Grid-Stride Loops

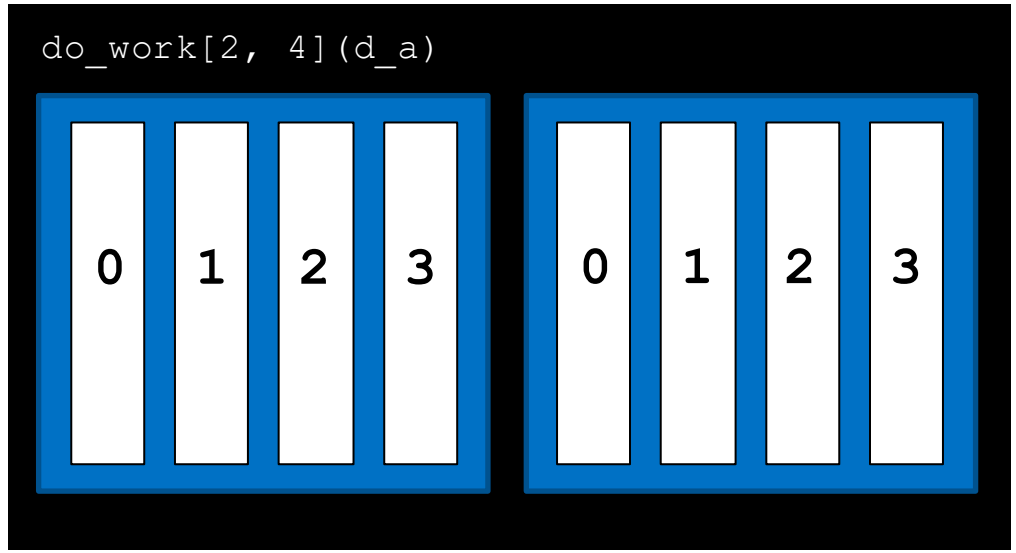
GPU
DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

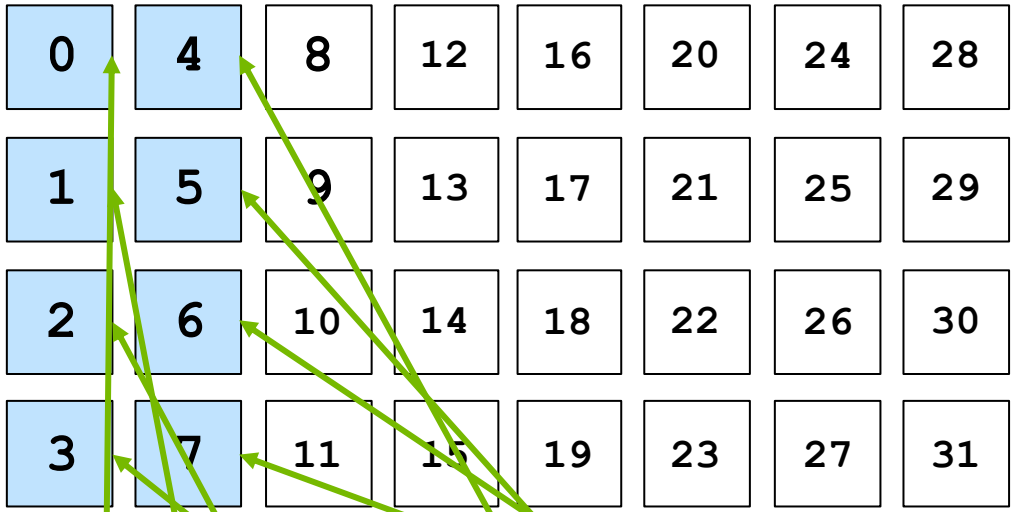
Often there are more data elements than there are threads in the grid

