



GPU Optimization Principals

Gordon Brown & Michael Wong

CppCon 2020 – Sep 2020

- Learning objectives:
 - Learn about the different levels of GPU optimization
 - Learn about good optimization practice
 - Learn about how to pick the right algorithm
 - Learn about avoiding divergent control flow
 - Learn about ensuring coalesced global memory access
 - Learn about vectorization

Add a topic

Yesterday ▾

The class is doing a great job at introducing the what kinds of things we can do using SYCL and how they can be done. I'm having a hard time putting things into perspective on when they should be used relative to just doing it all in a CPU sequentially. Are there any resources I can look at that might help answer questions such as (listed in the thread):


 16 replies Last reply 15 hours ago

Out of curiosity, is it possible to query for kind of "current timestamp" in the sycl kernel and pass it to the host (kind of `std::chrono::clock_type::now()` but on the device) ? I mean to use this "device timestamp" to check which of the kernels B / C executed first.

  **8 replies** Last reply 20 hours ago

so to call back to the earlier question about polling for the status of an `event` you can do this by calling `e.get_info<info::event::command_execution_status>()` and this will return `event_command_status` which can be either `submitted`, `running` or `complete`

 Next lecture is 12:50 MDT.

 quick note there have been some minor modifications to the materials in the repository and I have pushed the presentations for this morning so you should pull to get these changes

M are the slides for the first presentation on monday available somewhere?

M im guessing if its anywhere its in the repo

 **M** 2 replies Last reply 19 hours ago

0 1 11 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78 81 84 87 90 93 96 99 102 105 108 111 114 117 120 123 126 129 132 135 138 141 144 147 150 153 156 159 162 165 168 171 174 177 180 183 186 189 192 195 198 201 204 207 210 213 216 219 222 225 228 231 234 237 240 243 246 249 252 255 258 261 264 267 270 273 276 279 282 285 288 291 294 297 300 303 306 309 312 315 318 321 324 327 330 333 336 339 342 345 348 351 354 357 360 363 366 369 372 375 378 381 384 387 390 393 396 399 402 405 408 411 414 417 420 423 426 429 432 435 438 441 444 447 450 453 456 459 462 465 468 471 474 477 480 483 486 489 492 495 498 501 504 507 510 513 516 519 522 525 528 531 534 537 540 543 546 549 552 555 558 561 564 567 570 573 576 579 582 585 588 591 594 597 600 603 606 609 612 615 618 621 624 627 630 633 636 639 642 645 648 651 654 657 660 663 666 669 672 675 678 681 684 687 690 693 696 699 702 705 708 711 714 717 720 723 726 729 732 735 738 741 744 747 750 753 756 759 762 765 768 771 774 777 780 783 786 789 792 795 798 801 804 807 810 813 816 819 822 825 828 831 834 837 840 843 846 849 852 855 858 861 864 867 870 873 876 879 882 885 888 891 894 897 900 903 906 909 912 915 918 921 924 927 930 933 936 939 942 945 948 951 954 957 960 963 966 969 972 975 978 981 984 987 990 993 996 999 1002 1005 1008 1011 1014 1017 1020 1023 1026 1029 1032 1035 1038 1041 1044 1047 1050 1053 1056 1059 1062 1065 1068 1071 1074 1077 1080 1083 1086 1089 1092 1095 1098 1101 1104 1107 1110 1113 1116 1119 1122 1125 1128 1131 1134 1137 1140 1143 1146 1149 1152 1155 1158 1161 1164 1167 1170 1173 1176 1179 1182 1185 1188 1191 1194 1197 1200 1203 1206 1209 1212 1215 1218 1221 1224 1227 1230 1233 1236 1239 1242 1245 1248 1251 1254 1257 1260 1263 1266 1269 1272 1275 1278 1281 1284 1287 1290 1293 1296 1299 1302 1305 1308 1311 1314 1317 1320 1323 1326 1329 1332 1335 1338 1341 1344 1347 1350 1353 1356 1359 1362 1365 1368 1371 1374 1377 1380 1383 1386 1389 1392 1395 1398 1401 1404 1407 1410 1413 1416 1419 1422 1425 1428 1431 1434 1437 1440 1443 1446 1449 1452 1455 1458 1461 1464 1467 1470 1473 1476 1479 1482 1485 1488 1491 1494 1497 1500 1503 1506 1509 1512 1515 1518 1521 1524 1527 1530 1533 1536 1539 1542 1545 1548 1551 1554 1557 1560 1563 1566 1569 1572 1575 1578 1581 1584 1587 1590 1593 1596 1599 1602 1605 1608 1611 1614 1617 1620 1623 1626 1629 1632 1635 1638 1641 1644 1647 1650 1653 1656 1659 1662 1665 1668 1671 1674 1677 1680 1683 1686 1689 1692 1695 1698 1701 1704 1707 1710 1713 1716 1719 1722 1725 1728 1731 1734 1737 1740 1743 1746 1749 1752 1755 1758 1761 1764 1767 1770 1773 1776 1779 1782 1785 1788 1791 1794 1797 1800 1803 1806 1809 1812 1815 1818 1821 1824 1827 1830 1833 1836 1839 1842 1845 1848 1851 1854 1857 1860 1863 1866 1869 1872 1875 1878 1881 1884 1887 1890 1893 1896 1899 1902 1905 1908 1911 1914 1917 1920 1923 1926 1929 1932 1935 1938 1941 1944 1947 1950 1953 1956 1959 1962 1965 1968 1971 1974 1977 1980 1983 1986 1989 1992 1995 1998 2001 2004 2007 2010 2013 2016 2019 2022 2025 2028 2031 2034 2037 2040 2043 2046 2049 2052 2055 2058 2061 2064 2067 2070 2073 2076 2079 2082 2085 2088 2091 2094 2097 2100 2103 2106 2109 2112 2115 2118 2121 2124 2127 2130 2133 2136 2139 2142 2145 2148 2151 2154 2157 2160 2163 2166 2169 2172 2175 2178 2181 2184 2187 2190 2193 2196 2199 2202 2205 2208 2211 2214 2217 2220 2223 2226 2229 2232 2235 2238 2241 2244 2247 2250 2253 2256 2259 2262 2265 2268 2271 2274 2277 2280 2283 2286 2289 2292 2295 2298 2301 2304 2307 2310 2313 2316 2319 2322 2325 2328 2331 2334 2337 2340 2343 2346 2349 2352 2355 2358 2361 2364 2367 2370 2373 2376 2379 2382 2385 2388 2391 2394 2397 2400 2403 2406 2409 2412 2415 2418 2421 2424 2427 2430 2433 2436 2439 2442 2445 2448 2451 2454 2457 2460 2463 2466 2469 2472 2475 2478 2481 2484 2487 2490 2493 2496 2499 2502 2505 2508 2511 2514 2517 2520 2523 2526 2529 2532 2535 2538 2541 2544 2547 2550 2553 2556 2559 2562 2565 2568 2571 2574 2577 2580 2583 2586 2589 2592 2595 2598 2601 2604 2607 2610 2613 2616 2619 2622 2625 2628 2631 2634 2637 2640 2643 2646 2649 2652 2655 2658 2661 2664 2667 2670 2673 2676 2679

