

# Template numerical library for modern parallel architectures

**Tomáš Oberhuber**   **Jakub Klinkovský**   **Radek Fučík**  
**Vítězslav Žabka**   **Aleš Wodecki**



Department of Mathematics,  
Faculty of Nuclear Sciences and Physical Engineering,  
Czech Technical University in Prague



IMG 2019

# Why GPU?



	Nvidia V100	Intel Xeon E5-4660
Cores	5120 @ 1.3GHz	16 @ 3.0GHz
Peak perf.	15.7/7.8 TFlops	0.4 / 0.2 TFlops
Max. RAM	32 GB	1.5 TB
Memory bw.	900 GB/s	68 GB/s
TDP	300 W	120 W

≈ 8,000 \$

# Difficulties in programming GPUs?

Unfortunately,

- the programmer must have good knowledge of the hardware
- porting a code to GPUs often means rewriting the code from scratch
- lack of support in older numerical libraries

Numerical libraries which makes GPUs easily accessible are being developed.

# Template Numerical Library

**TNL** = Template Numerical Library

- is written in C++ and profits from meta-programming
- provides unified interface to multi-core CPUs and GPUs (via CUDA)
- wants to be user friendly
- [www.tnl-project.org](http://www.tnl-project.org)
- $\approx$  300k lines of templated code
- MIT license

# Arrays

Arrays are basic structures for memory management

- `TNL::Array< ElementType, DeviceType, IndexType, Allocator >`
- `DeviceType` says where the array resides
  - `TNL::Devices::Host` for CPU
  - `TNL::Devices::Cuda` for GPU
- `Allocator` performs the memory allocation
  - common CPU and GPU memory allocators
  - page-locked memory allocator
  - CUDA Unified Memory allocator
- I/O operations, elements manipulation ...

```
1 Array< float , Devices::Cuda , int > a( 100 );  
2 a.evaluate( [ ] __cuda_callable__ ( int i ) {  
3     return i%5; } );
```

# Vectors

Vectors add algebraic operations to arrays:

- `TNL::Vector< RealType, DeviceType, IndexType, Allocator >`
- addition, multiplication, scalar product,  $l_p$  norms ...

# Parallel reduction

Parallel reduction is an operation taking all array/vector elements as input and returns one value as output:

- array comparison
- scalar product
- $l_p$  norm
- minimal/maximal value
- sum of all elements

```
1 float sum( 0.0 )
2 for( int i = 0; i < size; i++ )
3     sum += a[ i ];
```

