Progress Report (Redacted)

Spectral Compute

Executive Summary

- Current performance gain is 47%
- It was mentioned during the call that 20-50% would be a significant performance win, so we felt a report would be appreciated at this point.
- We refrained from using our more exotic in-house technologies for now, so all results should be deployable as-is and provide no challenges with regards to maintaining the code.
- There is plenty of room to make more improvements as-is, however any time spent would be far more productive with access to the full codebase. (Which requires formalization of NDA's, etc)
- Time spent on work is roughly 24-man hours. (We had 3 engineers spend a working-day)
- Feel free to provide any feedback and/or comments. We look forward to it.

Summary of changes

- Improved video quality and performance by encoding the video only once. Instead of writing it using opencv and transcoding it with ffmpeg, frames are now written to ffmpeg via stdin. This causes the video encoder to run in parallel with the AI, and avoids repeated compression.
- Upgraded the dependent libraries (didn't yield any speedup on its own, on our GPUs at least).
- We enabled pytorch's "inference_mode", which is a stronger from of "no_grad" (tiny speedup)
- Materialise the input transpose before invoking pytorch, so faster CUDA kernels are used.
- NVTX integration to facilitate benchmarking with nvvp.
- Enabled in-place RelU (modest speedup).
- Reduced the amount of data sent back from the GPU for lip-patching by 75% by doing the float-to-int8 conversion on the GPU. This is also faster.
- Disable bias in forward conv2d layers (it does nothing in inference mode, and is wasting time).

None of the model changes affect the output. Code available on request.

Potential future work

Tons of stuff to do here. We haven't even changed any of the underlying CUDA yet, just taken the truly low-hanging fruit.

- There are long gaps where no cuda is run. This seems to be caused by the video encoding bottlenecking (even after our improvements).
  - Enabling ffmpeg's "ultrafast" preset yields a further 6% speedup and triples output video size. This is probably not useful.
  - A project to build a CUDA-accelerated video encoder could be carried out. This would be a big job, and may be less useful than you would expect: the GPU is already busy running the neural network, but the CPU is relatively idle. There has been much more work already done in the field of optimising video encoders for CPUs. It would likely be optimal to muck about with ffmpeg-cpu to ensure it uses AVX properly, and then have it run in parallel with the neural network (parallelism already set up). That way, you'd get true parallelism between NN and encoding (compared to running them alternately, were both done on the GPU). Note that measurements show that the cost of copying the frames to/from the GPU is pretty tiny, especially because only the portion that has to be "patched" (the lips) is actually transmitted, and we reduced the size of that by 75%.
- Profiling the cuda shows that pytorch is running multiple elementwise kernels sequentially, and that these account for a significant fraction of the time. Using our own cuda stuff to generate fused kernels here would speed up the model a fair bit.
- Only one CUDA stream is used. This is pretty typical for pytorch. Rewriting the python script using C++ (and our library for doing crazy stuff with cuda streams) would allow us to do software pipelining of frames (overlapping frame copies with kernels, and overlapping execution of multiple batches). We haven't bothered yet because the expected benefit is quite small due to the low amount of time spent doing copies. The value of this work would be increased once we apply more of our custom CUDA.
- The performance benefit of rewriting the python in C++ is not expected to be large, except insofar as it allows us to schedule the cuda streams better. Now we fixed the parallelism and so on, the python isn't a bottleneck (and profiling shows that its execution time is almost entirely hidden behind cuda and ffmpeg time).
- There are plenty of more ambitious cuda-level enhancements we can do by modifying pytorch, depending on how aggressively we get into mucking about with the standard tool.
- Recompiling stuff (including pytorch) with our fancy compiler should speed things up.

Benchmarking details

Since you said the face detection stuff was not of interest (you ran it ahead of time), we changed things so the face detector is run all at once upfront, its result cached in memory, and then the lipsyncing stuff run afterwards. The original implementation interleaves lipsync and face detection invocation using an async generator (which is a better approach in general, but annoying for the specific goals of our work here). Consequently, we took our baseline measurements at 77e02e04.

We tested using a video drawn from the wav2lip demo website (the middle video). We also used a version of that video looped six times, to measure things with a longer video (though the performance changes seem to scale fairly linearly).

Timing was done using the timing code we embedded in the program, as well as the nvtx API.

Can provide the raw timing data and nvvp files, if you want. Benchmarks were carried out on a machine with an NVIDIA GeForce GTX1080 GPU, an AMD Ryzen Threadripper 3970X 32-Core Processor, and 256GB of RAM.

Lastly, we did some testing using a static image as the input video (to explore degenerate cases of some code paths). The resulting lipsynced videos are *absolutely terrifying*, and we have a few not-merged optimisations that are useful only in this situation, if you're interested in that. We are more than 2x faster in this degenerate case.