Message class_brown_wong

⚡ B I S Ⓢ Ⓜ Ⓜ Ⓜ Ⓜ

Aa @ ☺ 📎 ➤

class brown wong

- Host allocates memory on device
- Host copies data from the app in host mem to the allocated device mem
- Host copies over and invokes a kernel on the device
- Host copies data from the allocated device memory back to the host memory to be read by the host app

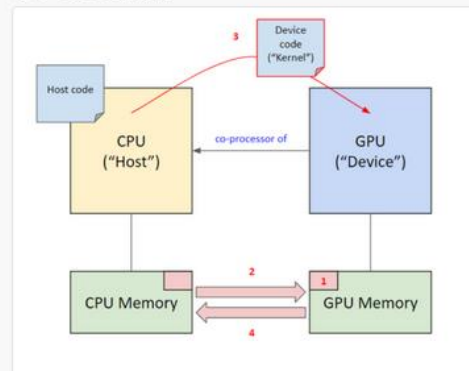
Are there any referential data points on what the potential range for the overhead of each of these steps can be? If it's something trivial like an single integer and the calc, how long does each step take? Which of the steps take more of the time?

In the code examples we've been doing (like scalar add) - how big does the data have to be before you start seeing performance gains? (including the time for the data transfer + blocking waiting)

Similarly with USM - it seems pretty neat. When would you want to use it over the accessor model when it comes to performance? Are there bounds at which one is better than the other?

I don't really have a grasp on the order of magnitude that these steps take (nanos, mics, millis, etc) and I'm not really sure how to think about the trade offs here.

CPU - GPU steps.png ▼



Those are some very good questions, but for the most part it comes down to the hardware and the problem. Since the GPU acts as a co-processor to the CPU you are very likely to see overhead from communication latency between the CPU and the GPU as well as a transfer overhead in many cases. Pin-pointing numbers for that is generally difficult, but a good rule of thumb is that if you can think of a way to formulate your problem using some of the ideas Michael talked about earlier, then chances are that, with enough data, you can benefit from using a GPU. Sorry for being a bit hand-wavy.

Notice also that GPUs, due to their various uses and general structure, have some interesting optimization you can do. I believe @Gordon Brown will be talking about some of these tomorrow.

R Hmm, that makes sense. Is there any data for any cpu / gpu combination? Would be nice to see relative values or getting a feel for the

Answer

- There is no silver bullet, but there are fundamentals that experienced practitioners reach for ...

There are different levels of optimizations you can apply

- Choosing the right algorithm
 - *This means choosing an algorithm that is well suited to parallelism*
- Basic GPU programming principles
 - *Such as coalescing global memory access or using local memory*
- Architecture specific optimisations
 - *Optimising for register usage or avoiding bank conflicts*
- Micro-optimisations
 - *Such as floating point dnorm hacks*

There are different levels of optimizations you can apply

- Choosing the right algorithm
 - *This means choosing an algorithm that is well suited to parallelism*
- Basic GPU programming principles
 - *Such as coalescing global memory access or using local memory*
- Architecture specific optimisations
 - *Optimising for register usage or avoiding bank conflicts*
- Micro-optimisations
 - *Such as floating point dnorm hacks*

This class will focus on these two

Choosing the right algorithm

What to parallelise on a GPU

- Find hotspots in your code base
 - *Looks for areas of your codebase that are bottlenecks that are taking up a lot of execution time*
- Look for areas of your codebase that are a good fit for GPU parallelism
 - *Or areas of you codebase that could be adapted for parallelism*

Follow good optimisation practice

- Follow an adaptive optimisation approach such as APOD
 - *Analyse your code base to find opportunities for parallelism*
 - *Parallelise a piece of code on the GPU*
 - *Optimise the algorithm to be more efficient on the GPU*
 - *Deploy the codebase to evaluate the performance with different inputs*
- Avoid over-optimisation
 - *You may reach a point where optimisations provide diminishing returns*
 - *It may be more efficient to look for another bottleneck in your codebase*

What to look for in an algorithm

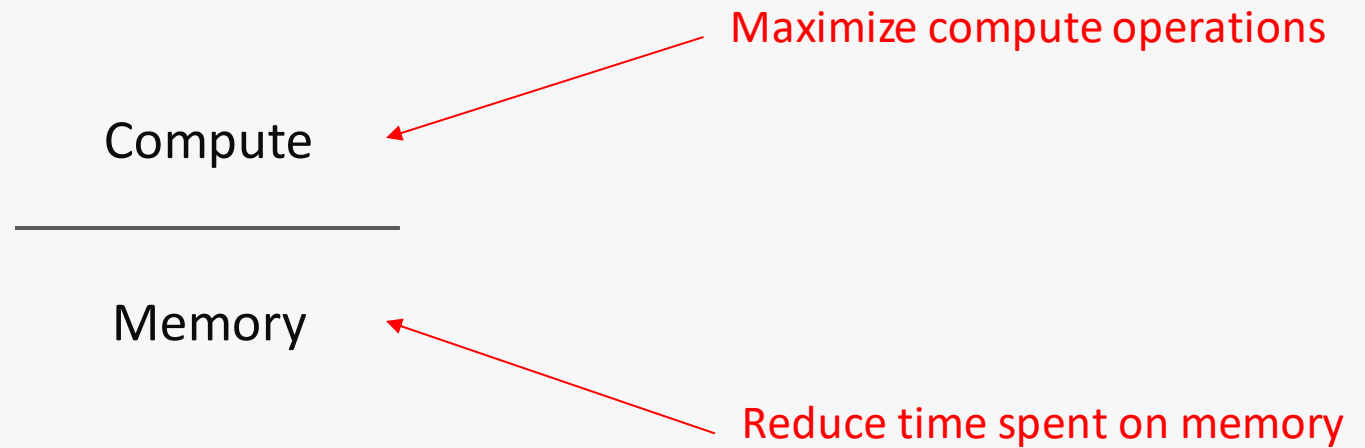
- Naturally data parallel
 - *Performing the same operation on multiple items in the computation*
- Large problem
 - *Enough work to utilise the GPU's processing elements*
- Independent progress
 - *Little or no dependencies between items in the computation*
- Non-divergent control flow
 - *Little or no branch or loop divergence*

Embarrassingly parallel algorithms

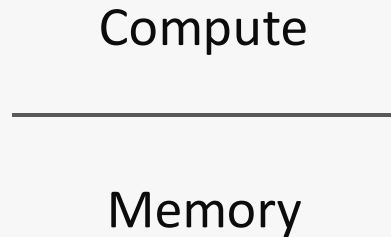
- Some problems are considered “embarrassingly parallel”
 - *The problem is naturally parallel*
 - *The problem has no communication between items in the computation*
- These kind of problems are perfect for the GPU

