\n", iteration_cnt);
}
__syncthreads();
```

Performance ratio

- 1.01x

Potential Performance Bottlenecks

Prefetch

- Original: Many global variables IO

```
 // sans =>
uint32_t atom1_id = cData.pKerconst_intracontrib_intraE_contributors_const[2*contributor_counte   6.46%   2.19%   Global   Load
uint32_t atom2_id = cData.pKerconst_intracontrib_intraE_contributors_const[2*contributor_counte   3.25%   1.65%   Global   Load
// <== sans

// Calculating vector components of vector going
// from first atom's to second atom's coordinates
float subx = calc_coords[atom1_id].x - calc_coords[atom2_id].x;   4.44%   2.74%   Global(2)   Load(2)
float suby = calc_coords[atom1_id].y - calc_coords[atom2_id].y;   4.27%   1.65%   Global(2)   Load(2)
float subz = calc_coords[atom1_id].z - calc_coords[atom2_id].z;   3.30%   1.65%   Global(2)   Load(2)

// Calculating atomic_distance
float dist = sqrt(subx*subx + suby*suby + subz*subz);   0.11%   2.74%
float atomic_distance = dist * cData.dockpars.grid_spacing;   0.08%   0.55%
```

Potential Performance Bottlenecks

Prefetch

NVIDIA DEVELOPER Home Blog Forums Docs Downloads Training

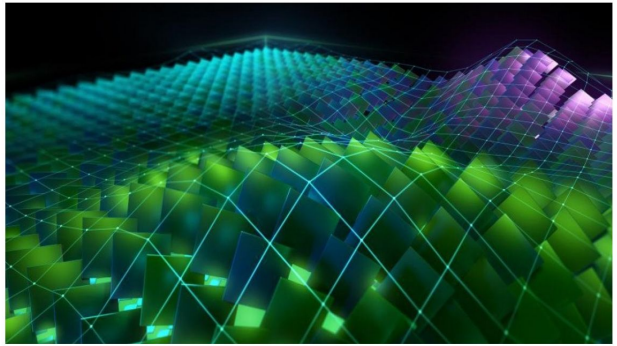
Technical Blog Search blog Filter

Simulation / Modeling / Design English

Boosting Application Performance with GPU Memory Prefetching

Mar 23, 2022 +10 Like Discuss (7)

By Rob Van der Wijngaart and Fred Oh



NVIDIA DEVELOPER Home Blog Forums Docs Downloads Training

Technical Blog Search blog Filter

A synchronization within the loop—for example, `syncthreads`—constitutes a memory fence and forces the loading of `arr` to complete at that point within the same iteration, not PDIST iterations later. The fix is to use asynchronous loads into shared memory, the simplest version of which is explained in the Pipeline interface section of the CUDA programmer guide. These asynchronous loads do not need to complete at a synchronization point, but only when they are explicitly waited on.

Here's the corresponding code:

```
#include <cuda_pipeline_primitives.h>
__shared__ double v[PDIST* BLOCKDIMX];
for (k=0; k<PDIST; ++k) { // fill the prefetch buffer asynchronously
  __pipeline_memcpy_async(&vsmem(k), &arr[threadIdx.x+k* BLOCKDIMX], 8);
  __pipeline_commit();
}
for (i=threadIdx.x, ctr=0; i<imax; i+= BLOCKDIMX, ctr++) {
  __pipeline_wait_prior(PDIST-1); //wait on needed prefetch value
  ctr_mod= ctr%PDIST;
  locvar = vsmem(ctr_mod);
  if ( i<imax-PDIST* BLOCKDIMX) { // prefetch one new value
    __pipeline_memcpy_async(&vsmem(ctr_mod), &arr[i+PDIST* BLOCKDIMX], 8);
    __pipeline_commit();
  }
  <more instructions using locvar, for example, transcendentals>
}
```

As each `__pipeline_wait_prior` instruction must be matched by a `__pipeline_commit` instruction, we put the latter inside the loop that prefills the prefetch buffer, before entering the main computational loop, to keep bookkeeping of matching instruction pairs simple.

Performance results

Figure 1 shows, for various prefetch distances, the performance improvement of a kernel taken from a financial

Performance ratio

- N/A

Potential Performance Bottlenecks

Conditional statements

