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

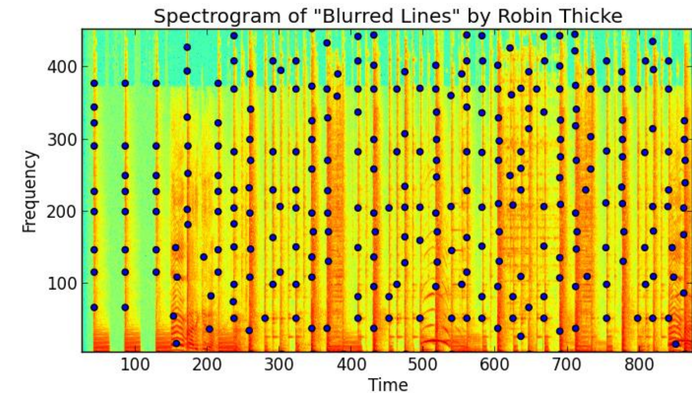
Speaker: Yuanzhe Li, Wayne State University

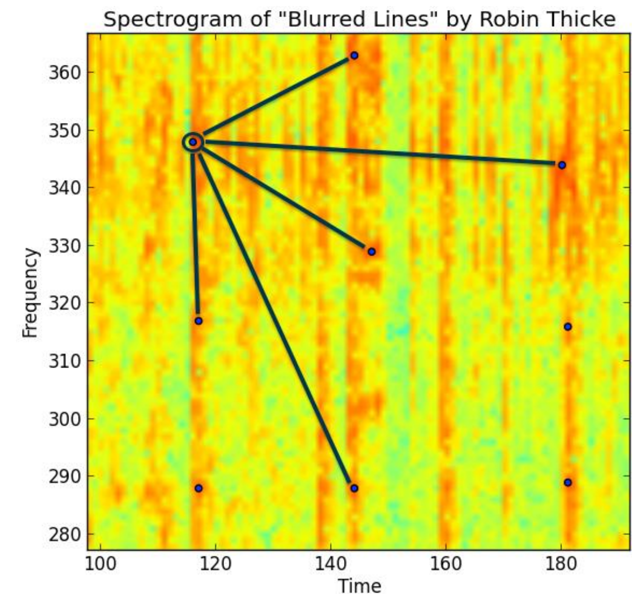
Existing works & Potential approach

- 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.

	Wall clock time	FFT	Max_Filter
Standard Python	421.11s	458.83s	8563.55s
Intel Python	348.44s	693.08s	7073.48s
IntPy 1 thread/proc	277.45s	389.84s	5584.07s

Thread Level Parallelism

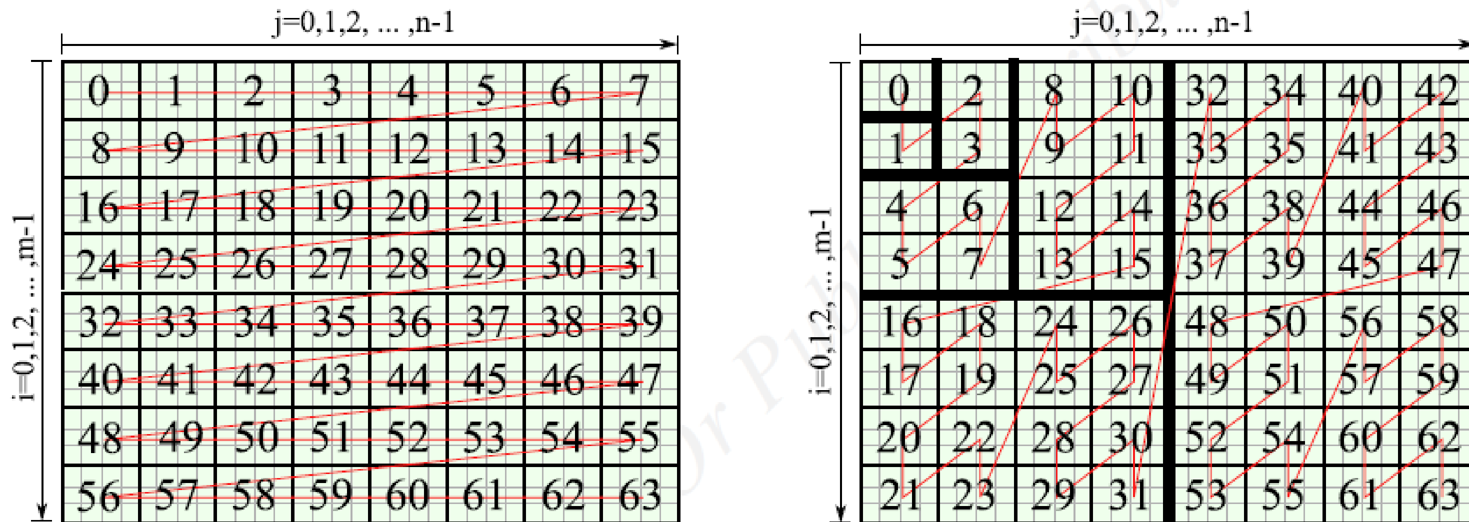
- Local comparisons can have thread level parallelism
- No parallelism when have multiple threads
- Scipy function has data dependency
  - Pointer for current element depends on previous
- Table timing are in wall clock time
- Performance implies high latency