---

GPU

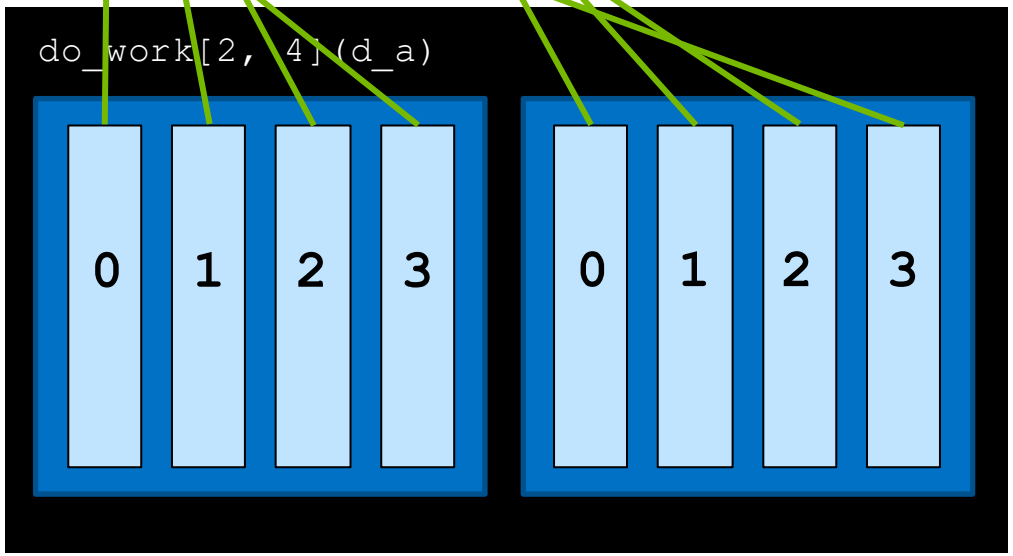


GPU
DATA



In such scenarios threads cannot work on only one element

GPU

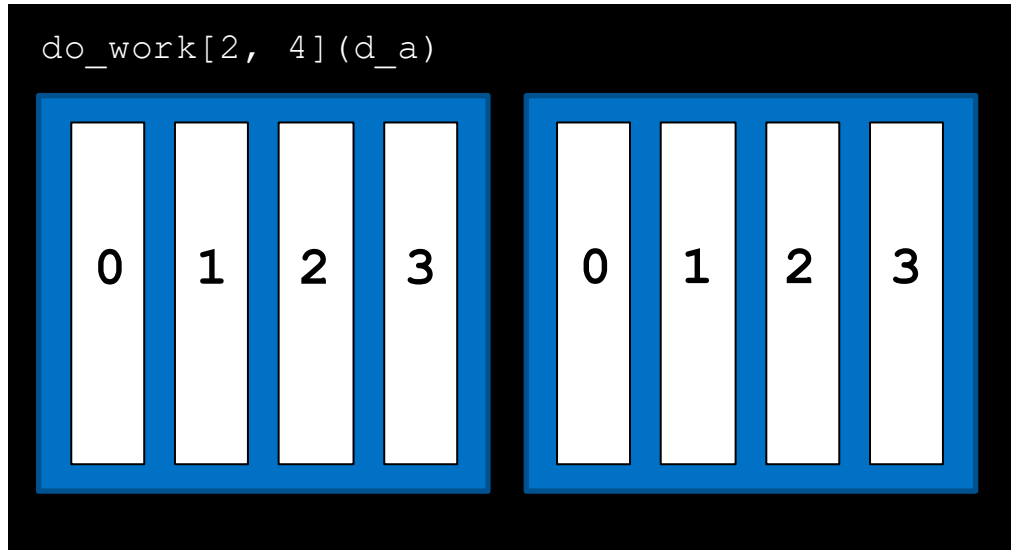


GPU
DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

… or else work is left undone

GPU

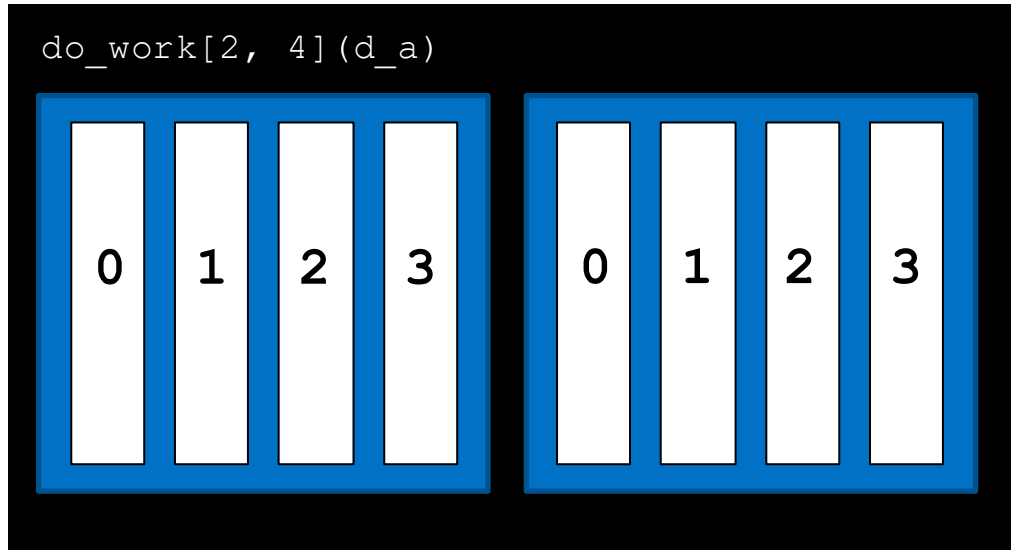


GPU
DATA

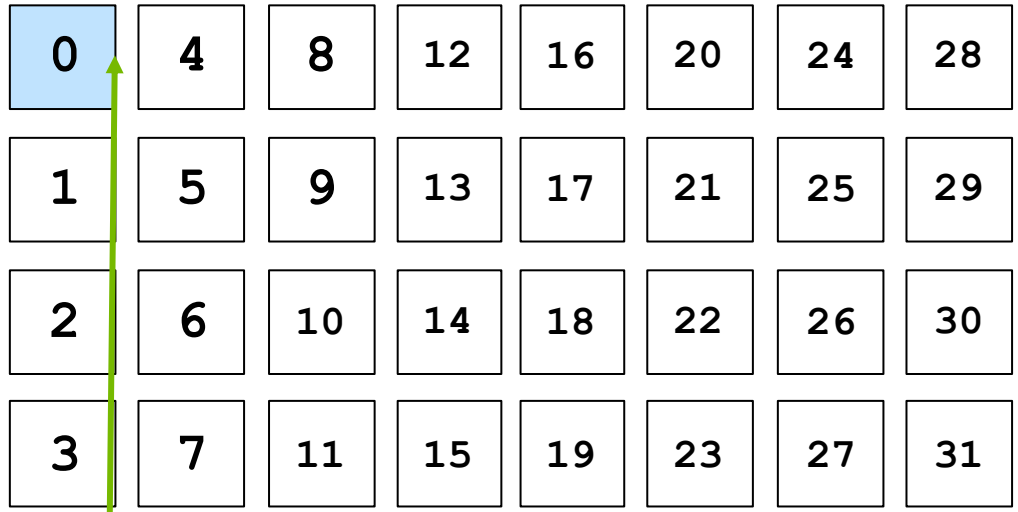
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

One way to address this programmatically is with a **grid-stride loop**

GPU

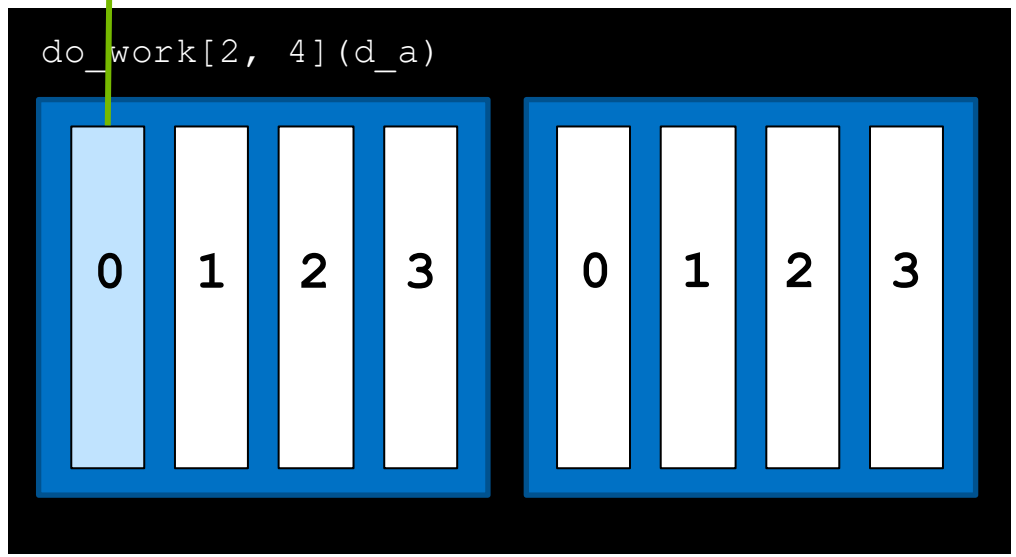


GPU
DATA

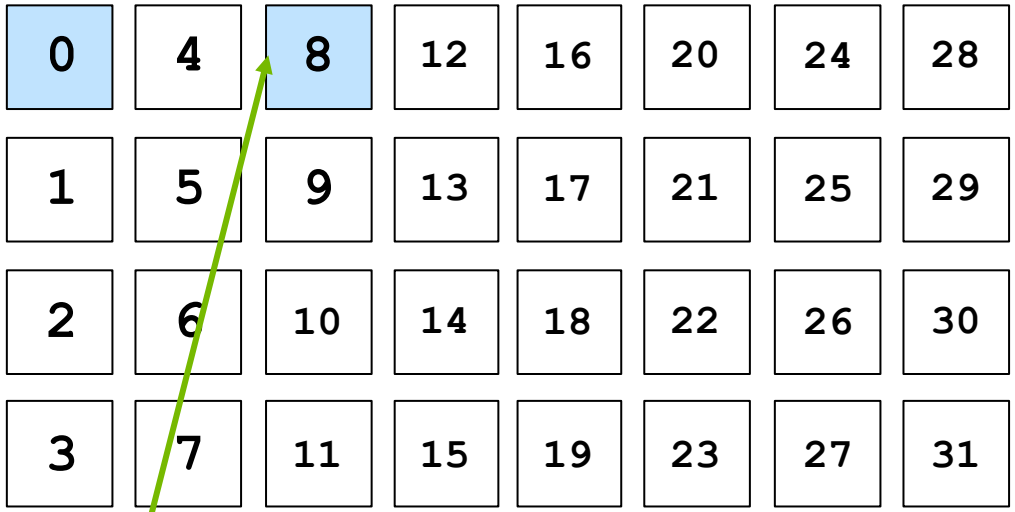


In a grid-stride loop, the thread's first element is calculated as usual, with **`cuda.grid()`**