# Parallel reduction on GPU = 150 lines of code

```
1 // Parallel reduction on GPU
2 //
3 // This code is a simplified version of the parallel reduction algorithm
4 // used in the TNL library. It is designed to be easy to understand
5 // and modify.
6 //
7 // The code is organized into several sections:
8 // - Initialization: Setting up the GPU device and the input data.
9 // - Parallel reduction: The main algorithm, which is implemented
10 //   as a series of recursive calls to a function named 'reduce'.
11 // - Finalization: Printing the result and cleaning up the GPU device.
12 //
13 // The 'reduce' function is the core of the algorithm. It takes an array
14 // of numbers and returns the sum of all elements. The array is
15 // passed to the function as a pointer to its first element and the
16 // number of elements. The function uses a recursive approach,
17 // splitting the array into two halves and reducing each half
18 // separately. The results of the two reductions are then combined
19 // to produce the final sum.
20 //
21 // The code is written in C++ and uses the CUDA library for GPU
22 // acceleration. It is a good example of how to write parallel
23 // code on a GPU.
24 //
25 // The code is 150 lines long, which is a reasonable size for a
26 // simple algorithm like this.
27 //
28 // The code is as follows:
29 //
30 // #include <cuda_runtime.h>
31 // #include <device_launch_parameters.h>
32 // #include <stdio.h>
33 // #include <string.h>
34 // #include <math.h>
35 //
36 // #define N 1024
37 //
38 // __global__ void reduce_kernel(int *a, int n)
39 // {
40 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
41 //     if (i < n)
42 //         a[i] = a[i] + 1;
43 // }
44 //
45 // int main()
46 // {
47 //     cudaDeviceProp prop;
48 //     cudaGetDeviceProperties(&prop, 0);
49 //     printf("GPU Device: %s\n", prop.name);
50 //
51 //     int n = N;
52 //     int *a = (int *) malloc(n * sizeof(int));
53 //     for (int i = 0; i < n; i++)
54 //         a[i] = 0;
55 //
56 //     cudaStream_t stream;
57 //     cudaStreamCreate(&stream);
58 //
59 //     dim3 blockDim(1024);
60 //     dim3 gridDim(1);
61 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
62 //
63 //     for (int i = 0; i < n; i++)
64 //         a[i] = a[i] + 1;
65 //
66 //     printf("Sum: %d\n", a[n-1]);
67 //
68 //     free(a);
69 //     cudaStreamDestroy(stream);
70 //
71 //     return 0;
72 // }
73 //
74 // The code is 150 lines long, which is a reasonable size for a
75 // simple algorithm like this.
76 //
77 // The code is as follows:
78 //
79 // #include <cuda_runtime.h>
80 // #include <device_launch_parameters.h>
81 // #include <stdio.h>
82 // #include <string.h>
83 // #include <math.h>
84 //
85 // #define N 1024
86 //
87 // __global__ void reduce_kernel(int *a, int n)
88 // {
89 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
90 //     if (i < n)
91 //         a[i] = a[i] + 1;
92 // }
93 //
94 // int main()
95 // {
96 //     cudaDeviceProp prop;
97 //     cudaGetDeviceProperties(&prop, 0);
98 //     printf("GPU Device: %s\n", prop.name);
99 //
100 //     int n = N;
101 //     int *a = (int *) malloc(n * sizeof(int));
102 //     for (int i = 0; i < n; i++)
103 //         a[i] = 0;
104 //
105 //     cudaStream_t stream;
106 //     cudaStreamCreate(&stream);
107 //
108 //     dim3 blockDim(1024);
109 //     dim3 gridDim(1);
110 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
111 //
112 //     for (int i = 0; i < n; i++)
113 //         a[i] = a[i] + 1;
114 //
115 //     printf("Sum: %d\n", a[n-1]);
116 //
117 //     free(a);
118 //     cudaStreamDestroy(stream);
119 //
120 //     return 0;
121 // }
122 //
123 // The code is 150 lines long, which is a reasonable size for a
124 // simple algorithm like this.
125 //
126 // The code is as follows:
127 //
128 // #include <cuda_runtime.h>
129 // #include <device_launch_parameters.h>