Basic GPU programming principles

Optimizing GPU programs means maximizing occupancy and throughput

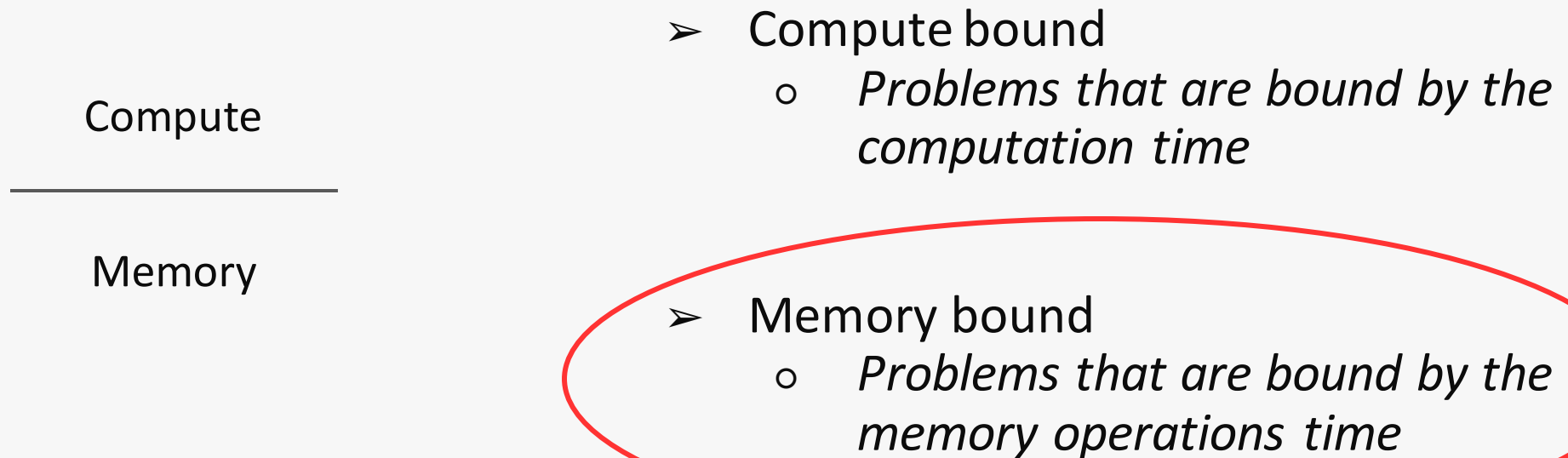


Optimizing GPU programs means maximizing throughput



- Compute bound
 - *Problems that are bound by the computation time*
- Memory bound
 - *Problems that are bound by the memory operations time*

Optimizing GPU programs means maximizing throughput



Most GPU problems
are memory bound

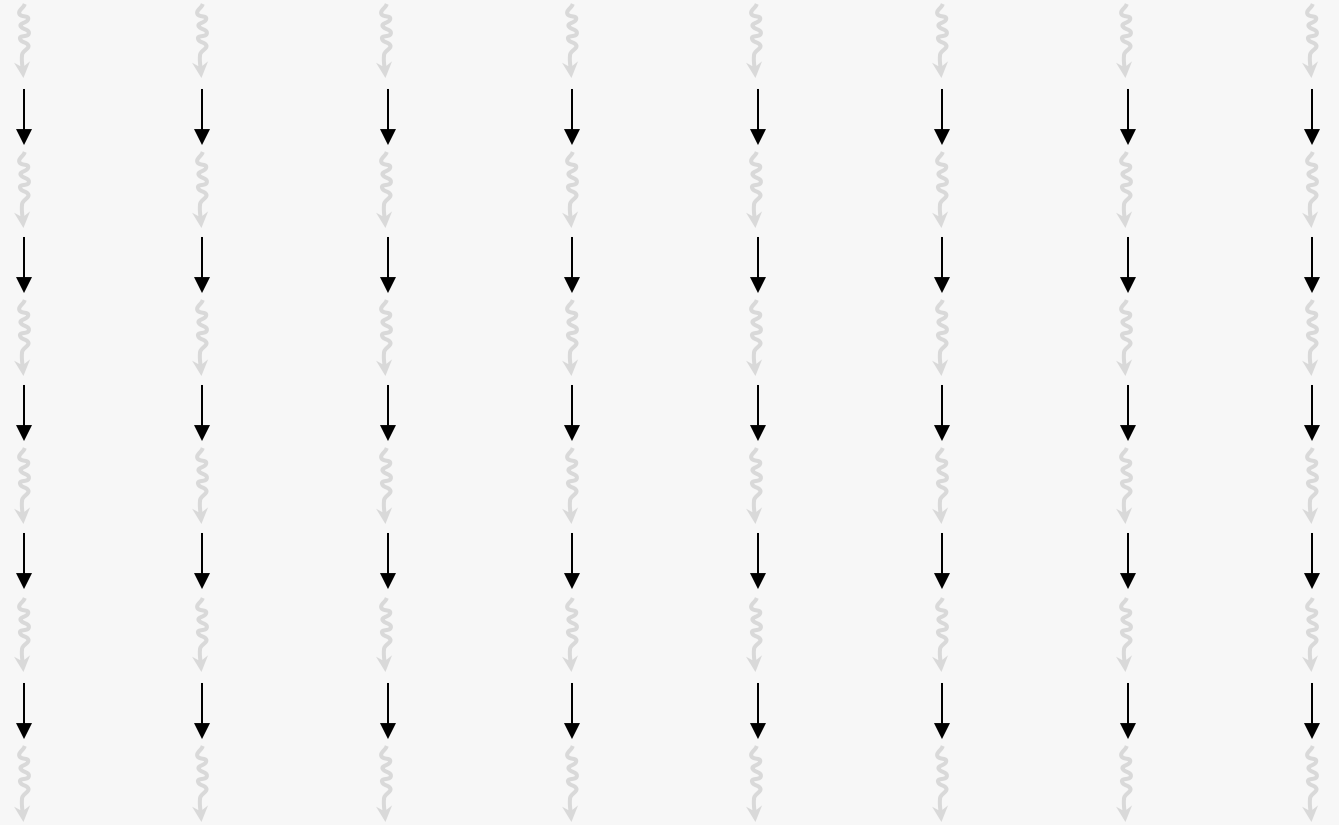
Optimizing GPU programs means maximizing occupancy and throughput

- Maximise compute operations per cycle
 - *Make effective utilisation of the GPU's hardware*
- Reduce time spent on memory operations
 - *Reduce latency of memory access*