GPU

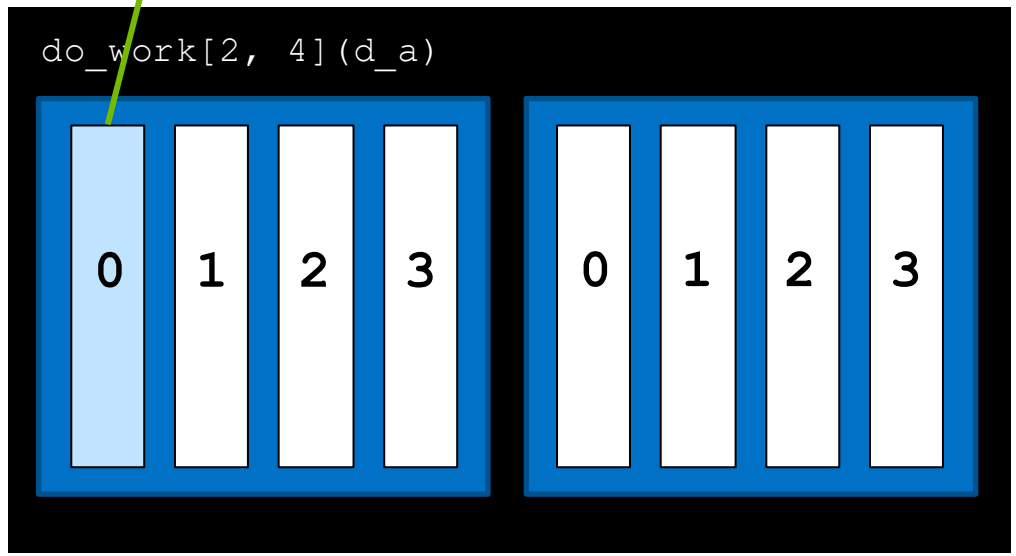


GPU
DATA

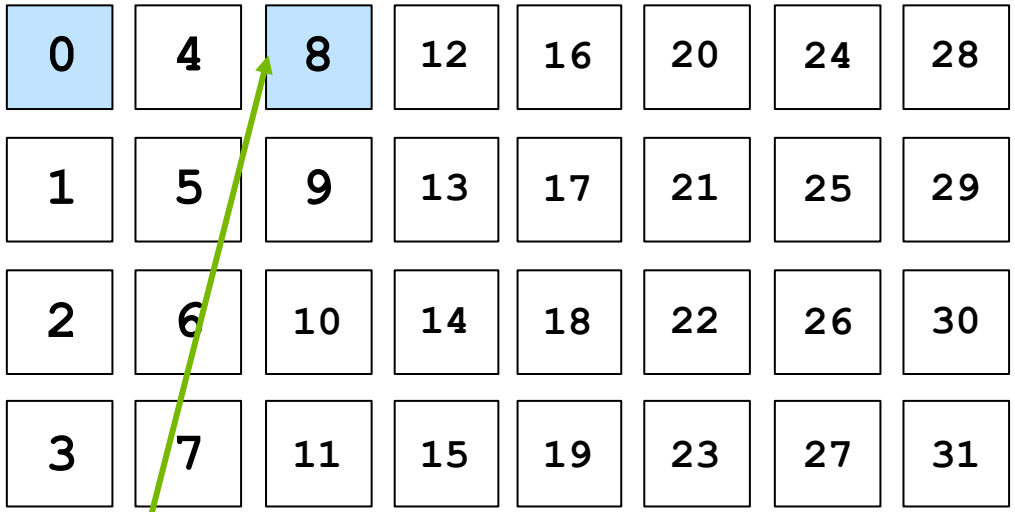


The thread then strides forward by the total number of threads in the grid (`blockDim.x * gridDim.x`), in this case `8`

GPU

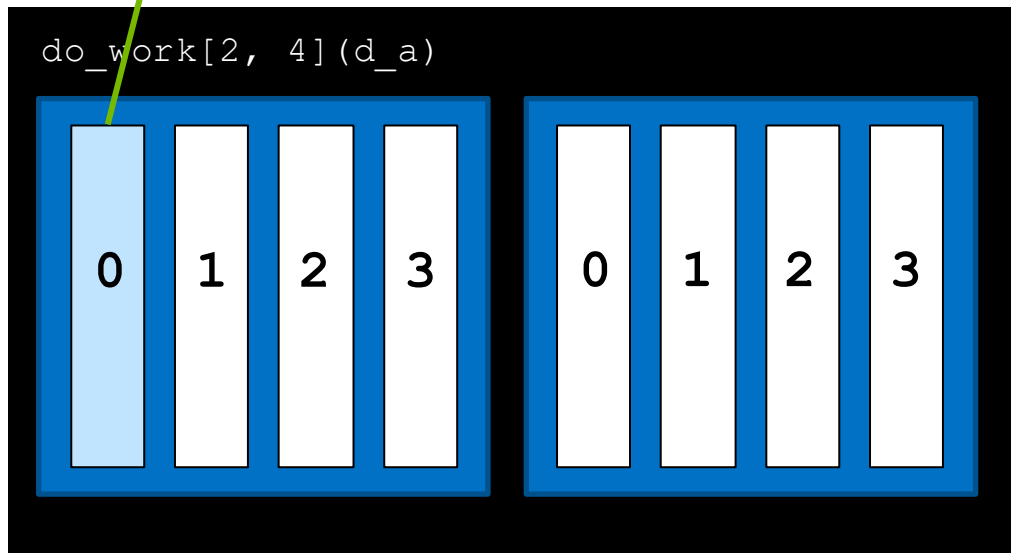


GPU
DATA



Numba provides another convenience function for this common calculation: **`cuda.gridsize()`**, returning the number of threads in the grid