130 // #include <stdio.h>
131 // #include <string.h>
132 // #include <math.h>
133 //
134 // #define N 1024
135 //
136 // __global__ void reduce_kernel(int *a, int n)
137 // {
138 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
139 //     if (i < n)
140 //         a[i] = a[i] + 1;
141 // }
142 //
143 // int main()
144 // {
145 //     cudaDeviceProp prop;
146 //     cudaGetDeviceProperties(&prop, 0);
147 //     printf("GPU Device: %s\n", prop.name);
148 //
149 //     int n = N;
150 //     int *a = (int *) malloc(n * sizeof(int));
151 //     for (int i = 0; i < n; i++)
152 //         a[i] = 0;
153 //
154 //     cudaStream_t stream;
155 //     cudaStreamCreate(&stream);
156 //
157 //     dim3 blockDim(1024);
158 //     dim3 gridDim(1);
159 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
160 //
161 //     for (int i = 0; i < n; i++)
162 //         a[i] = a[i] + 1;
163 //
164 //     printf("Sum: %d\n", a[n-1]);
165 //
166 //     free(a);
167 //     cudaStreamDestroy(stream);
168 //
169 //     return 0;
170 // }
171 //
172 // The code is 150 lines long, which is a reasonable size for a
173 // simple algorithm like this.
174 //
175 // The code is as follows:
176 //
177 // #include <cuda_runtime.h>
178 // #include <device_launch_parameters.h>
179 // #include <stdio.h>
180 // #include <string.h>
181 // #include <math.h>
182 //
183 // #define N 1024
184 //
185 // __global__ void reduce_kernel(int *a, int n)
186 // {
187 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
188 //     if (i < n)
189 //         a[i] = a[i] + 1;
190 // }
191 //
192 // int main()
193 // {
194 //     cudaDeviceProp prop;
195 //     cudaGetDeviceProperties(&prop, 0);
196 //     printf("GPU Device: %s\n", prop.name);
197 //
198 //     int n = N;
199 //     int *a = (int *) malloc(n * sizeof(int));
200 //     for (int i = 0; i < n; i++)
201 //         a[i] = 0;
202 //
203 //     cudaStream_t stream;
204 //     cudaStreamCreate(&stream);
205 //
206 //     dim3 blockDim(1024);
207 //     dim3 gridDim(1);
208 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
209 //
210 //     for (int i = 0; i < n; i++)
211 //         a[i] = a[i] + 1;
212 //
213 //     printf("Sum: %d\n", a[n-1]);
214 //
215 //     free(a);
216 //     cudaStreamDestroy(stream);
217 //
218 //     return 0;
219 // }
220 //
221 // The code is 150 lines long, which is a reasonable size for a
222 // simple algorithm like this.
223 //
224 // The code is as follows:
225 //
226 // #include <cuda_runtime.h>
227 // #include <device_launch_parameters.h>
228 // #include <stdio.h>
229 // #include <string.h>
230 // #include <math.h>
231 //
232 // #define N 1024
233 //
234 // __global__ void reduce_kernel(int *a, int n)
235 // {
236 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
237 //     if (i < n)
238 //         a[i] = a[i] + 1;
239 // }
240 //
241 // int main()
242 // {
243 //     cudaDeviceProp prop;
244 //     cudaGetDeviceProperties(&prop, 0);
245 //     printf("GPU Device: %s\n", prop.name);
246 //
247 //     int n = N;
248 //     int *a = (int *) malloc(n * sizeof(int));
249 //     for (int i = 0; i < n; i++)
250 //         a[i] = 0;
251 //
252 //     cudaStream_t stream;
253 //     cudaStreamCreate(&stream);
254 //
255 //     dim3 blockDim(1024);
256 //     dim3 gridDim(1);
257 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
258 //
259 //     for (int i = 0; i < n; i++)
260 //         a[i] = a[i] + 1;
261 //
262 //     printf("Sum: %d\n", a[n-1]);
263 //
264 //     free(a);
265 //     cudaStreamDestroy(stream);
266 //
267 //     return 0;
268 // }
269 //
270 // The code is 150 lines long, which is a reasonable size for a
271 // simple algorithm like this.
272 //
273 // The code is as follows:
274 //
275 // #include <cuda_runtime.h>
276 // #include <device_launch_parameters.h>
277 // #include <stdio.h>
278 // #include <string.h>