4 songs	**4P/7T**	**4P/4T**	**4p/1T**	**N/A**
	12.90s	16.31s	43.71s	N/A
369 songs	**28P/1T**	**28P/2T**	**14P/4T**	**1P/56T**
	273.49s	235.12s	273.00s	1507.98s

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



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

	ORIG	Loop Tiling	Cache Oblivious
Transpose	164.89s	162.01s	284.17s
No Trans	235.12s	208.76s	341.52s

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.

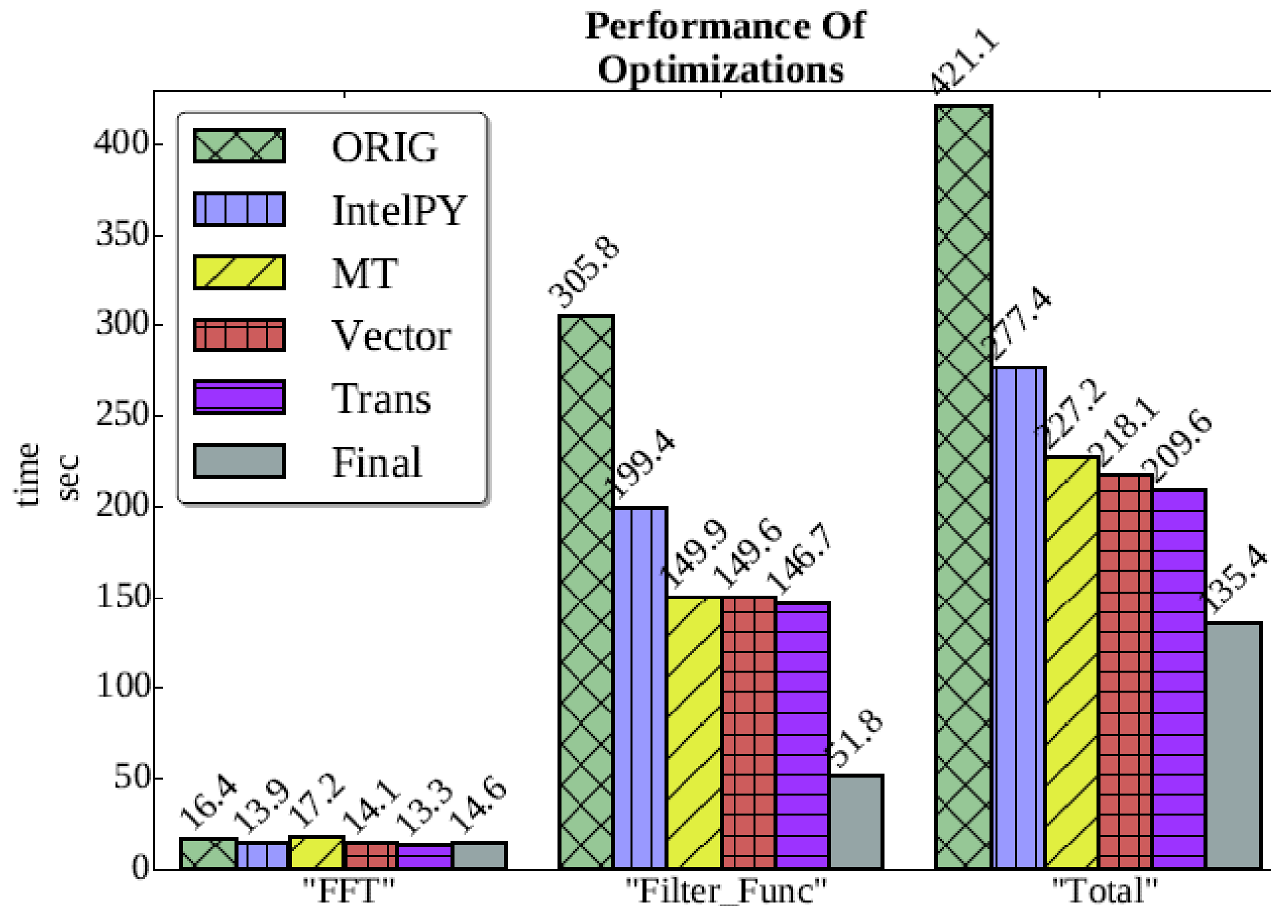
Vectorization

	Thread	Trans	Non-Trans
369 songs	28P/2T	138.72s	185.63s
	28P/1T	141.80s	220.44s
4 songs	4P/14T	9.78s	10.22s
	4P/7T	9.49s	10.35s
	4P/1T	20.02s	35.28s

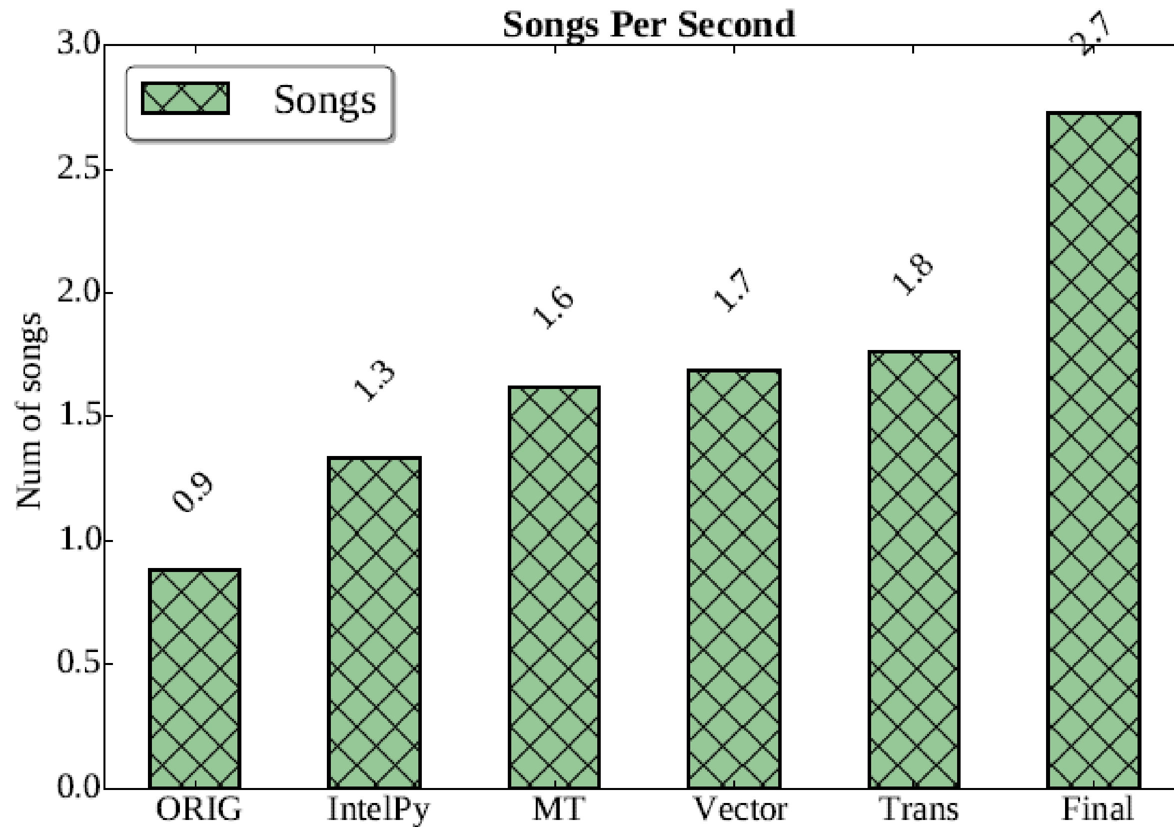
```
for i:= 0 to filter_size do
begin
{
 if (offset): update input;
 if (errors): update input;
 if (minimum):
    if (find min): update output;
 else:
    if (find max): update output;
}
end;
```

```
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 Songs per Sec



Performance analysis

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

$$P_{total} = \frac{Cores \times P_{core} \times VPUs \times l_{vec}}{S_{data}} = \frac{28 \times 2.6GHz \times 2 \times 32Bytes}{64Bytes}$$

$$= 582\ GFLOPs$$

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

$$Intensity = \frac{total\ operations}{total\ memory\ acess}$$

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