GPU

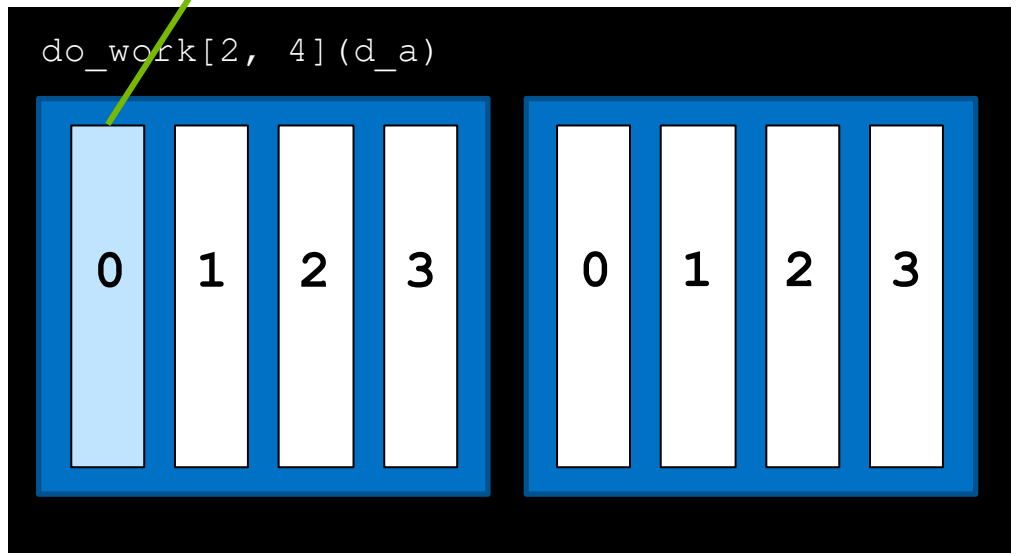


GPU
DATA

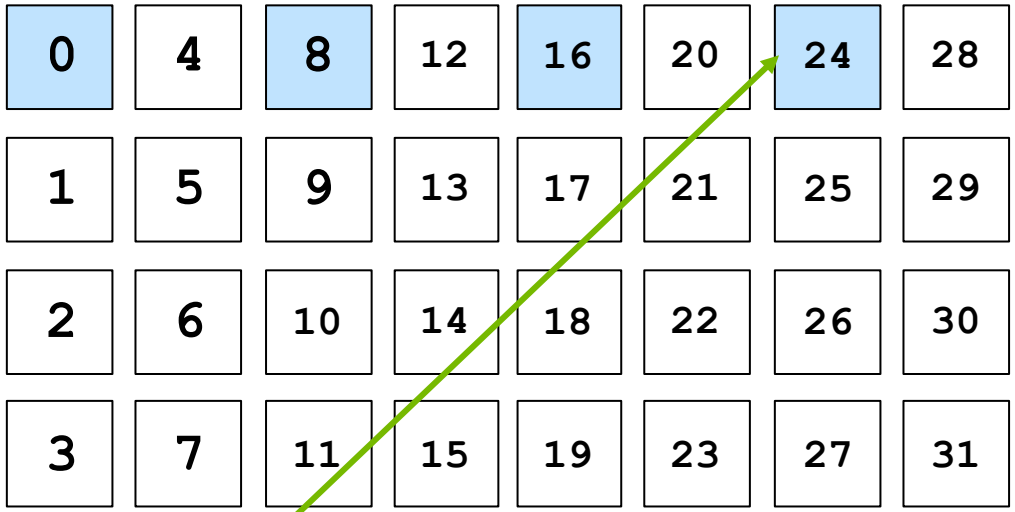
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

The thread continues in this way until its data index is greater than the number of data elements

GPU

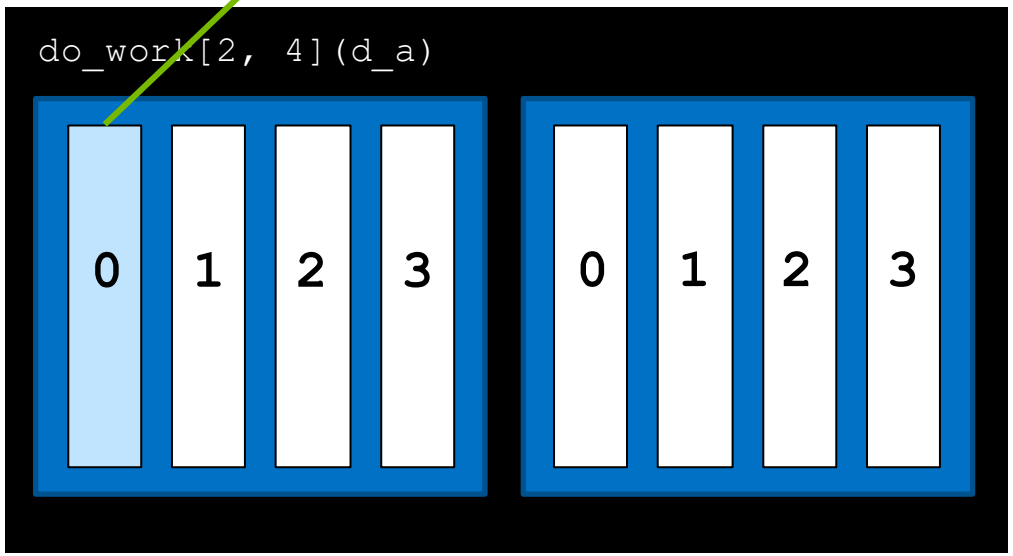


GPU
DATA



The thread continues in this way until its data index is greater than the number of data elements

GPU

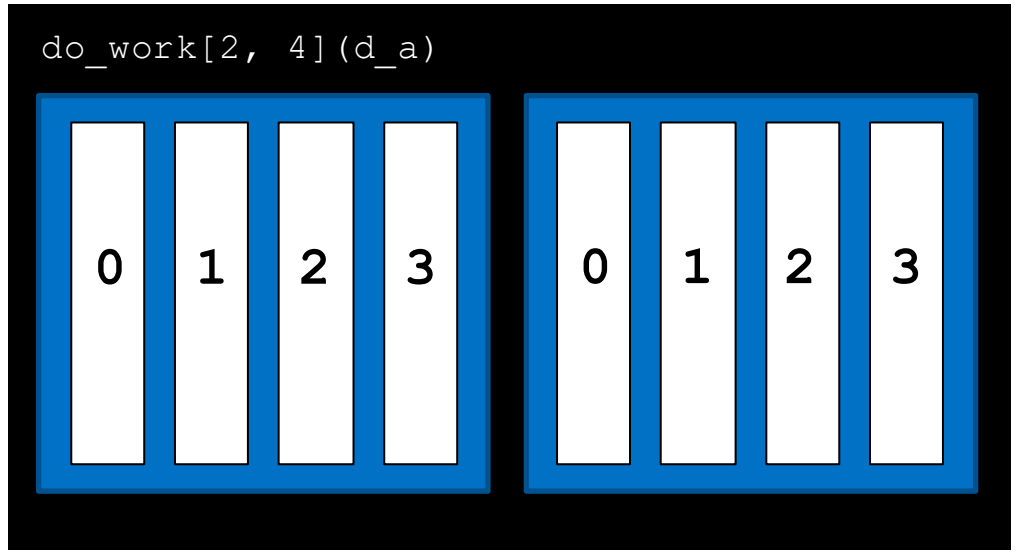


GPU
DATA

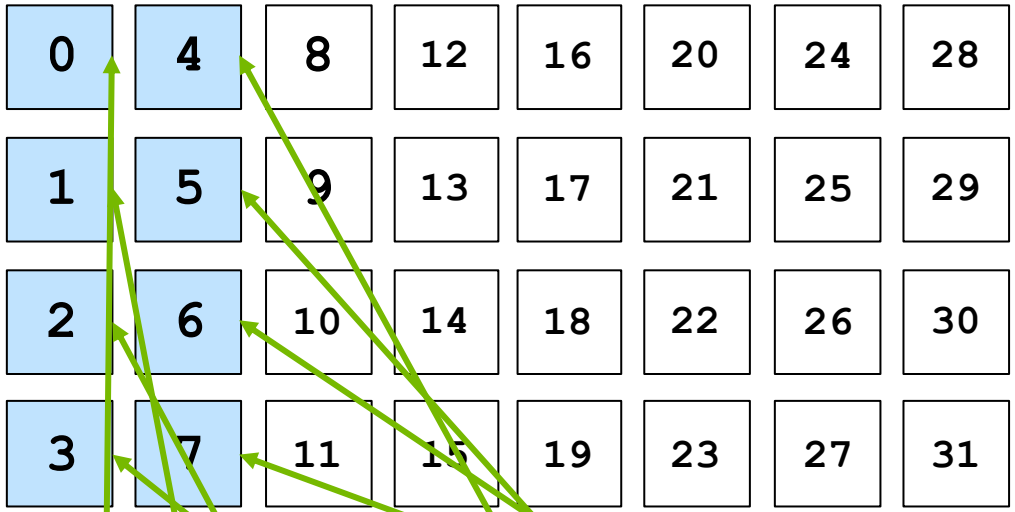
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in parallel using a grid stride loop...

