Boosting Python Performance on Intel Processors: A case study of optimizing music recognition

Speaker: Yuanzhe Li, Wayne State University
Existing works & Potential approach

• Interleaving Python with low level languages

• Existing studies:
  1. Cython: in code optimization, multithreading, labor intensive
  2. Library: integrate NumPy, transparent use of GPU
  3. Custom distribution: PyCUDA, Intel Python

• Potential approaches:
  1. Deeply optimized vs Generally optimized
  2. Optimized for one type accelerator
Music fingerprint and recognition algorithm

1. Extract digital data and apply FFT to the data to make spectrogram.
2. Identify local maxima (peaks) from “neighbors” (filter + image processing).
3. Collect peaks and create fingerprints (a set of unique hashes).
4. Match fingerprints of sample audio to the fingerprints in database.
Dejavu: Implementation and challenges

• Have multiprocessing implemented (pool)

• Design in Python:
  1. pyaudio for grabbing audio from microphone
  2. ffmpeg for converting audio files to .wav format
  3. numpy for taking the FFT of audio signals
  4. scipy in local maxima (peak) finding algorithms
  5. matplotlib for spectrograms and plotting

• Hotspot and challenges:
  • Local comparison on each input element
  • Peak identifying: Maximum filter function in scipy
  • Takes 72% of total running time
Why Intel?

• On Intel V.S. on GPU
  1. Require less labor, and easy to start.
  2. GPU more suitable for SIMD operation intensive work.
  3. Intel has more cache memory resources (better for this work).
  4. Some studies have been done on GPU. However, high performance implementation on Intel is unexplored.

• Intel has powerful support, like Intel Python (re-designed libraries), and MKL.
Intel ARCH and Performance

• Intel Xeon Haswell processor:
  • 2 sockets, 14 cores on each socket
  • On core, two hyper-threads, two 256-bit vector register for SIMD operations (AVX2).

• Timing data for FFT and Max_Filter are the total execution time of 28 cores.

<table>
<thead>
<tr>
<th></th>
<th>Wall clock time</th>
<th>FFT</th>
<th>Max_Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Python</td>
<td>421.11s</td>
<td>458.83s</td>
<td>8563.55s</td>
</tr>
<tr>
<td>Intel Python</td>
<td>348.44s</td>
<td>693.08s</td>
<td>7073.48s</td>
</tr>
<tr>
<td>IntPy 1 thread/proc</td>
<td>277.45s</td>
<td>389.84s</td>
<td>5584.07s</td>
</tr>
</tbody>
</table>
Thread Level Parallelism

• Local comparisons can have thread level parallelism
• No parallelism when have multiple threads
• Scipy function has data dependency
  • Pointer for current element depends on previous
• Table timing are in wall clock time
• Performance implies high latency

<table>
<thead>
<tr>
<th></th>
<th>4 songs</th>
<th>369 songs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4P/7T</td>
<td>4P/4T</td>
</tr>
<tr>
<td></td>
<td>12.90s</td>
<td>16.31s</td>
</tr>
<tr>
<td></td>
<td>28P/1T</td>
<td>28P/2T</td>
</tr>
<tr>
<td></td>
<td>273.49s</td>
<td>235.12s</td>
</tr>
</tbody>
</table>
Memory Latency

• High memory and L3 cache access
• Irregular memory access
• Output matrix is the transpose of input matrix
  ▪ One cache line read requires 8 writes to scattered cache lines (element type of double)
  ▪ Loop tiling, cache oblivious, output matrix transposition
• Improve on input is possible but not implemented
Loop tiling, cache oblivious, and performance

<table>
<thead>
<tr>
<th>j=0,1,2,...,n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>16 17 18 19 20 21 22 23</td>
</tr>
<tr>
<td>24 25 26 27 28 29 30 31</td>
</tr>
<tr>
<td>32 33 34 35 36 37 38 39</td>
</tr>
<tr>
<td>40 41 42 43 44 45 46 47</td>
</tr>
<tr>
<td>48 49 50 51 52 53 54 55</td>
</tr>
<tr>
<td>56 57 58 59 60 61 62 63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j=0,1,2,...,n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 8 10 32 34 40 42</td>
</tr>
<tr>
<td>1 3 9 11 33 35 41 43</td>
</tr>
<tr>
<td>4 6 12 14 36 38 44 46</td>
</tr>
<tr>
<td>5 7 13 15 37 39 45 47</td>
</tr>
<tr>
<td>16 18 24 26 48 50 56 58</td>
</tr>
<tr>
<td>17 19 25 27 49 51 57 59</td>
</tr>
<tr>
<td>20 22 28 30 52 54 60 62</td>
</tr>
<tr>
<td>21 23 29 31 53 55 61 63</td>
</tr>
</tbody>
</table>

