Vectorizing common HEP analysis algorithms

Author:

Jaydeep Nandi

Mentors:

Jim Pivarski

David Lange

Contents

1. Scalar programming and vectorization
2. Vectorization and GPU
3. Autovectorization
4. Vectorization of HEP analysis algorithms
    1. Argproduct
    2. Local reduction
5. Conclusions

Scalar Pogramming and Vectorization

Scalar Programming

- Works on scalar (individual elements) values of an array.
- Applies an operation over the elements via a loop, usually a **for-loop** or **while** loop.
- Low-level, and tedious to write.
- Virtually all programming languages support scalar programming.

Scalar Programming

- **For example:** If we wish to add two compatible vectors (of same shape and castable type) ***A*** and ***B*** together and store it in **C**, a typical scalar program (in Python) would look like:

```python
for i in range(len(A)):
    C[i] = A[i] + B[i]
```

- It operates on each element of the arrays sequentially.

What is Vectorization?

- Converting a scalar program to a vector program.
- It is also known as Array Programming.
- Wikipedia Definition: “In computer science, **array programming languages** (also known as vector or **multidimensional** languages) generalize operations on scalars to apply transparently to vectors, matrices, and higher-dimensional arrays.“
- Allows applying an operation on multiple data items simultaneously

Array Languages

- Some languages and libraries support vectorization by default. Examples include:
  - **Python: Numpy, Scipy etc.**
  - **MATLAB**
  - **GNU OCTAVE**
  - **R etc.**
- Usually they compute the vectors under the hood as efficient "**C**" or "**FORTRAN**" implementations.

Array Languages

- Recollect the addition of all elements of two vectors to give a new vector, which we implemented as a scalar code:

```
for i in range(len(A)):
    C[i] = A[i] + B[i]
```

- In an array language, it is quite simple to write:

```
C = A + B
```

So why Vectorize?

- Simple to write; no need of writing loops incessantly.
- Expressive, from a mathematical point of view.
- But those are not the major reason for vectorising code.
  - Most modern CPUs and GPUs provide support for vectorised code, which runs very efficiently, operating in a SIMD style.
  - Can take advantage of SSE (Streaming SIMD Extensions) and AVX instructions, which operate on 4, 8 or more data simultaneously.
  - GPUs take it even further: Can operate on a large number, typically in thousands, of data at once. (More on it later!)

Vectorization and GPUs

General Purpose GPU (GPGPU) Programming

- Works on Single Instruction Multiple Thread (SIMT) architecture.
- Processes multiple (depending on the number of threads available) data elements simultaneously.
- However, it requires parallel instructions, which don’t have a dependency that can create a data race.
- Can be programmed with CUDA for Nvidia cards, and OpenCL for AMD cards and others.
- Primarily in C; bindings exist for other languages like PyCUDA and PyOPENCL for Python.

Vectorization and GPGPU programming

- Vectorized codes are inherently parallel. They apply same instructions on multiple data elements of the vector at once.
- Is in perfect agreement with GPGPU philosophy.
- Vectorized codes thus give very high speedups if operated on the GPU.

Autovectorization

Autovectorization

- Modern compilers allow the vectorization of simple sequential loops into efficient SIMD instructions for CPU, typically through a compiler switch.
- Some compilers which support this:
  - **GCC**: Supports through switch ***–ftree-vectorize*** or **–O3** level optimization.
  - **ICC:** From Intel. Reported to be better than GCC at vectorising code.
  - **MSVC and others.**
- They check if a loop is safe to be vectorised, and if it is, they vectorise the code. The list of such possible vectorizable loops fo GCC can be found here.

Autovectorization

- As an example, consider again the addition of two vectors. A simple c loop that does is :

```
for ( int i=0; i<length(A); i++)
    C[i] = A[i] + B[i];
```

- The corresponding assembly dump (relevant part) is:

```
vmovdqu xmm0, XMMWORD PTR [r9+rax]
add edx, 1
vinserti128 ymm0, ymm0, XMMWORD PTR [r9+16+rax], 0x1
vpaddd ymm0, ymm0, YMMWORD PTR [r13+0+rax]
vmovups XMMWORD PTR [rcx+rax], xmm0
```

- Note the vector instructions in the assembly (**vmovdqu**, **vinserti** etc)

Autovectorization

- Advantages:
  - Sufficiently simpler than writing SSE or AVX instructions.
  - Easy to debug.
  - Portable.
- There is however, a major disadvantage. They cannot vectorise a loop if there is a loop carried dependency.

Barriers to Autovectorization

- The major barrier is a loop carried dependency. They are variables whose particular value depends on the order of the execution.
- As an example consider the problem of finding the max value of all elements in an array **A**.

```
for (int i=0; i<N; i++)
    maxval = max(maxval, A[i]);
```

- The value in maxval at any instance depends on the order of loop execution. So, it’s a loop carried dependency, and the compiler cannot autovectorize it.