GPU

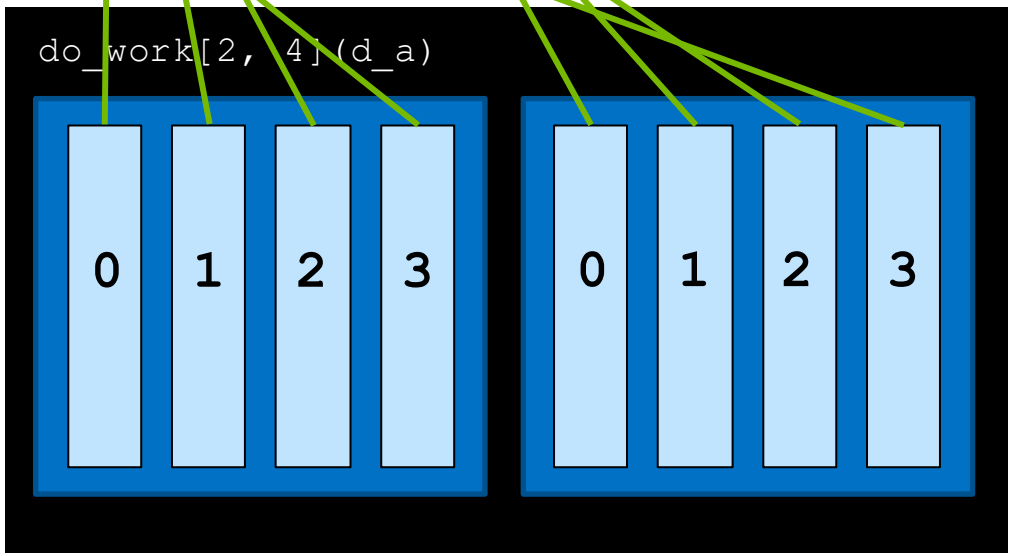


GPU
DATA

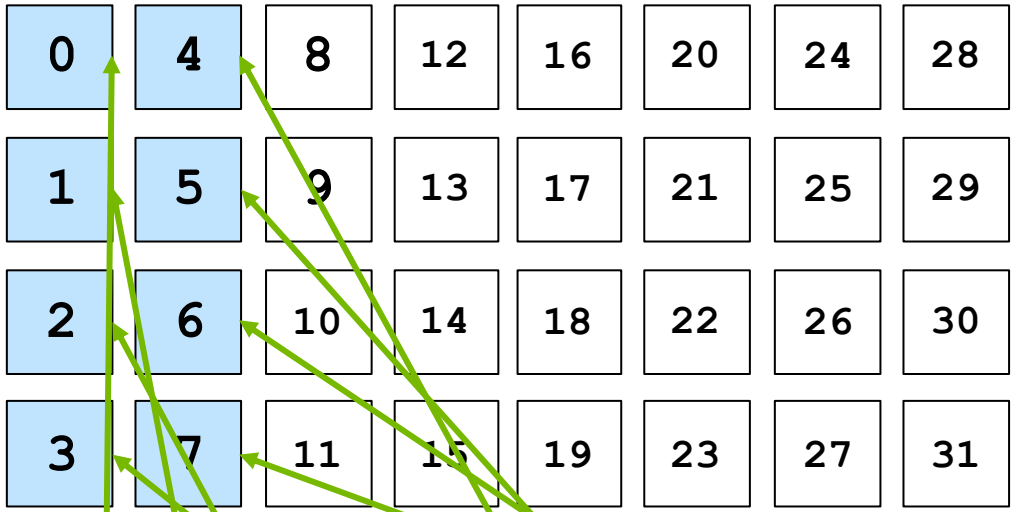


… all elements are covered

GPU

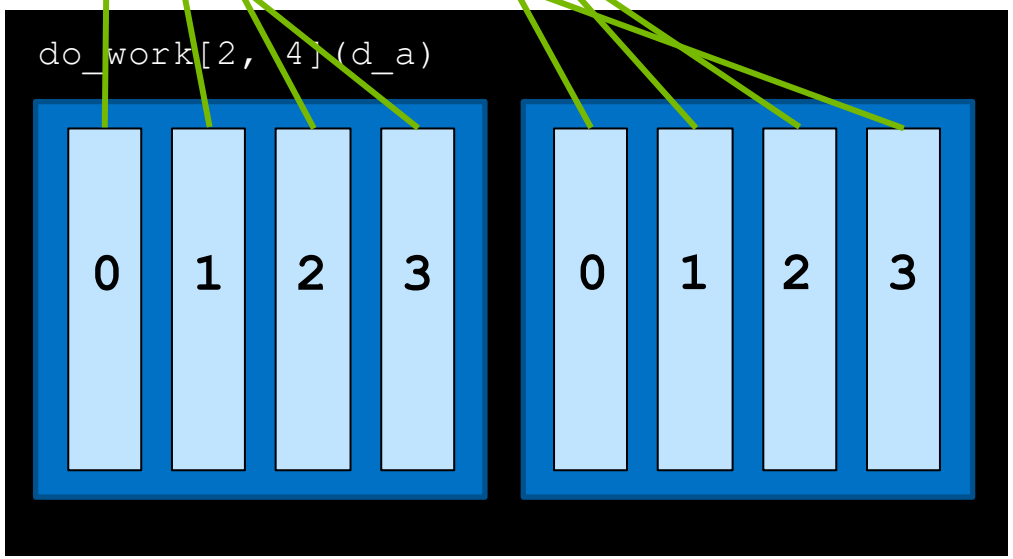


GPU
DATA

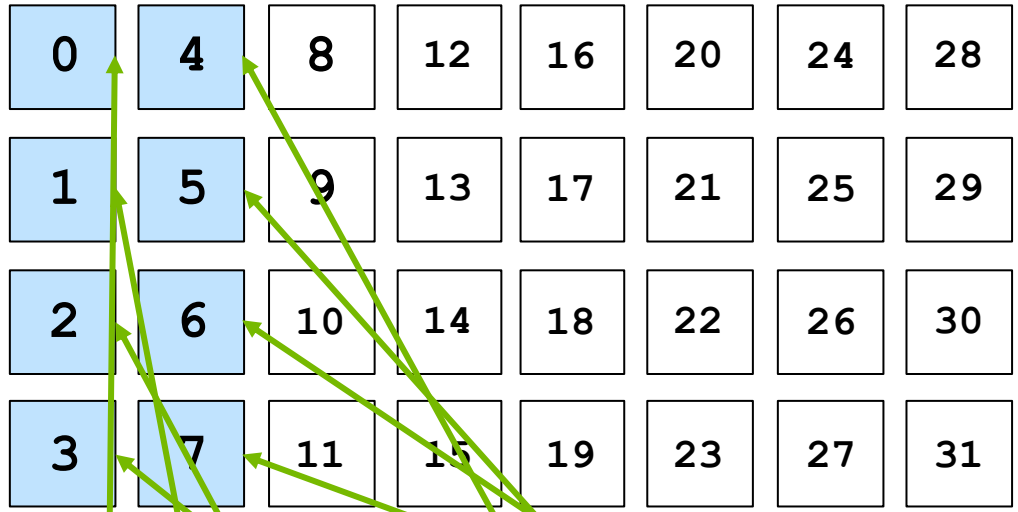


Additionally the device **coalesces** memory reads/writes into as few transactions as possible for performance...

GPU

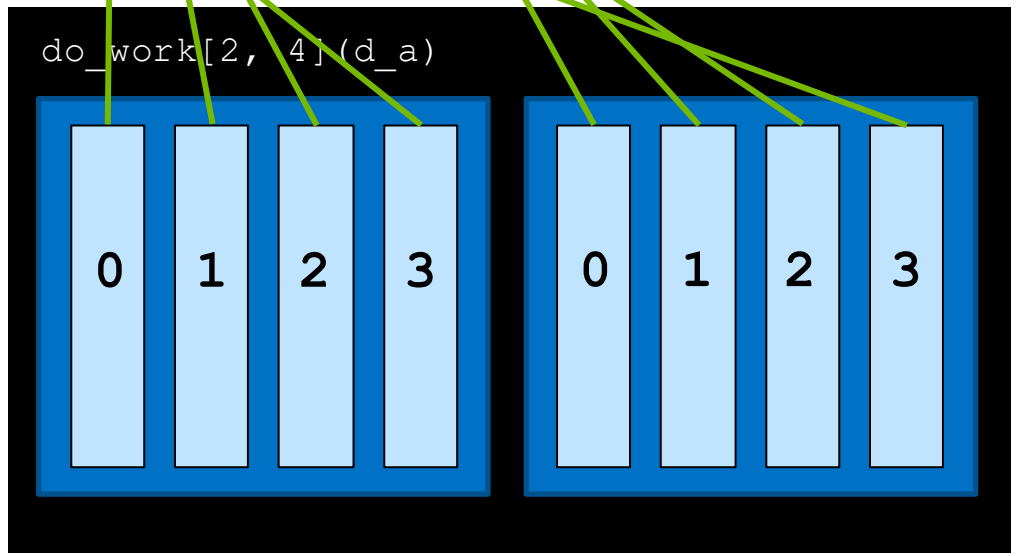