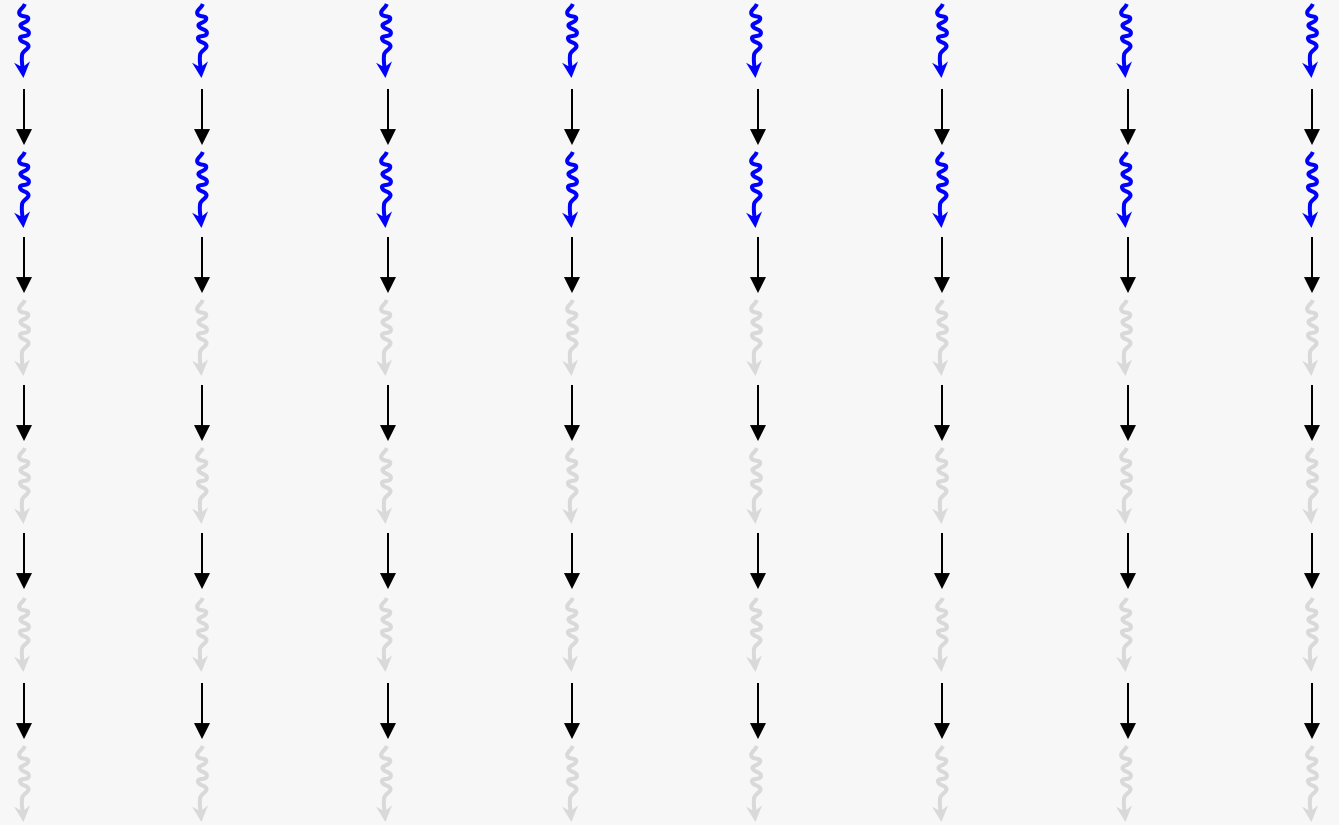
Avoid divergent control flow

- Divergent branches and loops can cause inefficient utilisation
 - *If consecutive work-items execute different branches they must execute separate instructions*
 - *If some work-items execute more iterations of a loop than neighbouring work-items this leaves them doing nothing*

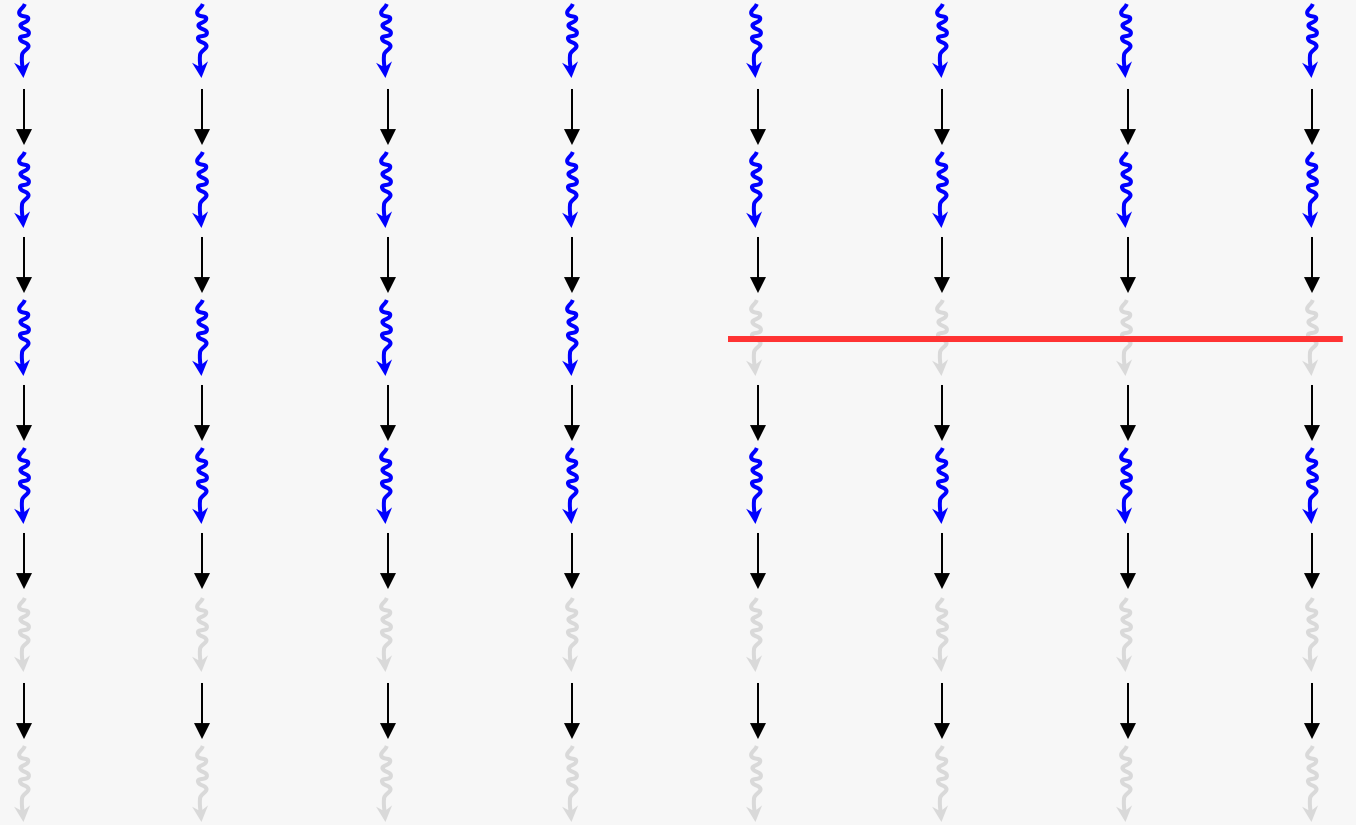
```
a[globalId] = 0;  
  
if (globalId < 4) {  
    a[globalId] = x();  
} else {  
    a[globalId] = y();  
}
```



