master’s project notes

Contents

1 Evaluation of NumPy vs CuPy in npstructures and BioNumpy

To initiate exploration of what CuPy does well in comparison to NumPy, I am carefully benchmarking each step of the k-mer counting pipeline, going from an input fasta file of 20 million reads to the finished counter object, in order to fully understand which parts of the npstructures and BioNumpy code is lacking relative performance using both NumPy and CuPy as the backend array module.

This pipeline, as is performed in the BioNumpy library, can be abstracted into n parts: 1) reading and formatting a chunk of reads from the input fasta file, resulting in a ragged-array containing a single read per row, where each base is represented as a single unsigned 8-bit integer. 2) Converting the chunk of reads into a compact bit-array representation, only using 2 bits per base. 3) retrieving each 31-mer from the bit-array and representing each 31-mer as a 64-bit integer. 4) ...

1.1 Reading fasta file and creating chunks of reads

Practically all of the time spent when preparing each chunk in the chunk generator is spent in OneLineBuffer's get_data() method. Further examining this method revealed that the entire process of looping through each chunk (with the chunk size set to the default 5M bytes), spent only 3.4 of 13.8 total seconds in a

2 Examining counter implementations

2.1 The npstructures counter object

The counter object found in the npstructures package is a specialized hashtable used for counting frequencies of unique 64-bit data values. The original npstructures counter object and its underlying hashtable are implemented in Python, and therefore rely heavily on NumPy to perform its underlying functionalities efficiently. Describe the npstructures counter object ...

Diagrams?

2.2 CUDA counter

One of the primary issues with the npstructures counter object is that its implementation is constrained by the functionalities provided by NumPy for good performance. This results in a sequence of, perhaps unnecessary, array operations for initialization, counting and lookups. In an attempt to mitigate this, I implemented a simpler and more straightforward counter object in CUDA and used the pybind11 library to create Python bindings for the implementation as to create a drop-in replacement for the npstructures counter object.

Describe the CUDA counter implementation ...

2.3 Performance comparison

...

Comparison between different hashtable capacity sizes for the CUDA hashtable implementation. We expect faster initialization, counting and lookup when the capacity is larger. The following bar chart show the time spent processing counts in the hashtable for 20 million reads of 150 bases. The counter object was initialized with 20 million unique kmers as keys, and the fasta file containing the reads was in chunks of 10 million bytes using BioNumPy. Since the counter object was initialized with 20 million keys, the theoretical smallest possible capacity of the hashtable was 20 million slots. The tests benchmarked the total time spent waiting for the count call, summarized from all the chunks making up the 20 million reads.

...

3 Fast poisson logpmf

3.1 The implementations

The poisson logpmf is something aaaa

We have four different implementations performing (poisson logpmf?) ... :

The first implementation we consider is the implementation provided by the SciPy [1]. This