GPU
DATA



And grid stride loops support this **memory coalescing** because threads executing in parallel will access adjacent data elements

GPU



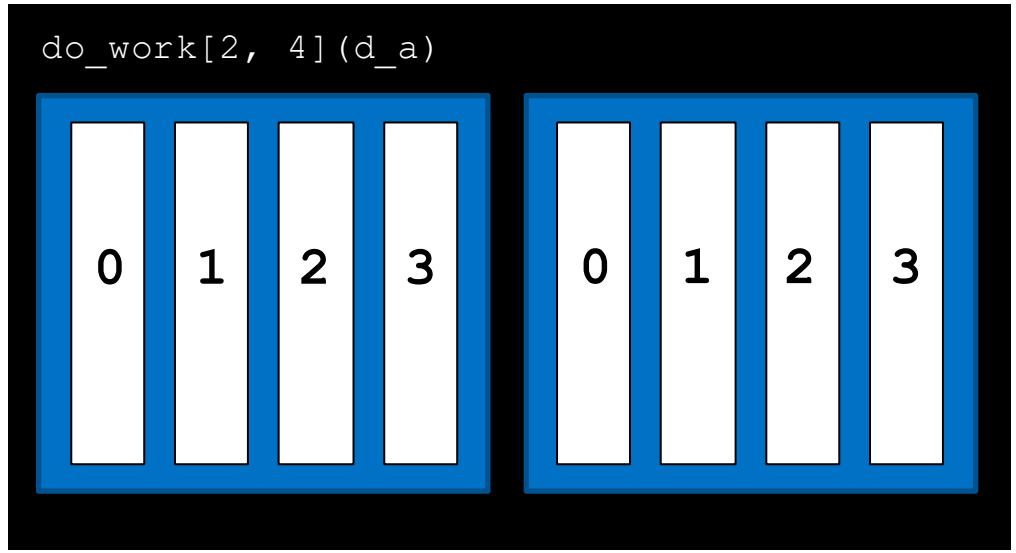
GPU
DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

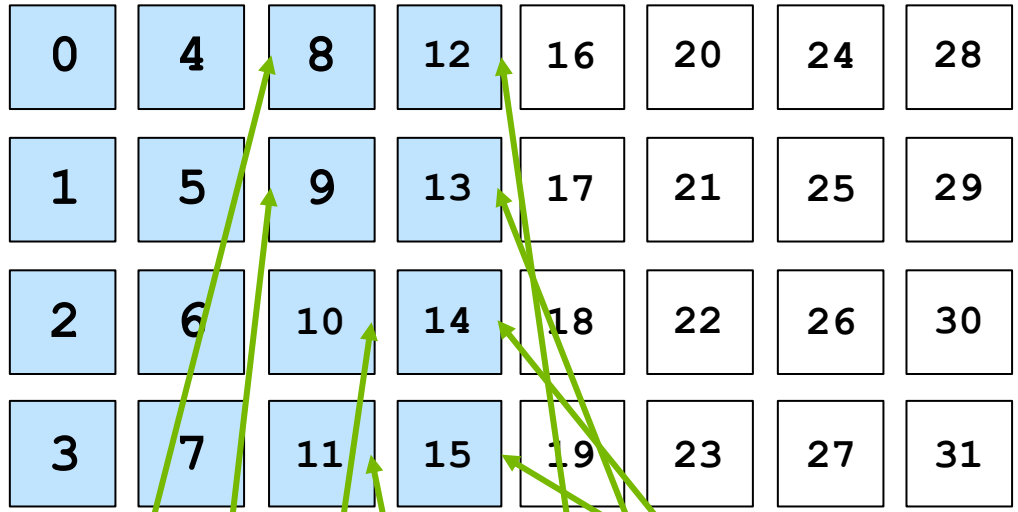
With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