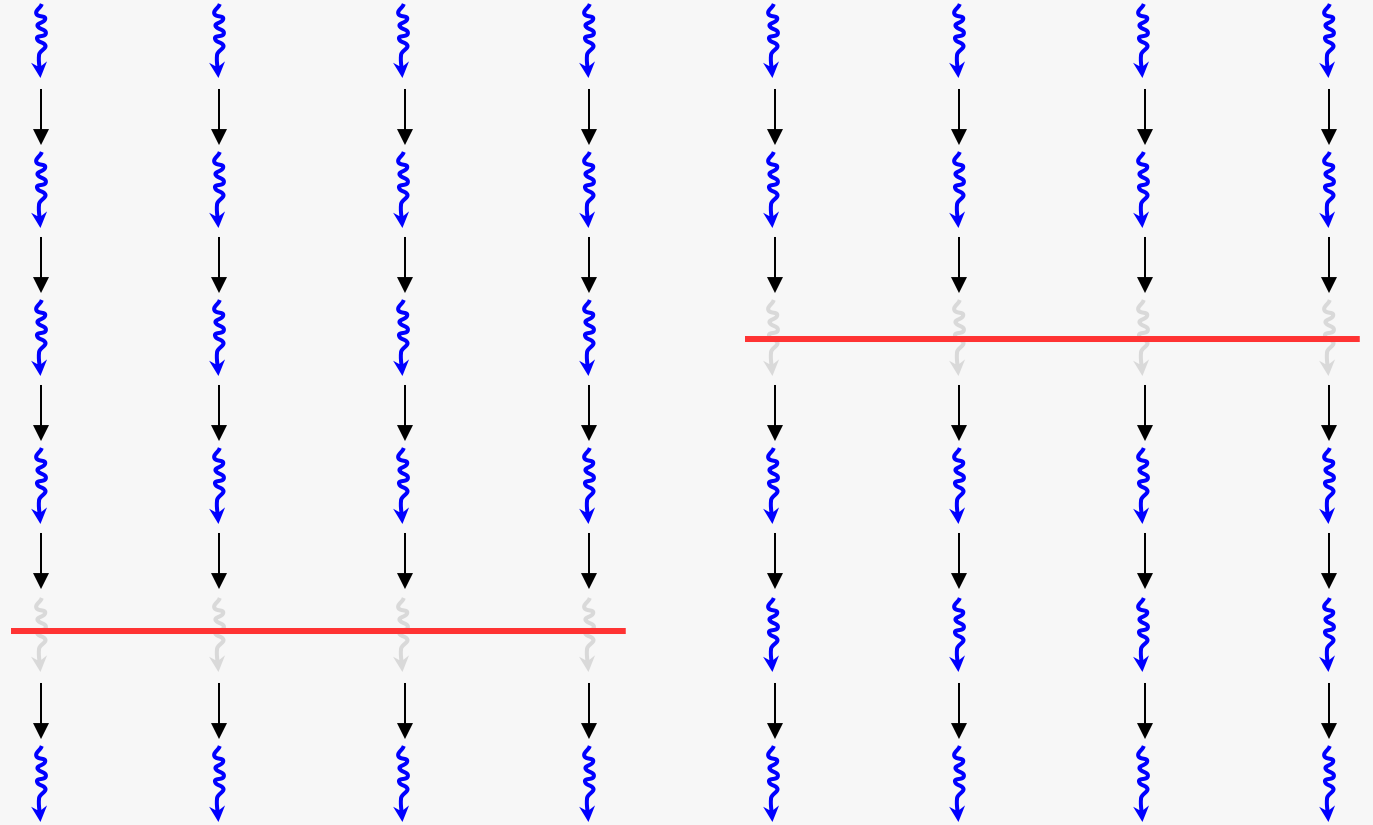
```
a[globalId] = 0;  
  
if (globalId < 4) {  
    a[globalId] = x();  
} else {  
    a[globalId] = y();  
}
```



```
a[globalId] = 0;  
  
if (globalId < 4) {  
    a[globalId] = x();  
} else {  
    a[globalId] = y();  
}
```



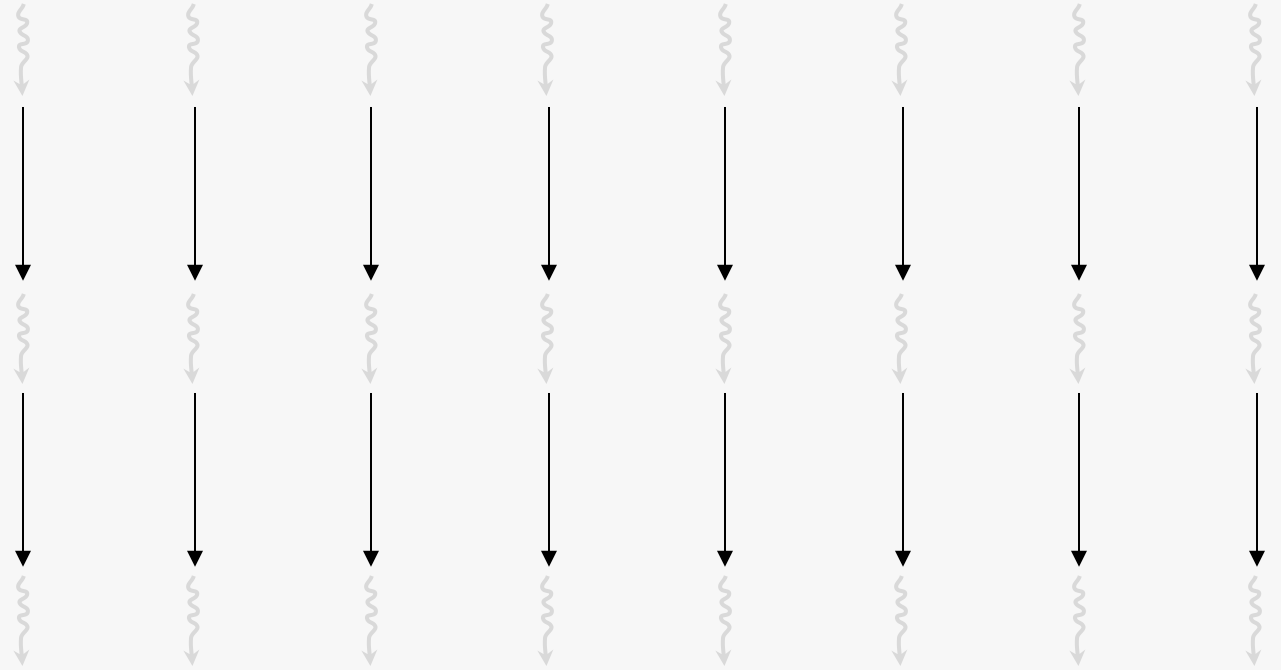
```
a[globalId] = 0;  
  
if (globalId < 4) {  
    a[globalId] = x();  
} else {  
    a[globalId] = y();  
}
```