279 // #include <math.h>
280 //
281 // #define N 1024
282 //
283 // __global__ void reduce_kernel(int *a, int n)
284 // {
285 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
286 //     if (i < n)
287 //         a[i] = a[i] + 1;
288 // }
289 //
290 // int main()
291 // {
292 //     cudaDeviceProp prop;
293 //     cudaGetDeviceProperties(&prop, 0);
294 //     printf("GPU Device: %s\n", prop.name);
295 //
296 //     int n = N;
297 //     int *a = (int *) malloc(n * sizeof(int));
298 //     for (int i = 0; i < n; i++)
299 //         a[i] = 0;
300 //
301 //     cudaStream_t stream;
302 //     cudaStreamCreate(&stream);
303 //
304 //     dim3 blockDim(1024);
305 //     dim3 gridDim(1);
306 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
307 //
308 //     for (int i = 0; i < n; i++)
309 //         a[i] = a[i] + 1;
310 //
311 //     printf("Sum: %d\n", a[n-1]);
312 //
313 //     free(a);
314 //     cudaStreamDestroy(stream);
315 //
316 //     return 0;
317 // }
318 //
319 // The code is 150 lines long, which is a reasonable size for a
320 // simple algorithm like this.
321 //
322 // The code is as follows:
323 //
324 // #include <cuda_runtime.h>
325 // #include <device_launch_parameters.h>
326 // #include <stdio.h>
327 // #include <string.h>
328 // #include <math.h>
329 //
330 // #define N 1024
331 //
332 // __global__ void reduce_kernel(int *a, int n)
333 // {
334 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
335 //     if (i < n)
336 //         a[i] = a[i] + 1;
337 // }
338 //
339 // int main()
340 // {
341 //     cudaDeviceProp prop;
342 //     cudaGetDeviceProperties(&prop, 0);
343 //     printf("GPU Device: %s\n", prop.name);
344 //
345 //     int n = N;
346 //     int *a = (int *) malloc(n * sizeof(int));
347 //     for (int i = 0; i < n; i++)
348 //         a[i] = 0;
349 //
350 //     cudaStream_t stream;
351 //     cudaStreamCreate(&stream);
352 //
353 //     dim3 blockDim(1024);
354 //     dim3 gridDim(1);
355 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
356 //
357 //     for (int i = 0; i < n; i++)
358 //         a[i] = a[i] + 1;
359 //
360 //     printf("Sum: %d\n", a[n-1]);
361 //
362 //     free(a);
363 //     cudaStreamDestroy(stream);
364 //
365 //     return 0;
366 // }
367 //
368 // The code is 150 lines long, which is a reasonable size for a
369 // simple algorithm like this.
370 //
371 // The code is as follows:
372 //
373 // #include <cuda_runtime.h>
374 // #include <device_launch_parameters.h>
375 // #include <stdio.h>
376 // #include <string.h>
377 // #include <math.h>
378 //
379 // #define N 1024
380 //
381 // __global__ void reduce_kernel(int *a, int n)
382 // {
383 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
384 //     if (i < n)
385 //         a[i] = a[i] + 1;
386 // }
387 //
388 // int main()
389 // {
390 //     cudaDeviceProp prop;
391 //     cudaGetDeviceProperties(&prop, 0);
392 //     printf("GPU Device: %s\n", prop.name);
393 //
394 //     int n = N;
395 //     int *a = (int *) malloc(n * sizeof(int));
396 //     for (int i = 0; i < n; i++)
397 //         a[i] = 0;
398 //
399 //     cudaStream_t stream;
400 //     cudaStreamCreate(&stream);
401 //
402 //     dim3 blockDim(1024);
403 //     dim3 gridDim(1);
404 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
405 //
406 //     for (int i = 0; i < n; i++)
407 //         a[i] = a[i] + 1;
408 //
409 //     printf("Sum: %d\n", a[n-1]);
410 //
411 //     free(a);
412 //     cudaStreamDestroy(stream);
413 //
414 //     return 0;
415 // }
416 //
417 // The code is 150 lines long, which is a reasonable size for a
418 // simple algorithm like this.
419 //
420 // The code is as follows:
421 //
422 // #include <cuda_runtime.h>
423 // #include <device_launch_parameters.h>
424 // #include <stdio.h>
425 // #include <string.h>
426 // #include <math.h>
427 //
428 // #define N 1024
429 //