Picture is snapped from “Parallel Programming and Optimization with Intel Xeon Phi Coprocessor”

<table>
<thead>
<tr>
<th></th>
<th>ORIG</th>
<th>Loop Tiling</th>
<th>Cache Oblivious</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transpose</td>
<td>164.89s</td>
<td>162.01s</td>
<td>284.17s</td>
</tr>
<tr>
<td>No Trans</td>
<td>235.12s</td>
<td>208.76s</td>
<td>341.52s</td>
</tr>
</tbody>
</table>

wayne.edu
Vectorization

• On core Vector Processing Unit (VPU)
• 256 bits vector register = 4 double type data
• Scipy implementation has no use of vector registers
  ▪ Logical branches kill vectorization for dependency
• Moving the branches out of loop.
• Vector reduction has poor performance on AVX2.
  ▪ Auto generated vector code, hand write intrinsic code.
## Vectorization

<table>
<thead>
<tr>
<th>Thread</th>
<th>Trans</th>
<th>Non-Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>369 songs</td>
<td>28P/2T</td>
<td>138.72s</td>
</tr>
<tr>
<td></td>
<td>28P/1T</td>
<td>141.80s</td>
</tr>
<tr>
<td>4 songs</td>
<td>4P/14T</td>
<td>9.78s</td>
</tr>
<tr>
<td></td>
<td>4P/7T</td>
<td>9.49s</td>
</tr>
<tr>
<td></td>
<td>4P/1T</td>
<td>20.02s</td>
</tr>
</tbody>
</table>

```python
for i := 0 to filter_size do
begin
{ if (offset): update input;
  if (errors): update input;
  if (minimum):
    if (find min): update output;
  else:
    if (find max): update output;
} end;
```

```python
if (offset): update input;
if (errors): update input;
if (minimum):
  for i := 0 to filter_size do
  begin
  { if (find min): update output;
  }
  end;
else:
  for i := 0 to filter_size do
  begin
  { if (find max): update output;
  }
  end;
else  \\ same as above
```
Wall Clock Timing for Optimizations

Performance Of Optimizations

- ORIG
- IntelPY
- MT
- Vector
- Trans
- Final

- "FFT"
- "Filter_Func"
- "Total"
Performance of Songs per Sec

![Bar chart showing the performance of songs per second for different versions: ORIG, IntelPy, MT, Vector, Trans, and Final. The bars show the number of songs per second, with the Final version achieving the highest.]
Performance analysis

• Peak memory bandwidth: 136 GB/s
• Peak processor performance in double:

\[ P_{\text{total}} = \frac{\text{Cores} \times P_{\text{core}} \times \text{VPUs} \times l_{\text{vec}}}{S_{\text{data}}} = \frac{28 \times 2.6 \text{GHz} \times 2 \times 32 \text{Bytes}}{64 \text{Bytes}} = 582 \text{ GFLOPs} \]

• Roofline model
  • relates performance to off-chip memory bandwidth
  • reveals traffic between L1 cache and DRAM

\[ \text{Intensity} = \frac{\text{total operations}}{\text{total memory accesses}} \]
Performance analysis (cont.)

• Best intensity is obtained when both peak performance and maximum bandwidth are achieved (35.3 FLOPS/Byte)

• Computation requires 841 operations, 841 elements, and one memory write in each iteration

• High intensity when 841 elements are in L1 cache (52.56)

• Low intensity when 841 elements are in DRAM (1/8). Giving the worst performance (2.06 GFLOPS)
Performance analysis (cont.)

• Real performance is calculated as dividing total operations by total running time

• A special test with 28 copies of one selected song
  • no idle cores
  • same workload on each core

• 52.27 GFLOPS, latency bounded
Contributions & Future Works

1. Apply music recognition algorithm to Intel processor efficiently
2. Give details for optimizing Python libraries from multiple aspects
3. Our redesigned function also works for other Python projects
4. The idea is also applicable to other libraries
5. Potential works on irregular input access
   • von Neumann neighborhood structure