Rapid Development for Manycore Devices with Numba and NumbaPro

Siu Kwan Lam and Graham Markall

CONTINUUM ANALYTICS

Introduction

- When **speed of execution and speed of development** are important, ***Numba*** provides a mechanism to get native-code performance out of Python syntax, targeting X86, multicore X86 and NVIDIA GPUs.
- Why Python for technical computing?
  - Ecosystem of libraries for scientific computing (NumPy, SciPy, Matplotlib, etc.).
  - "Just enough" support for implementing just-in-time compilation of array-oriented code:
    - Operator overloading,
    - annotation of methods with decorators,
    - and arrays in typed containers from NumPy.

Interface

Numba exposes an interface for just-in-time compilation through decorators:

`@jit` : marks a function for just-in-time compilation

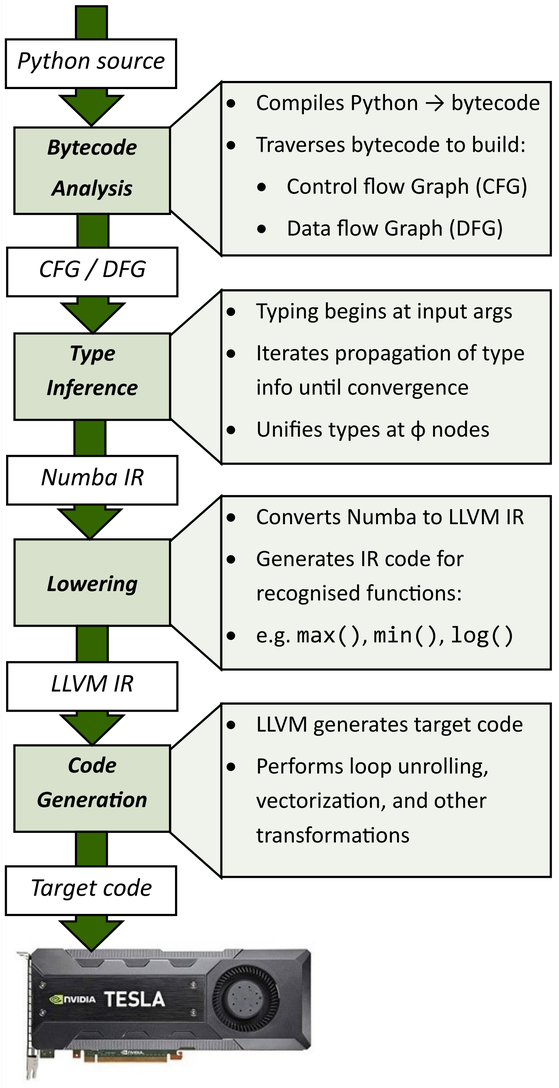
`@vectorize` : create a "universal function" - write the kernel for one element of an array, runtime handles application to all elements of input vectors.

`@guvectorize` : generalization of `@vectorize` operating on non-scalar input types.

Example Matrix Multiplication Kernel:

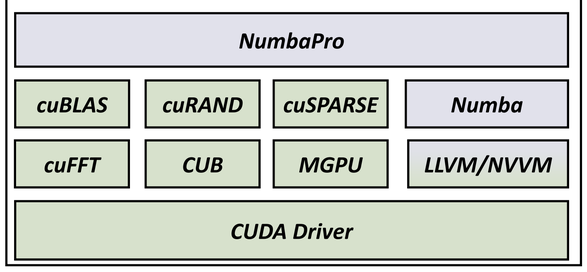
```
@cuda.jit(argtypes=[f4[:,:], f4[:,:], f4[:,:]])
def cu_square_matrix_mul(A, B, C):
    tx, ty = cuda.threadIdx.x, cuda.threadIdx.y
    bx, by = cuda.blockIdx.x, cuda.blockIdx.y
    bw, bh = cuda.blockDim.x, cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh

    if x >= n or y >= n:
        return

    C[y, x] = 0
    for i in range(n):
        C[y, x] += A[y, i] * B[i, x]
```

The Numba Compilation Process



CUDA Libraries

NumbaPro provides access to NVIDIA's CUDA libraries:



Example Application: Sparse PCA

The Algorithm

Principal Component Analysis (PCA) is an important tool in data analytics for identifying the most important combinations of observables in large datasets. Sparse Principal Component Analysis (SPCA) [1], is a novel algorithm for approximating the principal components with sparsity enforcement. Enforcing sparsity improves the interpretability of the principal components by limiting the principal components to each only contain a few observables.