430 // __global__ void reduce_kernel(int *a, int n)
431 // {
432 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
433 //     if (i < n)
434 //         a[i] = a[i] + 1;
435 // }
436 //
437 // int main()
438 // {
439 //     cudaDeviceProp prop;
440 //     cudaGetDeviceProperties(&prop, 0);
441 //     printf("GPU Device: %s\n", prop.name);
442 //
443 //     int n = N;
444 //     int *a = (int *) malloc(n * sizeof(int));
445 //     for (int i = 0; i < n; i++)
446 //         a[i] = 0;
447 //
448 //     cudaStream_t stream;
449 //     cudaStreamCreate(&stream);
450 //
451 //     dim3 blockDim(1024);
452 //     dim3 gridDim(1);
453 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
454 //
455 //     for (int i = 0; i < n; i++)
456 //         a[i] = a[i] + 1;
457 //
458 //     printf("Sum: %d\n", a[n-1]);
459 //
460 //     free(a);
461 //     cudaStreamDestroy(stream);
462 //
463 //     return 0;
464 // }
465 //
466 // The code is 150 lines long, which is a reasonable size for a
467 // simple algorithm like this.
468 //
469 // The code is as follows:
470 //
471 // #include <cuda_runtime.h>
472 // #include <device_launch_parameters.h>
473 // #include <stdio.h>
474 // #include <string.h>
475 // #include <math.h>
476 //
477 // #define N 1024
478 //
479 // __global__ void reduce_kernel(int *a, int n)
480 // {
481 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
482 //     if (i < n)
483 //         a[i] = a[i] + 1;
484 // }
485 //
486 // int main()
487 // {
488 //     cudaDeviceProp prop;
489 //     cudaGetDeviceProperties(&prop, 0);
490 //     printf("GPU Device: %s\n", prop.name);
491 //
492 //     int n = N;
493 //     int *a = (int *) malloc(n * sizeof(int));
494 //     for (int i = 0; i < n; i++)
495 //         a[i] = 0;
496 //
497 //     cudaStream_t stream;
498 //     cudaStreamCreate(&stream);
499 //
500 //     dim3 blockDim(1024);
501 //     dim3 gridDim(1);
502 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
503 //
504 //     for (int i = 0; i < n; i++)
505 //         a[i] = a[i] + 1;
506 //
507 //     printf("Sum: %d\n", a[n-1]);
508 //
509 //     free(a);
510 //     cudaStreamDestroy(stream);
511 //
512 //     return 0;
513 // }
514 //
515 // The code is 150 lines long, which is a reasonable size for a
516 // simple algorithm like this.
517 //
518 // The code is as follows:
519 //
520 // #include <cuda_runtime.h>
521 // #include <device_launch_parameters.h>
522 // #include <stdio.h>
523 // #include <string.h>
524 // #include <math.h>
525 //
526 // #define N 1024
527 //
528 // __global__ void reduce_kernel(int *a, int n)
529 // {
530 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
531 //     if (i < n)
532 //         a[i] = a[i] + 1;
533 // }
534 //
535 // int main()
536 // {
537 //     cudaDeviceProp prop;
538 //     cudaGetDeviceProperties(&prop, 0);
539 //     printf("GPU Device: %s\n", prop.name);
540 //
541 //     int n = N;
542 //     int *a = (int *) malloc(n * sizeof(int));
543 //     for (int i = 0; i < n; i++)
544 //         a[i] = 0;
545 //
546 //     cudaStream_t stream;
547 //     cudaStreamCreate(&stream);
548 //
549 //     dim3 blockDim(1024);
550 //     dim3 gridDim(1);
551 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
552 //
553 //     for (int i = 0; i < n; i++)
554 //         a[i] = a[i] + 1;
555 //
556 //     printf("Sum: %d\n", a[n-1]);
557 //
558 //     free(a);
559 //     cudaStreamDestroy(stream);
560 //
561 //     return 0;
562 // }
563 //
564 // The code is 150 lines long, which is a reasonable size for a
565 // simple algorithm like this.
566 //
567 // The code is as follows:
568 //
569 // #include <cuda_runtime.h>
570 // #include <device_launch_parameters.h>
571 // #include <stdio.h>
572 // #include <string.h>
573 // #include <math.h>
574 //
575 // #define N 1024
576 //
577 // __global__ void reduce_kernel(int *a, int n)
578 // {