...

```
for (int i = 0; i <
    globalId; i++) {
    do_something();
}
```

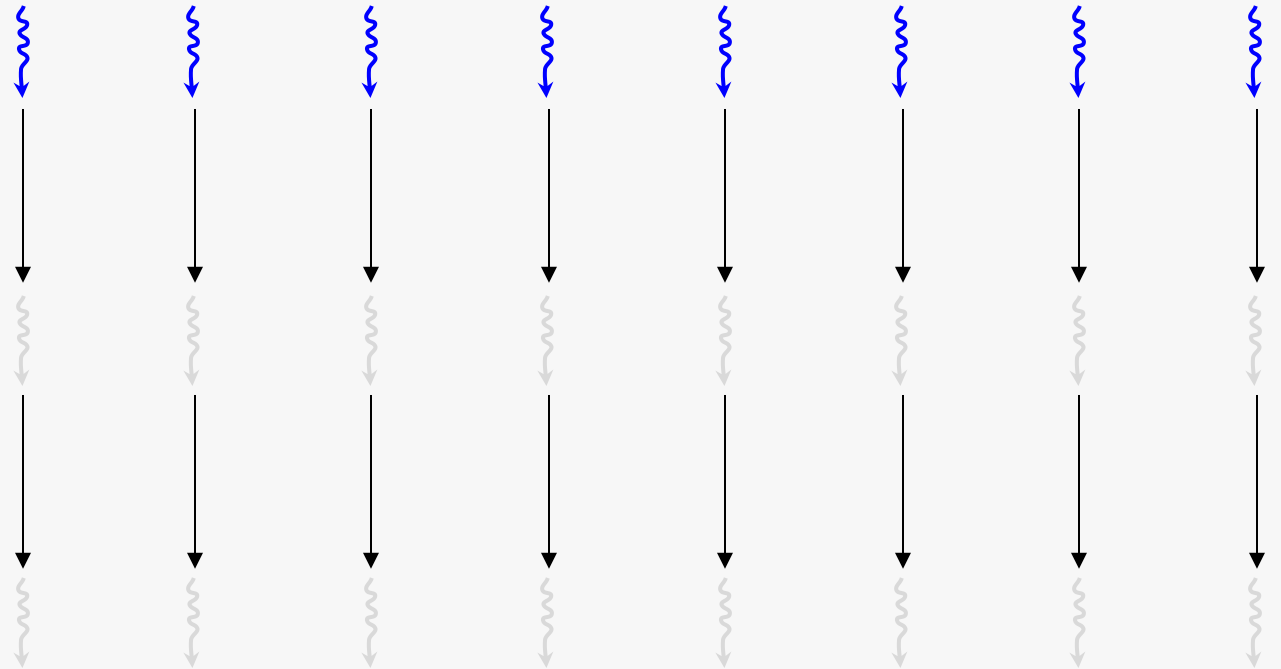
...



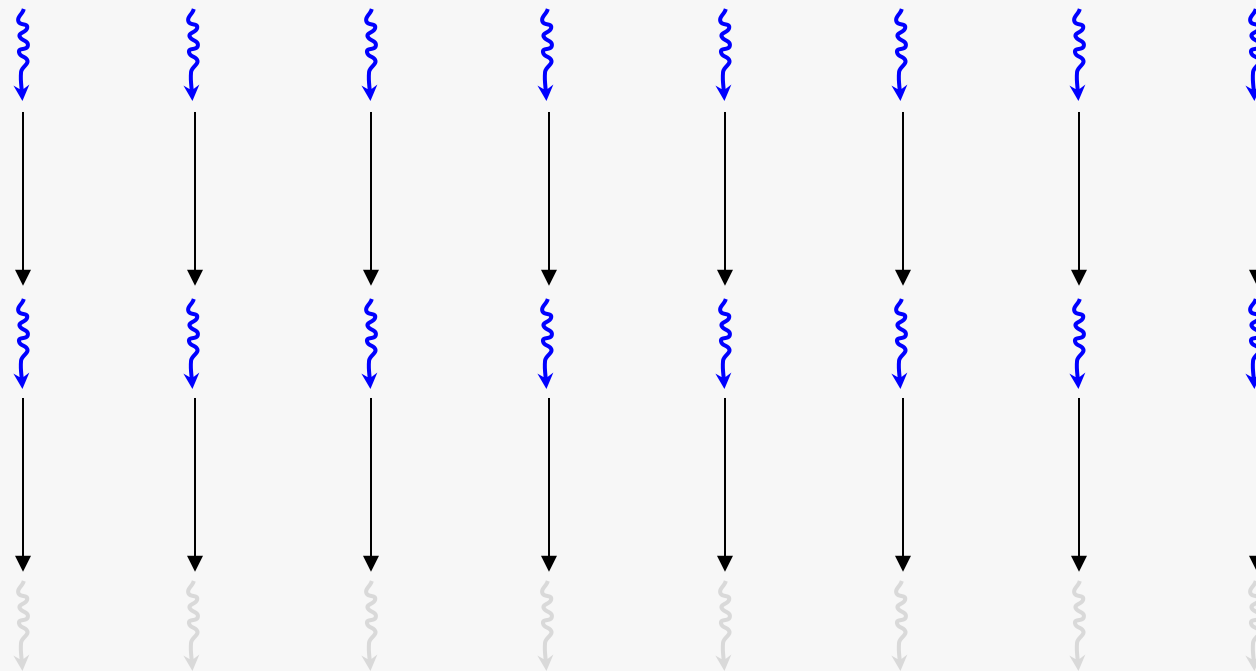
...

```
for (int i = 0; i <
    globalId; i++) {
    do_something();
}
```

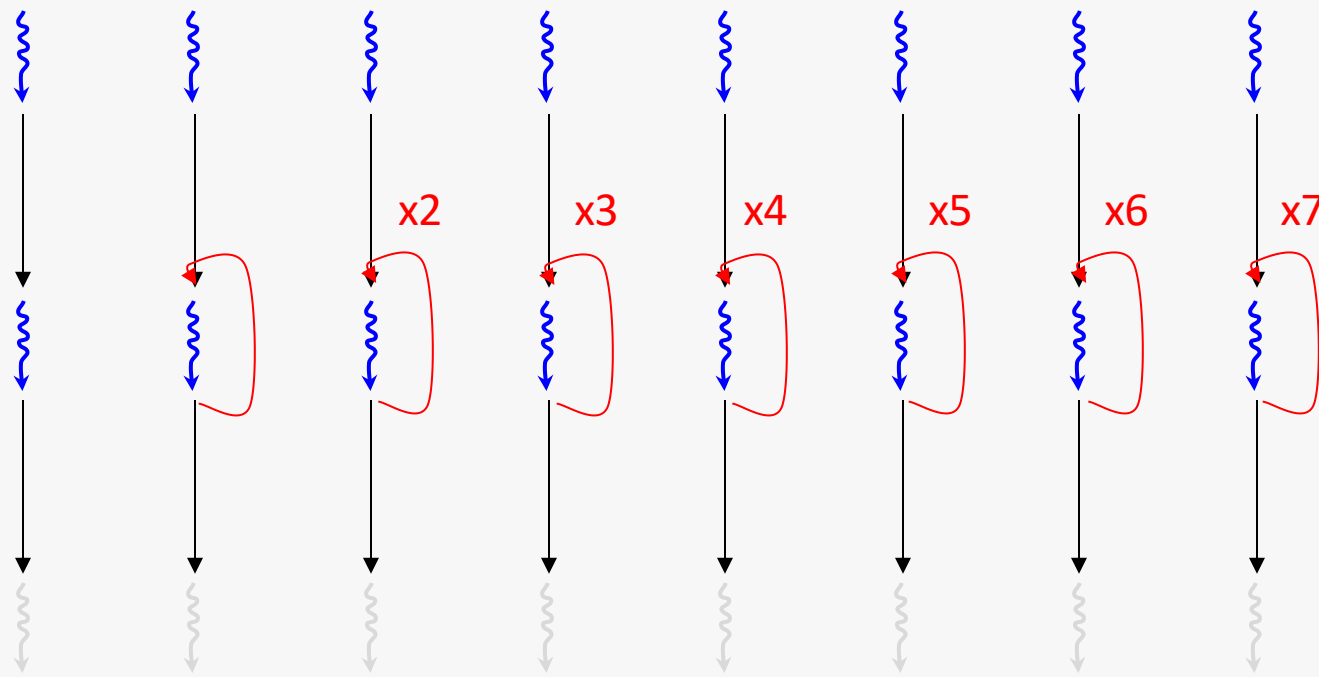
...