```
bool notDoubleH = !(
                    atom1_type_vdw_hb == 0 ||
                    atom1_type_vdw_hb == 1 ||
                    atom1_type_vdw_hb == 2 ||
                    atom1_type_vdw_hb == ATYPE_HG_IDX
                  )
                  &&
                  (
                    atom2_type_vdw_hb == 0 ||
                    atom2_type_vdw_hb == 1 ||
                    atom2_type_vdw_hb == 2 ||
                    atom2_type_vdw_hb == ATYPE_HG_IDX
                  );
```

9.46%	7.55%	Global	Load

```
bool notDoubleH = (atom1_type_vdw_hb > 2 && atom1_type_vdw_hb != ATYPE_HG_IDX)
                  ||
                  (atom2_type_vdw_hb > 2 && atom2_type_vdw_hb != ATYPE_HG_IDX);
```

48	2.94%	2.54%

Potential Performance Bottlenecks

Conditional statements

```
if
(
  (
    (
      (atom1_type_vdw_hb == ATYPE_CG_IDX) ||
      (atom1_type_vdw_hb == ATYPE_HG_IDX) ||
      (atom1_type_vdw_hb == ATYPE_NG_IDX) ||
      (atom1_type_vdw_hb == ATYPE_OG_IDX)
    )
    &&
    (
      atom2_type_vdw_hb == ATYPE_G0_IDX
    )
  )
  ||
  (
    (
      atom1_type_vdw_hb == ATYPE_G0_IDX
    )
    &&
    (
      (atom2_type_vdw_hb == ATYPE_CG_IDX) ||
      (atom2_type_vdw_hb == ATYPE_HG_IDX) ||
      (atom2_type_vdw_hb == ATYPE_NG_IDX) ||
      (atom2_type_vdw_hb == ATYPE_OG_IDX)
    )
  )
)
```

0.62% 9.84%

```
if
(
  (atom1_type_vdw_hb > ATYPE_G0_IDX && atom2_type_vdw_hb == ATYPE_G0_IDX)
  ||
  (atom1_type_vdw_hb == ATYPE_G0_IDX && atom2_type_vdw_hb > ATYPE_G0_IDX)
)
```

48 0.15% 4.44%

Performance ratio

- 1.02x

Potential Performance Bottlenecks

Remove unnecessary atomic operations

```
ATOMICSUBI32(&gradient[atom1_id].x, priv_intra_gradient_x);
ATOMICSUBI32(&gradient[atom1_id].y, priv_intra_gradient_y);
ATOMICSUBI32(&gradient[atom1_id].z, priv_intra_gradient_z);

ATOMICADDI32(&gradient[atom2_id].x, priv_intra_gradient_x);
ATOMICADDI32(&gradient[atom2_id].y, priv_intra_gradient_y);
ATOMICADDI32(&gradient[atom2_id].z, priv_intra_gradient_z);
```

44	8.82%	3.17%	Global
44	3.28%	1.27%	Global
44	2.09%	1.27%	Global
43	2.25%	2.54%	Global
39	2.31%	0.63%	Global
38	2.13%	0.63%	Global

Time

8.67 s

```
gradient[atom1_id].x -= priv_intra_gradient_x;
gradient[atom1_id].y -= priv_intra_gradient_y;
gradient[atom1_id].z -= priv_intra_gradient_z;

