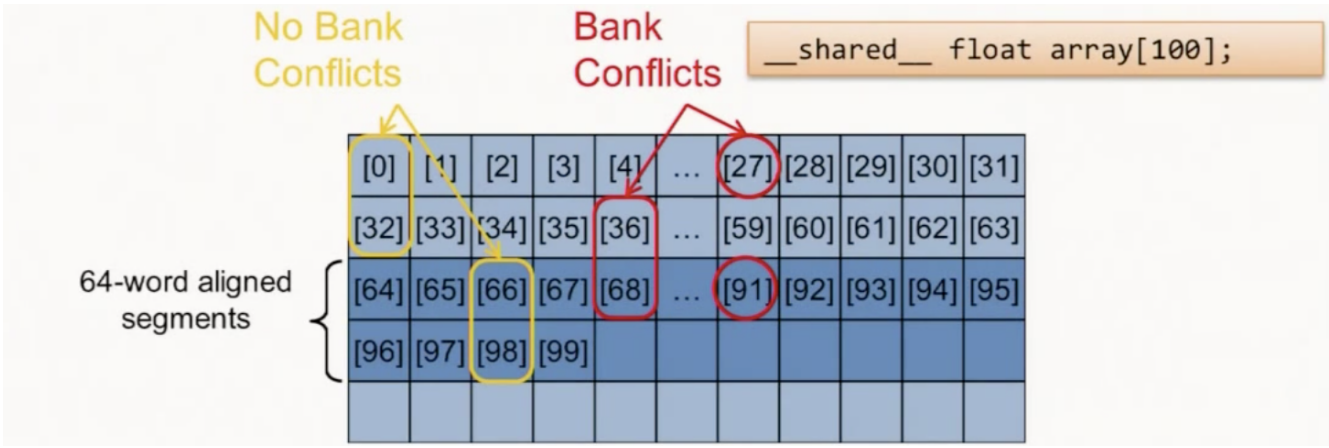
essential cuda optimization techniques

- nvvp is the NVIDIA visual profiler
- in deep learning, you should pipeline so that the next batch is being fetched while the current batch is being executed.
- you want to figure out if the current bottleneck in the pipeline is because the GPU isn't going fast enough or because the CPU isn't fetching and putting data onto the GPU fast enough.
- execute the kernels that take up most compute time first.
- three kinds of bottlenecks:
  - memory bandwidth: host ↔ device transfer bandwidth, is the amount transferred at once vs time taken by GPU batch in sync.
  - latency: due to memory latency on GPU threads (this is not host to device transfer, but local device memory accesses that are not fast register accesses)
  - compute bound → see below about common code paths and warps
- A "warp" (or "wavefront") is the most basic unit of scheduling of the NVIDIA (or AMD) GPU. Other equivalent definitions include: "is the smallest executable unit of code" OR "processes a single instruction over all of the threads in it at the same time" OR "is the minimum size of the data processed in SIMD fashion"
- a device's hardware implementation groups adjacent threads within a block into warps. A warp is considered active from the time its threads begin executing to the time when all threads in the warp have exited from the kernel. There is a maximum number of warps which can be concurrently active on a Streaming Multiprocessor (SM). Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. Occupancy varies over time as warps begin and end, and can be different for each SM.
- compute optimizations on gpu
  - the hardware is designed such that all threads in a warp have to execute the same instruction (with different operands allowed ofc). so if your consecutive threads that fall into the same warp have different code paths due to conditionals etc, this screws you over. if different threads in a warp take different code paths that is called divergence. bad!
  - so ideally, make your threads have the same code path! conditionals are bad. across different warps though, its ok to take different code paths → assuming you have thought about it enough to know how your threads get assigned to warps, which is not simple.
  - if you're compute bound and have loops, its good to write code in a way that makes it easier for compiler to unroll loops, using c++ template parameters for number of iterations or constants even, if possible.
  - fast math instructions are available for some common math functions. these approximate the actual IEEE values but can be quite a bit faster → useful in neural nets.
- memory bound optimizations:
  - however data is laid out, make sure once you bring in a cache line into the the cache the threads in your warp (consecutive threads) operate on that data before fetching the next cache line. don't make threads so that consecutive threads need an element from a different cache line. you are thrashing big time if you do that

- lay out your warps so that all threads in each warp can get all the data they need from memory with the least number of memory read transactions possible
- shared memory is split into banks. each bank has a set bandwidth for each reading transaction. so if your data is laid out such that different threads in a warp go to different banks in shared memory while accessing the data, or if they read common data from the same bank, we will get maximum efficiency of reads from shared memory. on the other hand, if multiple threads are trying to read different slots in the same bank, we are screwed. this read will take multiple clock cycles.



- biggest memory trick is to avoid frequent trips to global memory. bad bank management in shared memory is still relatively ok.