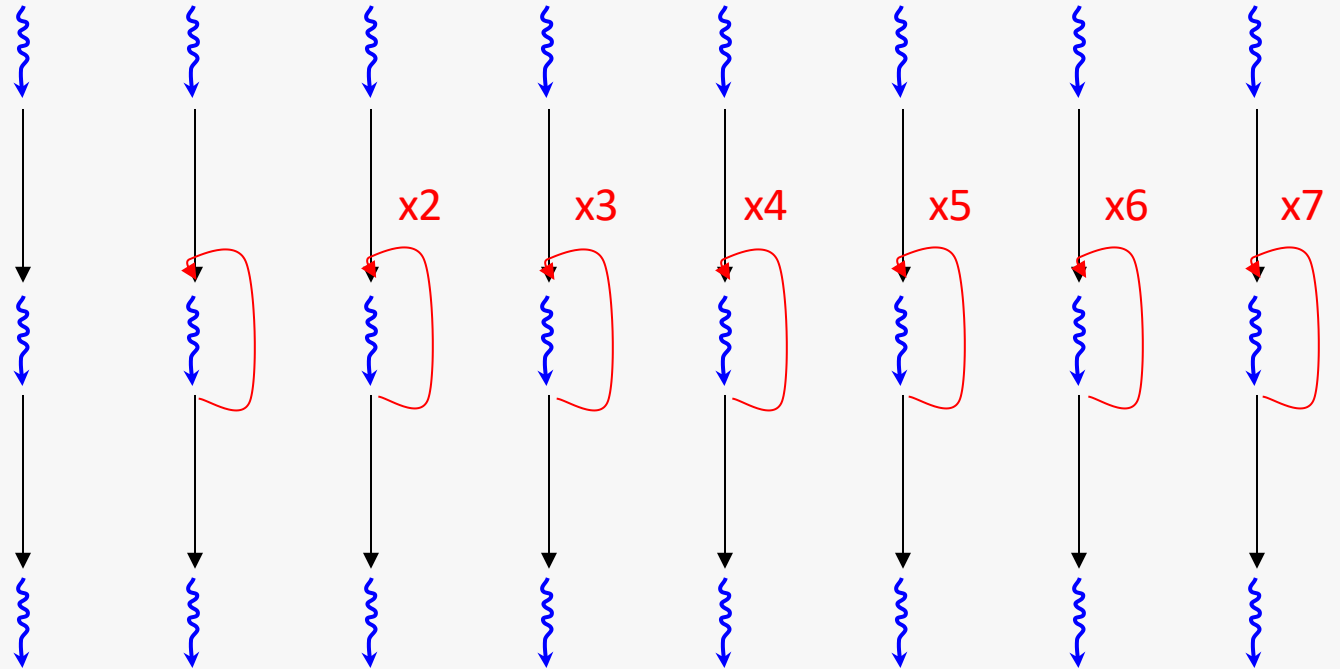
```
...  
  
for (int i = 0; i <  
    globalId; i++) {  
    do_something();  
}  
  
...
```




```
...  
  
for (int i = 0; i <  
    globalId; i++) {  
    do_something();  
}  
  
...
```



```
...  
  
for (int i = 0; i <  
    globalId; i++) {  
    do_something();  
}  
  
...
```



Coalesced global memory access

- Reading and writing from global memory is very expensive
 - *It often means copying across an off-chip bus*
- Reading and writing from global memory is done in strides
 - *This means accessing data that is physically close together in memory is more efficient*

Make use of vector operations

- GPUs are vector processors
 - *Each processing element is capable of wide instructions which can operate on multiple elements of data at once*
- Many compilers can auto-vectorise
 - *This can affect the amount of performance gain you may see in vectorising your kernels*

Make use of local memory

- Local memory is much lower latency to access than global memory
 - *Cache commonly accessed data and temporary results in local memory rather than reading and writing to global memory*
- Using local memory is not necessarily always more efficient
 - *If data is not accessed frequently enough to warrant the copy to local memory you may not see a performance gain*

Synchronise work-groups when necessary

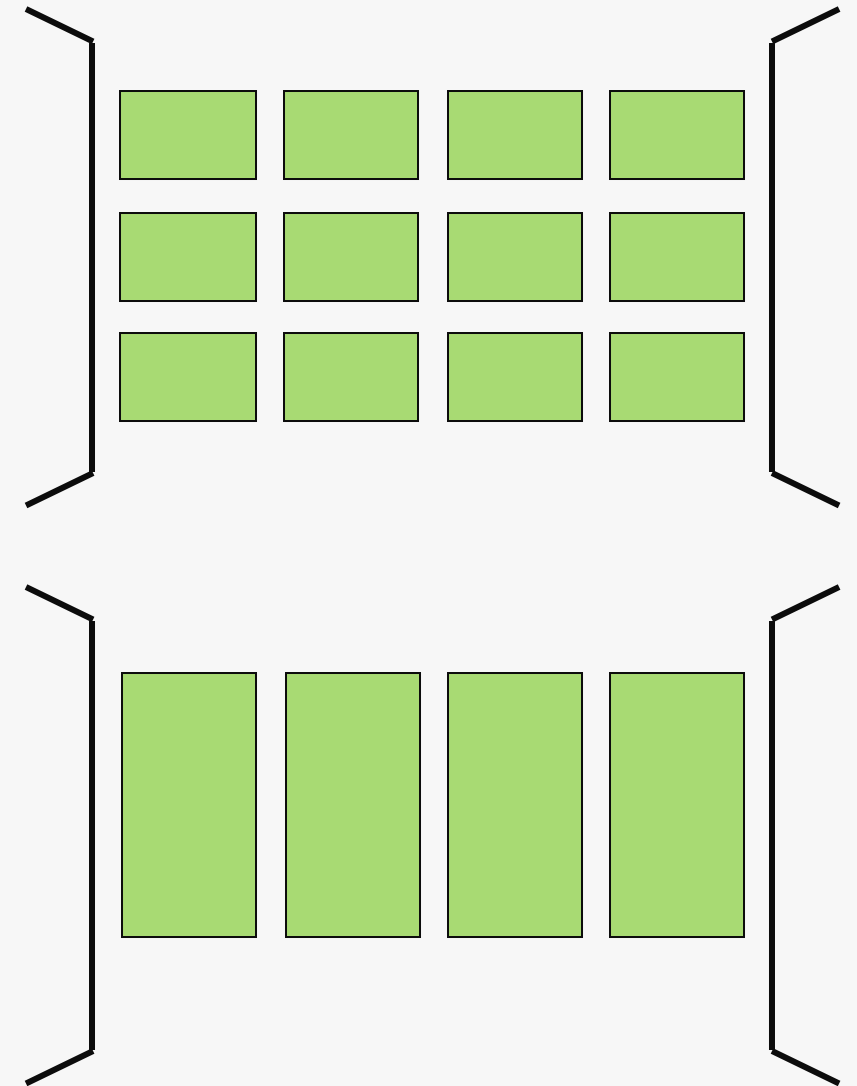
- Synchronising with a work-group barrier waits for all work-items to reach the same point
 - *Use a work-group barrier if you are copying data to local memory that neighbouring work-items will need to access*
 - *Use a work-group barrier if you have temporary results that will be shared with other work-items*