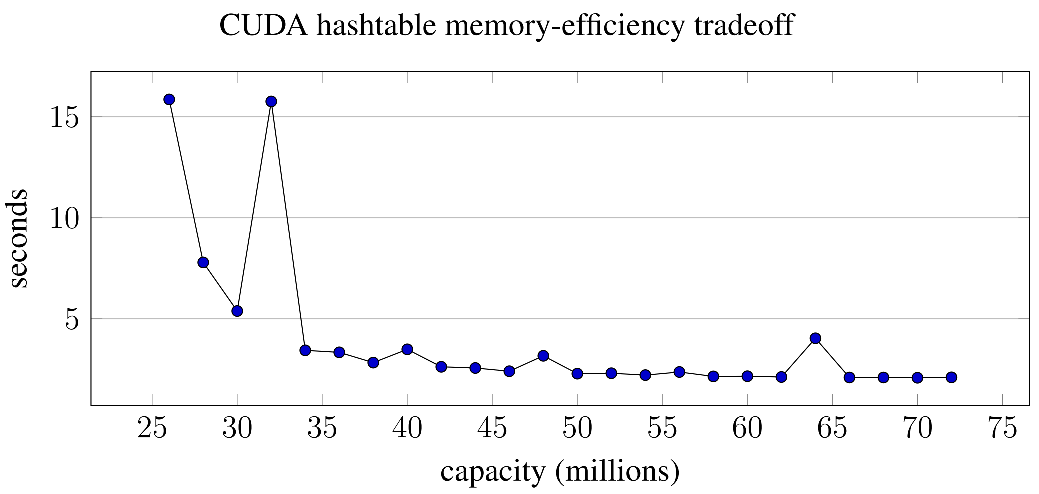

Figure 1: The total time spent waiting for the count calls when counting 20 million reads of 150 bases each. As expected, the tradeoff between memory and efficiency becomes apparent as larger capacities result in faster counting times. Although we quickly reach a point of dimishing returns, we see small improvements all the way up to a capacity of 70 million slots, with a capacity of 70 million slots spending a total of 2.079 seconds counting 20 million reads. The benchmark was only ran from 26 million up to 72 million slots. Time consumption grew exponentially with smaller capacities than 26 million for this counter of 20 million keys.

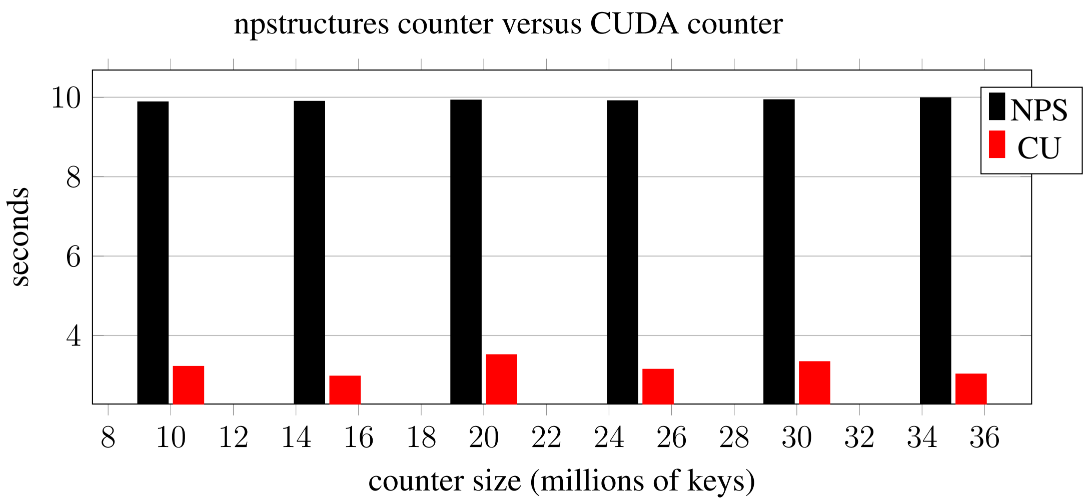

Figure 2: CUDA counter uses a capacity factor of 2.0. The chunk size used to read the 20 million reads was 10 million bytes.

implementation runs on the CPU.

```
from scipy.stats import poisson

def poisson_logpmf(k, r):
    return poisson.logpmf(k, r)
```

The second implementation is an implementation created by I&K using NumPy [2] and SciPy. This implementation also runs on the CPU, but is nearly twice as fast as the one provided by the SciPy library when the k and r matrices have 40 million elements each.

```python
import numpy as np
import scipy

def poisson_logpmf(k, r):
    return k * np.log(r) - r - scipy.special.gammaln(k+1)
```

The third implementation piggybacks off of the previous NumPy implementation, but is ported to CuPy [3]. This implementation achieves large speed increases over the NumPy equivalent for large matrices.

```python
import cupy as cp

def poisson_logpmf(k, r):
    return k * cp.log(r) - r - cupyx.scipy.special.gammaln(k+1)
```