---

GPU

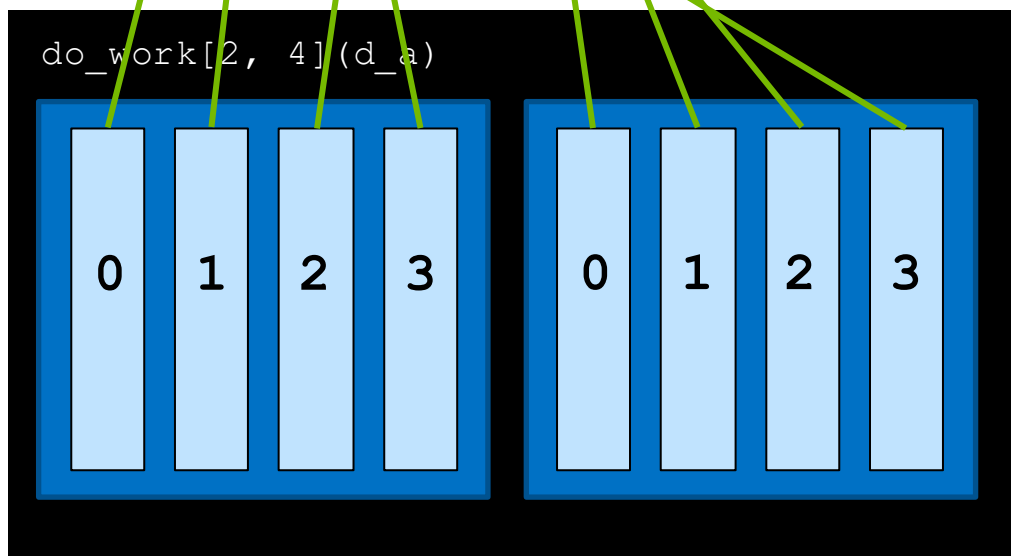


GPU
DATA



With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

GPU



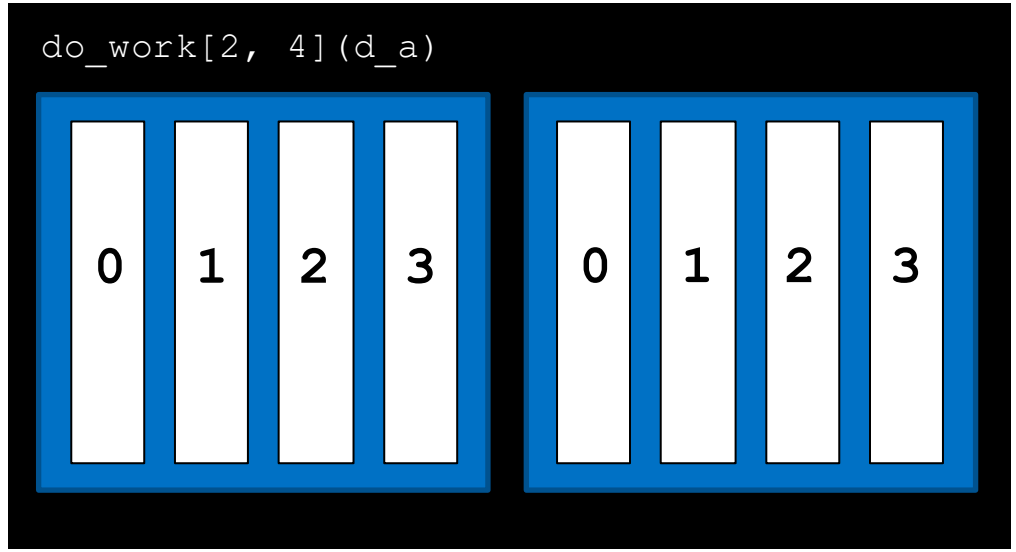
GPU
DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

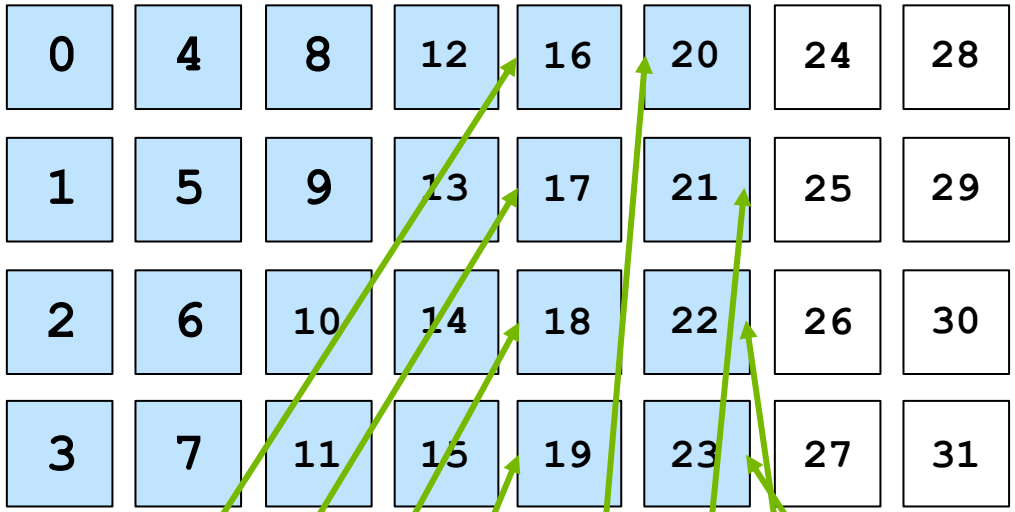
With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