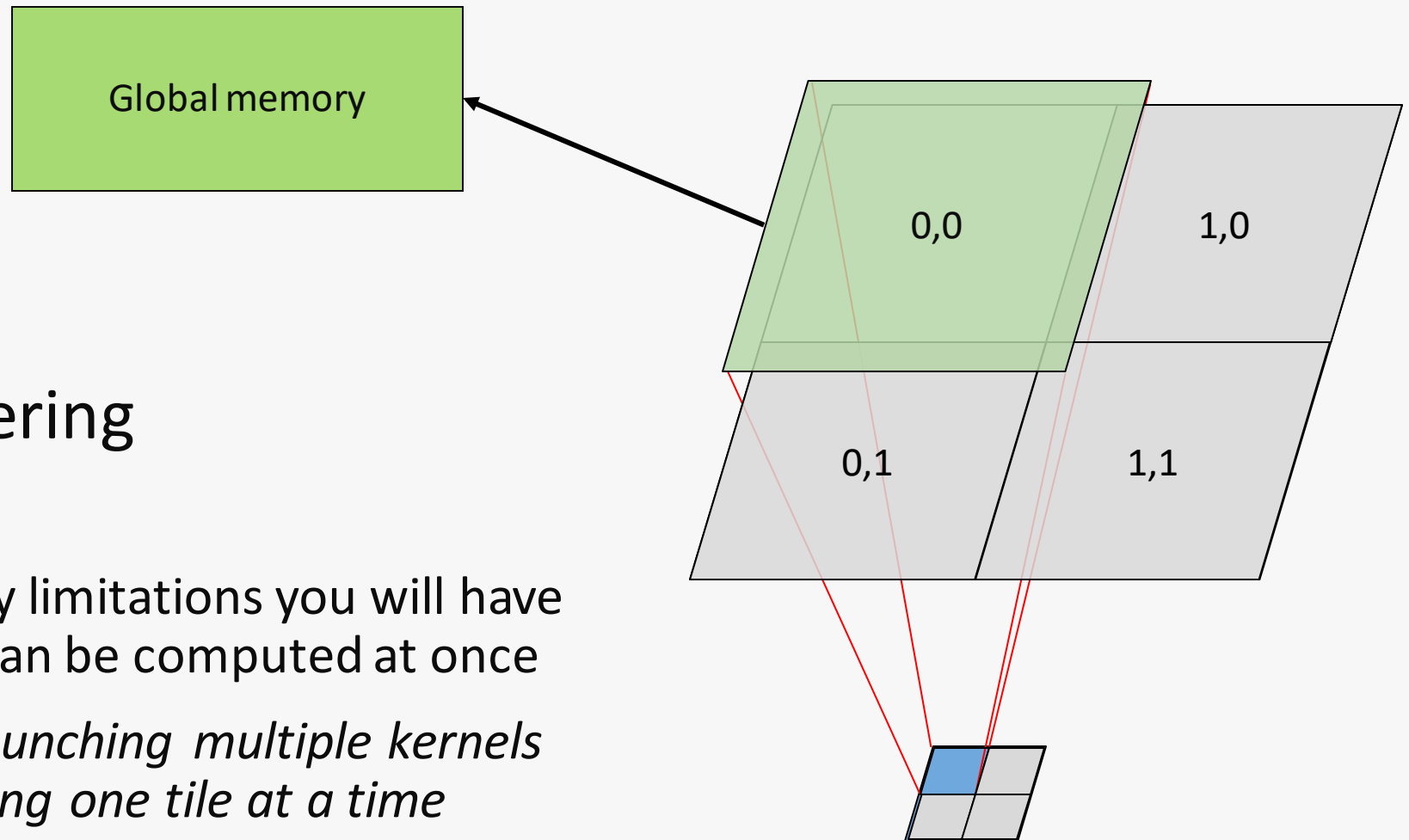
Choosing a good work-group size

- The occupancy of a kernel can be limited by a number of factors of the GPU
 - *Processing elements, compute units, local memory, registers*
- You can query the preferred work-group size once the kernel is compiled
 - *However this is not guaranteed to give you the best performance*
- It's good practice to benchmark various work-group sizes and choose the best
 - *Different algorithms will work better with different work-group sizes depending on their need for local memory and registers*

Batch work together

- Hitting occupancy limitations of a GPU can *lead to drops in performance gain*
 - *This is because single work-items are having to do more chunks of work*
- *Batching work for each work-item allows reusing cached data*
 - *Batching work that share neighbouring data allows you to further share local memory and registers*





Use double buffering

- If you hit memory limitations you will have more data than can be computed at once
 - *This means launching multiple kernels each computing one tile at a time*
- If you double buffer then you can hide the latency of data movement

Further tips

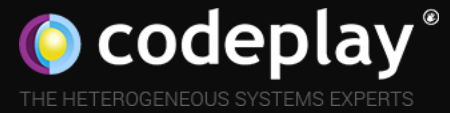
- Use profiling tools to gather more accurate information about your programs
 - *SYCL provides kernel profiling*
 - *Most OpenCL implementations provide proprietary profiler tools*
- Follow vendor optimisation guides
 - *Most OpenCL vendors provide optimisation guides that detail recommendations on how to optimise programs for their respective GPU*

Key takeaways

Porting to the GPU can give significant performance gain providing the problem is well suited to GPU parallelism

Reduce time spent on memory operations by using lower latency memory and optimizing the access patterns

Be aware of occupancy limitations and apply optimization to increase occupancy and hide the latency of data movement



Questions?