The CUDA implementation (just the kernel).

```cuda
__global__ void poisson_logpmf_kernel(const int *k, const float *r,
    float *out)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  out[i] = k[i] * logf(r[i]) - r[i] - lgammaf(k[i]+1);
}
```

Both the CuPy and the CUDA implementations allow for all, some or none of the input matrices to be located in the GPU's global memory. Any input not located in GPU memory will be copied to the device in order for the computation to be performed by the GPU. Only when both inputs are located on the host will the returned results also be located on the host, meaning that the input matrices will be copied from the host to the device for computation, before the result is copied from the device back to host. This introduces some extra variations for the CuPy and CUDA implementations, resulting in the following set of variations for both of them:

- When both of the input matrices are located on the host, and the returning data is located on the host: hh2h
- When one of the input matrices are located on the host and the other on the device, and the returning data is located on the device: dh2d
- When both of the input matrices are located on the device, and the returning data is located on the device: dd2d

We omit the variation where both the inputs are located on the host and the output is located on the device.

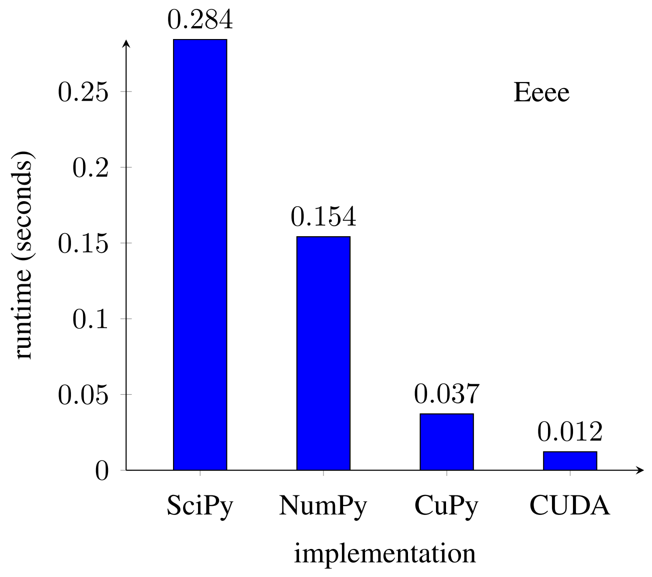

Figure 3: Runtime in seconds averaged over 100 function calls where len(k) = len(r) = 40,000,000

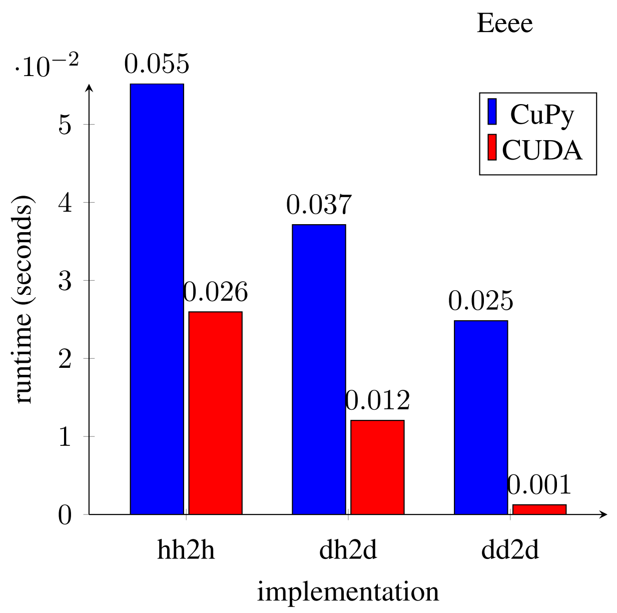

Figure 4: Runtime in seconds averaged over 100 function calls where len(k) = len(r) = 40,000,000

References

[1] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: `10.1038/s41592-019-0686-2`.

[2] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[3] Ryosuke Okuta et al. “CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations”. In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017. URL: `http://learningsys.org/nips17/assets/papers/paper_16.pdf`.