Barriers to Autovectorization

- The assembly output of the loop is

```
mov edx, DWORD PTR [rax]
add rax, 4
cmp ecx, edx
jl .L12
cmp rax, rsi
jne .L11
mov edx, ecx
```

- Note the absence of vector instructions in the assembly, indicating an unvectorized loop.
- The reduction operation can be vectorised, and will be dealt with shortly.

Vectorization of HEP analysis algorithms

Algorithms Considered

- Argproduct
- Local Reduction
- And others.

- For the complete list of algorithms and their detailed analysis, please refer to the complete report available here.

- Note that the algorithms are not physics-specific algorithms per se, but primitives which can be used to build them. They are meant to serve the purpose of demonstrating the efficiency of vectorization.

Argproduct

Motivation: Relativistic mass and boson detection

- A typical application in HEP is operating on the combinations or pairs of different particles in an event.
- Higgs bosons and Z bosons decay so quickly that they're gone before they ever reach a detector. All we can detect is the electrons and/or muons, and sometimes not even that, if an electron or muon flies past the detector without entering it.
- One powerful technique relies on relativistic mass: Higgs bosons have a (fairly) well-defined mass and Z bosons have a (fairly) well-defined mass (with some variation due to quantum effects, some due to measurement error). When a particle decays, its decay products have the same total energy E as the original particle (a scalar number), as well as the same total momentum P.

Motivation: Relativistic mass and boson detection

- The relativistic mass is given as $m^2 = E^2 - P^2$.
- So if we compute mass from the total energy and total momentum of a set of particles, that mass will be approximately single-valued if the set of particles came from a particle of a given mass.
- The particles, that came from Higgs or Z boson decay can be determined by calculating the masses of ***pairs*** of muons and electrons.
- The particles that originate from Higgs or Z will add up to the same mass as them (91 GeV for Z boson for example).
- With **argproduct**, we can generate the indices of pairs of particles from same or different sets, which can then be used in the pairwise mass calculation as above.

Argproduct

- Argproduct returns two arrays, consisting of the index of first element in the pair, and another array consisting of the index of the second element in the pair.
- Argproduct isn’t inherently vectorizable, it has a loop carried dependency.
- Some definitions:
  - **Events:** Refers to a set of particles or elements that have some property in common.
  - **Offsets:** Gives the start and end indices of an event. The start indices are given by **starts**, while the stop indices by **stops**.
  - **Parent:** An array that gives the event id for every element or particle.
  - **Local and Global Operations:** If a operation is done ithin the elements of an event only, it is local in nature. Global operations work on particles from all events.

Argproduct

- Sequential python code:

```
for i in range(events):
pairs_i = 0
for j in range(starts1[i], stops1[i]):
    for k in range(strats2[i], stops2[i]):
        first[pairs_i] = j
        second[pairs_i] = k
```

- Note that pairs_i is a loop carried dependency.

Argproduct

- The code can be vectorised if we have the information of
  - **Parents** array of every pair.
  - The number of elements in either array, say given by **counts** array, and the **starts** and **stops** arrays for the two particle sets.
  - A running index array for all the pairs in all the events.
- An important thing to notice is that if we consider the linear indexed arrays as matrices, then the first and second arrays are given by the row index and column index of the matrix.
- So, we just need a way to convert the linear index to matrix index.

Argproduct

- For a simple illustration, consider the two arrays

```
arr1 = [‘a’, ‘b’, ‘c’]
arr2 = [1,2]
```

- The output of argproduct will be

```
first = [0,0,1,1,2,2]
second = [0,1,0,1,0,1]
```

- Which correspond to the pairs

```
[[‘a’, 1],[‘a’, 2],[‘b’, 1],[‘b’, 2],[‘c’, 1],[‘c’, 2]]
```

Argproduct

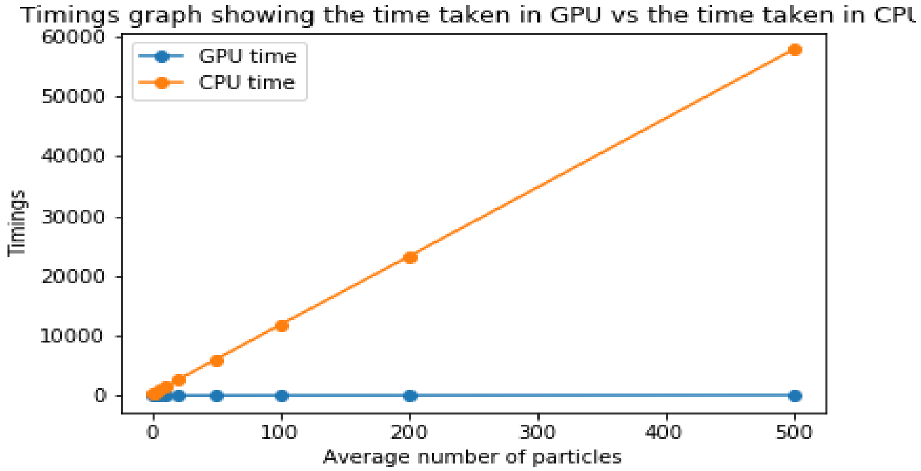
- Now, there are 3*2 = 6 possible pairs. Notice that if we have a running index array:
index = [0,1,2,3,4,5]
- Then **first** can be generated as the integer division of elements in **index** with number of elements in second array, given by **counts,** and **second** is the result of **modulo** operation on index by **counts.**