gradient[atom2_id].x += priv_intra_gradient_x;
gradient[atom2_id].y += priv_intra_gradient_y;
gradient[atom2_id].z += priv_intra_gradient_z;
```

45	10.51%	3.67%	Global(2)
45	1.53%	1.84%	Global(2)
46	1.38%	1.84%	Global(2)
45	9.79%	3.06%	Global(2)
42	1.17%	1.84%	Global(2)
43	1.04%	1.84%	Global(2)

Time

1.21 s

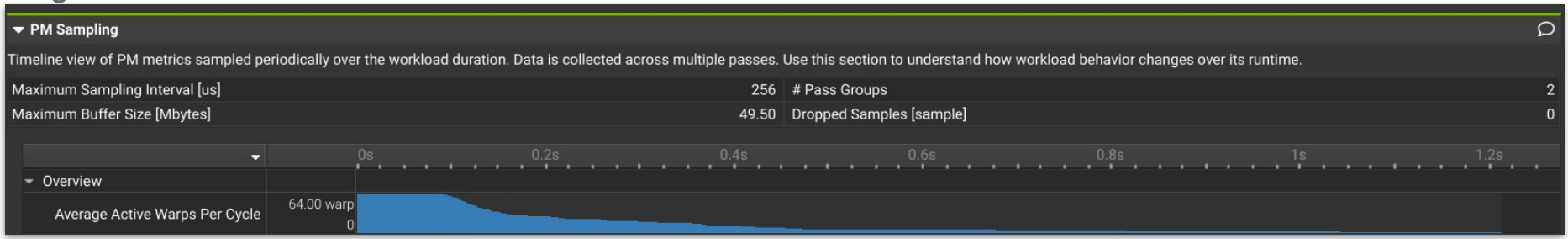
Performance ratio

- 4.68x

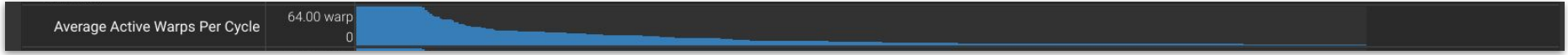
Potential Performance Bottlenecks

Algorithm modification

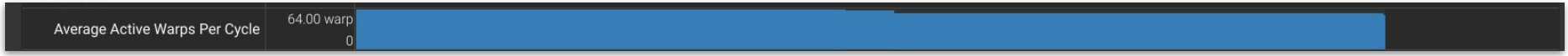
Original



Modified Test 1



Modified Test 2



Performance ratio

- 1.00x ~ 1.18x

Overall Improvement

Overall Improvement

7.79

Size of blockDim

1.38

Algorithm modification

1.18

Conditional statements

1.02

printf() in kernel

1.01

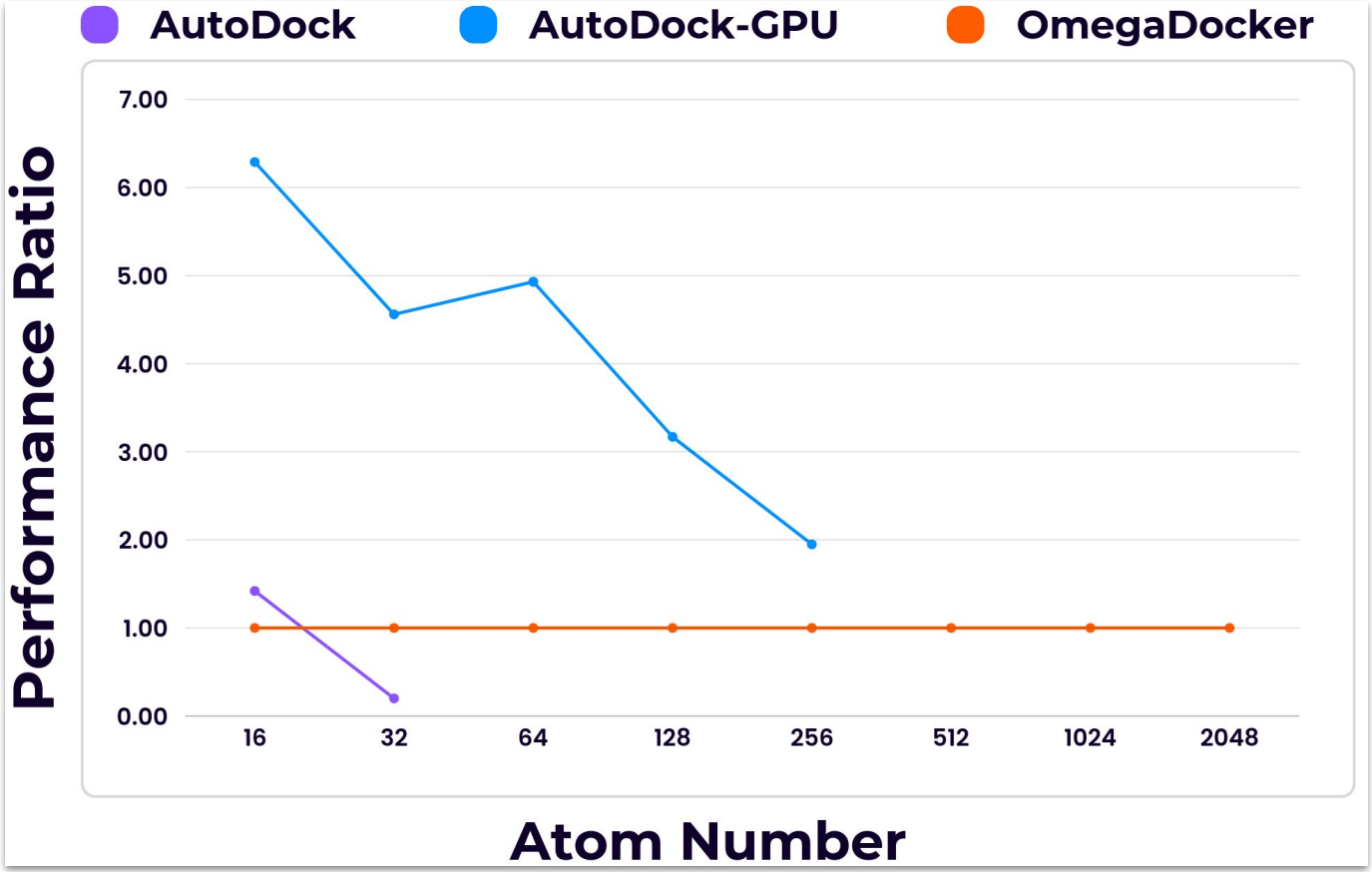
Prefetch