The SPCA algorithm consists of random number generation, several basic linear algebra routines and *k*-selection, with the *k*-selection step dominating the computation time. A *k*-selection finds the largest or smallest *k* items in an unordered list, and can be generalized to a sort.

Implementation Discussion

The main challenge is the efficient implementation of *k*-selection. An initial implementation was based on *quicksort*, which is fast for small cases but does not scale due to the use of shared memory. To improve scalability, we implemented a GPU *radixsort* and a *mergesort*-based *segmented-sort* [2]. The radixsort has high throughput for large datasets but has high initial overheads. The segmented-sort performs a sort on each segment inside an array. Figure 1 shows a benchmark of the segmented-sort and radixsort running in batches of 1, 10, and 100. Table 1 summarises the recommended sorting algorithm based on the results of this benchmark.

Experiments

We developed two SPCA implementations: a CPU version using NumPy and a GPU version using Numba/NumbaPro.

Figure 2 shows the normalized performance of the GPU code over the CPU baseline code. For large inputs, the GPU implementation shows a speedup of ~28x over the single-threaded CPU implementation. Assuming linear scaling for a parallel CPU implementation, this would result in a ~3.5x speedup of the GPU implementation over a parallel CPU implementation executing on 8 cores.

Source code: https://github.com/ContinuumIO/numbapro-spca

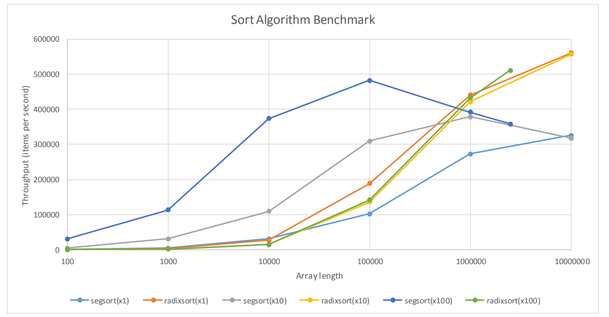


Figure 1. Throughput of segmented mergesort and radixsort for batches of 1, 10, and 100 an a Tesla K20c.

Data size/ No. of arrays	Up to 256 elements	Up to 1000 elements	Up to 10000 elements	Above 10000 elements
1 array	Quicksort	Segmented mergesort	Radixsort	Radixsort
< 10 arrays	Quicksort	Segmented mergesort	Segmented mergesort	Radixsort
> 10 arrays	Quicksort	Segmented mergesort	Segmented mergesort	Radixsort

Table 1. Recommended sorting algorithm by input array length and count based on benchmark results.

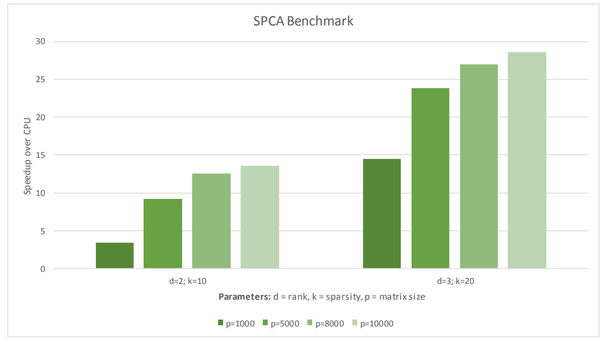


Figure 2. Speedup of SPCA implementation on a Tesla K20c over a single core of an Intel i7-4820K.

References

[1] Asteris, Megasthenis, Dimitris Papailiopoulos, and Alexandros Dimakis. "Nonnegative Sparse PCA with Provable Guarantees." *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* 2014: 1728-1736.

[2] "Segmented Sort and Locality Sort - Modern GPU - NVlabs." 2013. http://nvlabs.github.io/moderngpu/segsort.html

Get Numba using the Anaconda package manager:

```
conda install numba
conda install numbapro
```

Github: https://github.com/numba/numba