---

GPU

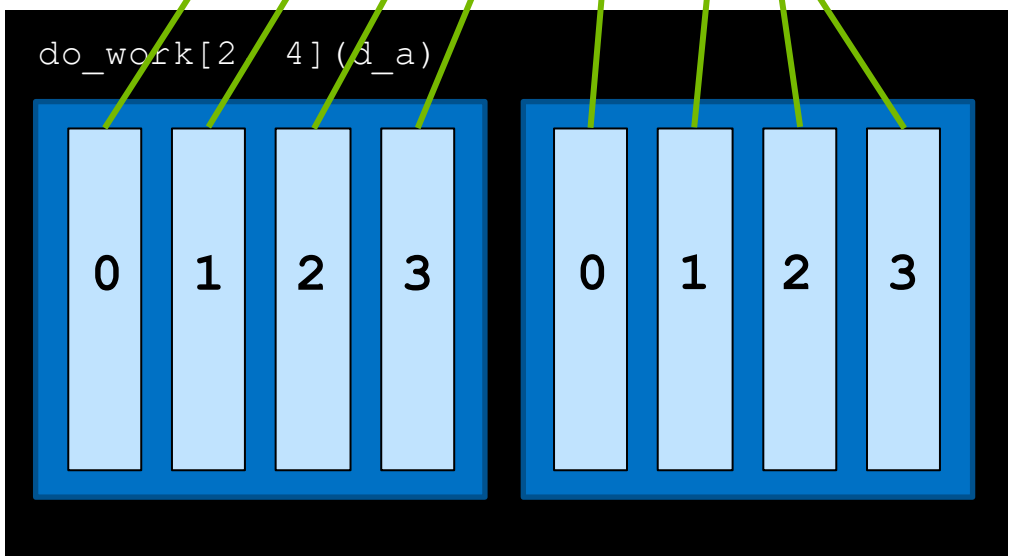


GPU
DATA



With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

GPU



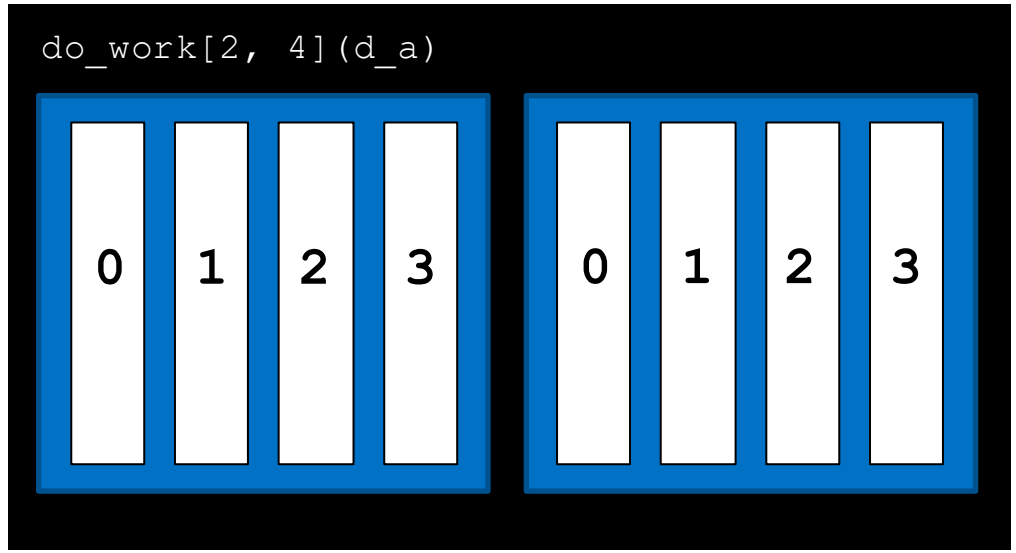
GPU
DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

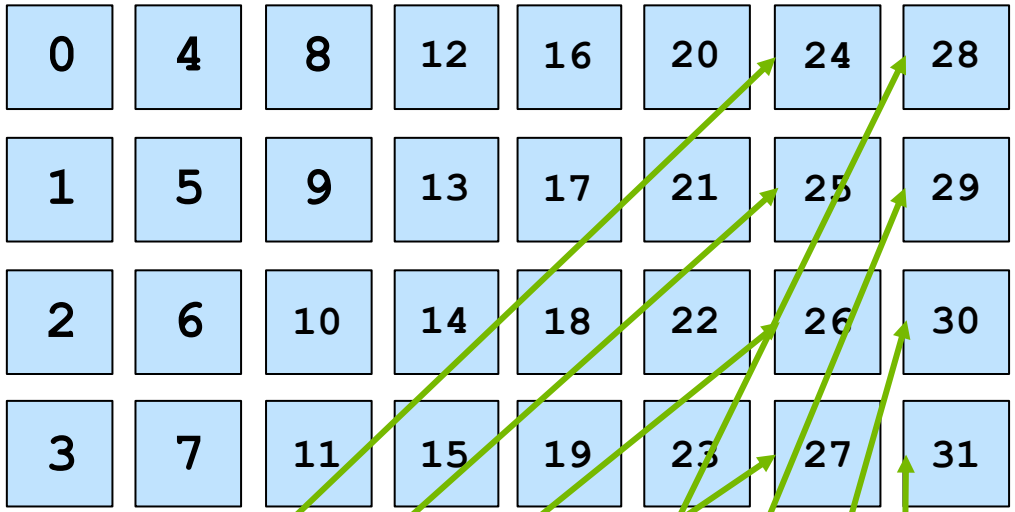
With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

---

GPU

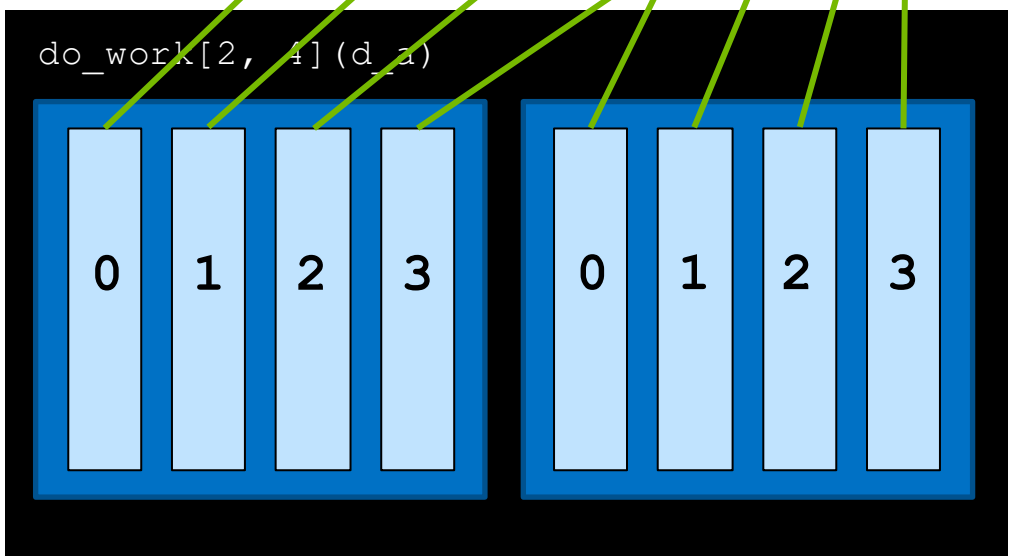


GPU
DATA



With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

GPU



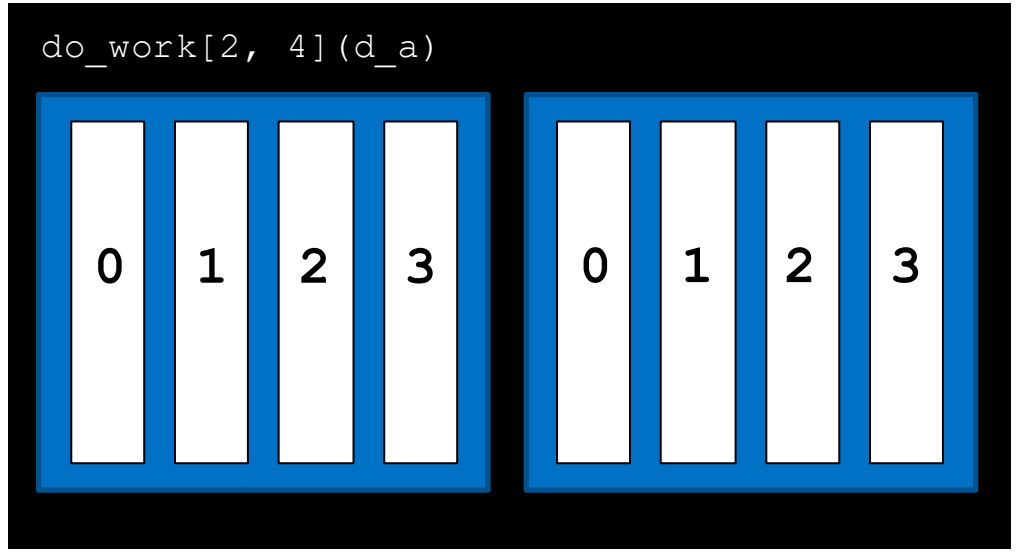
GPU
DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered with the performance advantage of memory coalescing

---

GPU



www.nvidia.com/dli