**first = ** index // counts
second = index%counts

- The operations are vectorised.
- The per-event case then can be derived by giving the proper offset to the indices.

Performance Improvements of argproduct

- Vectorizing the argproduct gives a high speedup, especially when considered between GPU and CPU.



Local Reduction

Motivation: Generated Reconstructed particle matching

- A particular application in HEP is to find the matching between generated particles and reconstructed particles.
- Generated particles are the truth or ideal values, and reconstructed particles are the detected ones in an experiment.
- Gen-reco matching attempts to find which reconstructed particles belong to a particular generated particle class.
- Usually achieved by minimizing an optimization criteria, which we can call the predicate function. An example of that is the minimum deltaR criteria.
- We aim to find the particle pair that achieves the best value of optimization criteria in an event.
- This requires local reduction, that is reduction *per event*.

Local Reduction

- Reduces an array per-event. The reduction operator can be any associative operator like **max()**, **min()** or **sum().**
- Significantly tough to vectorise.
- We shall consider max(), and the rest can be simply substituted in place of max()
- Sequential code

```
for i in range(events):
    for i in range(starts[i], stops[i]):
        val = max(val, A[i])
```

Local reduction

- The reduction has been implemented in parallel through a modified version of Hillis-Steele algorithm.
- Let us consider a single event first. The pseudocode for the upsweep phase of the algorithm is

```
for d=0 to ($\log_2 n$ -1) do
    forall k=0 to n-1 by $2^{d+1}$ do
        a[k+$2^{d+1}$-1] = max(a[k+$2^d$-1], a[k+$2^{d+1}$-1])
```

- The algorithm proceeds by tree reducing the array elements, until a single element remains.

Local Reduction

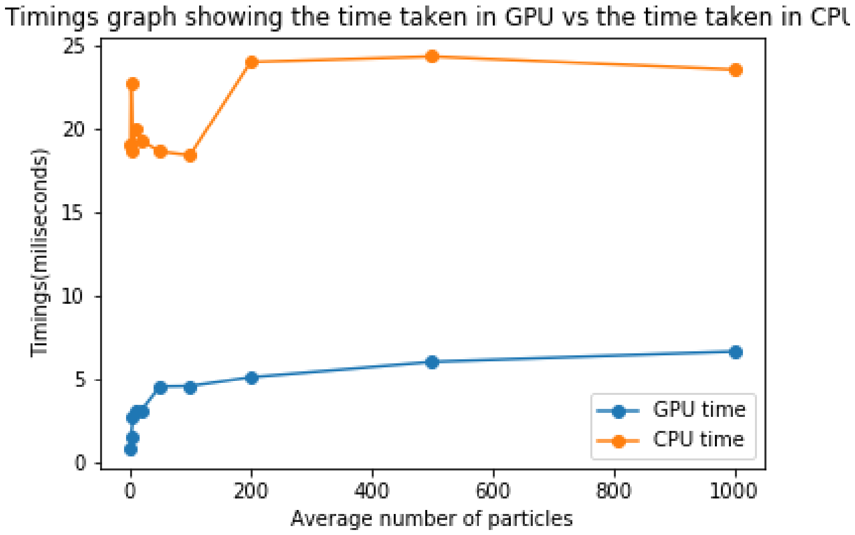
- To extend it to per-event case, we will again require the parents array for the data.
- It will serve as a mask for the array elements, so that data isn't accessed across events.

The modified pseudocode will then look like

```
for d=0 to ($\log_2 n$ -1) do
    forall k=0 to n-1 by $2^{d+1}$ do
        if (parents[$k+2^{d+1}-1$] == parents[$k+2^{d+1}-1$])
            a[$k+2^{d+1}-1$] = max(a[$k+2^{d}-1$], a[$k+2^{d+1}-1$])
```

Local Reduction performance

- The vectorised GPU code performs moderately well in comparison to the sequential code. The timings as a function of average number of particles are given below:



Conclusions

- Vectorization and parallelization of code achieves superior performance over sequential implementation.
- Vectorized code can be offloaded to GPU, achieving high runtime speeds.
- Code remains clean and efficient.

Thank You!