N/A

Remove unnecessary atomic operations

4.68

Availability and Performance



Not Everything, Not yet.

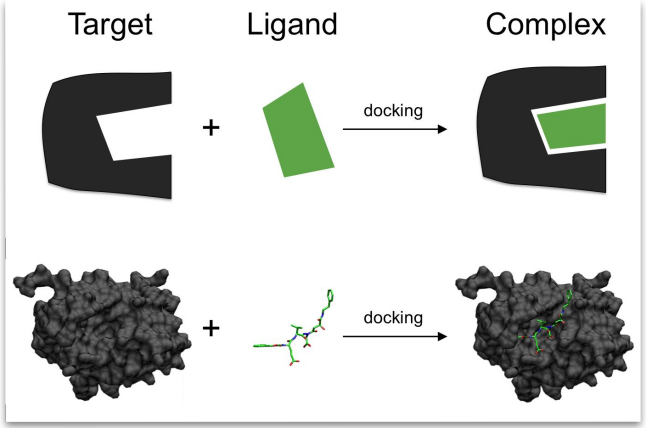
Further improvement

- Prefetch
- Other tricks for accelerating the IO of global memory
- Tensor Cores utilization

Learning never ends

- Nsight Compute
- Profiling tricks
- Advanced CUDA programming

分子對接



Dream Chaser 團隊成員長庚大學醫學系高銓澤博士、人工智慧學系馬誠佑老師、與資工系李季青老師，解除對接軟體 AutoDock 的原子數量限制，並保持穩定的效能。

— NVIDIA Mentors: Anthony Chang, Ying-Ja Chen

分子對接是加速藥物開發不可或缺的重要工具，有效的電腦計算結果，能節省大量後期生物實驗與臨床試驗的費用與時間。然而就算在電腦的輔助之下，藥物開發的平均時長約十年，且 2024 年六月的研究指出，2018 年的平均藥物開發成本約爲 8.793 億美元，並且逐年上升。近年除了小分子藥物以外，較小的生物分子如 miRNA 與 aptamer 對於調節細胞生長與功能的研究需求也是與日俱增，由於其可能成爲潛在的抗癌新星，讓許多學者投入相關研究，然而目前支援 GPU 的分子對接軟體有藥物原子數量上限的限制，使得相關研究難以順利進行。Dream Chaser 團隊與 NVIDIA 成員合作，在解除原子數量限制的情況下，透過 CUDA 加速獲得穩定的效能。