579 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
580 //     if (i < n)
581 //         a[i] = a[i] + 1;
582 // }
583 //
584 // int main()
585 // {
586 //     cudaDeviceProp prop;
587 //     cudaGetDeviceProperties(&prop, 0);
588 //     printf("GPU Device: %s\n", prop.name);
589 //
590 //     int n = N;
591 //     int *a = (int *) malloc(n * sizeof(int));
592 //     for (int i = 0; i < n; i++)
593 //         a[i] = 0;
594 //
595 //     cudaStream_t stream;
596 //     cudaStreamCreate(&stream);
597 //
598 //     dim3 blockDim(1024);
599 //     dim3 gridDim(1);
600 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
601 //
602 //     for (int i = 0; i < n; i++)
603 //         a[i] = a[i] + 1;
604 //
605 //     printf("Sum: %d\n", a[n-1]);
606 //
607 //     free(a);
608 //     cudaStreamDestroy(stream);
609 //
610 //     return 0;
611 // }
612 //
613 // The code is 150 lines long, which is a reasonable size for a
614 // simple algorithm like this.
615 //
616 // The code is as follows:
617 //
618 // #include <cuda_runtime.h>
619 // #include <device_launch_parameters.h>
620 // #include <stdio.h>
621 // #include <string.h>
622 // #include <math.h>
623 //
624 // #define N 1024
625 //
626 // __global__ void reduce_kernel(int *a, int n)
627 // {
628 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
629 //     if (i < n)
630 //         a[i] = a[i] + 1;
631 // }
632 //
633 // int main()
634 // {
635 //     cudaDeviceProp prop;
636 //     cudaGetDeviceProperties(&prop, 0);
637 //     printf("GPU Device: %s\n", prop.name);
638 //
639 //     int n = N;
640 //     int *a = (int *) malloc(n * sizeof(int));
641 //     for (int i = 0; i < n; i++)
642 //         a[i] = 0;
643 //
644 //     cudaStream_t stream;
645 //     cudaStreamCreate(&stream);
646 //
647 //     dim3 blockDim(1024);
648 //     dim3 gridDim(1);
649 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
650 //
651 //     for (int i = 0; i < n; i++)
652 //         a[i] = a[i] + 1;
653 //
654 //     printf("Sum: %d\n", a[n-1]);
655 //
656 //     free(a);
657 //     cudaStreamDestroy(stream);
658 //
659 //     return 0;
660 // }
661 //
662 // The code is 150 lines long, which is a reasonable size for a
663 // simple algorithm like this.
664 //
665 // The code is as follows:
666 //
667 // #include <cuda_runtime.h>
668 // #include <device_launch_parameters.h>
669 // #include <stdio.h>
670 // #include <string.h>
671 // #include <math.h>
672 //
673 // #define N 1024
674 //
675 // __global__ void reduce_kernel(int *a, int n)
676 // {
677 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
678 //     if (i < n)
679 //         a[i] = a[i] + 1;
680 // }
681 //
682 // int main()
683 // {
684 //     cudaDeviceProp prop;
685 //     cudaGetDeviceProperties(&prop, 0);
686 //     printf("GPU Device: %s\n", prop.name);
687 //
688 //     int n = N;
689 //     int *a = (int *) malloc(n * sizeof(int));
690 //     for (int i = 0; i < n; i++)
691 //         a[i] = 0;
692 //
693 //     cudaStream_t stream;
694 //     cudaStreamCreate(&stream);
695 //
696 //     dim3 blockDim(1024);
697 //     dim3 gridDim(1);
698 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
699 //
700 //     for (int i = 0; i < n; i++)
701 //         a[i] = a[i] + 1;
702 //
703 //     printf("Sum: %d\n", a[n-1]);
704 //
705 //     free(a);
706 //     cudaStreamDestroy(stream);
707 //
708 //     return 0;
709 // }
710 //
711 // The code is 150 lines long, which is a reasonable size for a
712 // simple algorithm like this.
713 //
714 // The code is as follows:
715 //
716 // #include <cuda_runtime.h>
717 // #include <device_launch_parameters.h>
718 // #include <stdio.h>
719 // #include <string.h>
720 // #include <math.h>
721 //
722 // #define N 1024
723 //
724 // __global__ void reduce_kernel(int *a, int n)
725 // {
726 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
727 //     if (i < n)
728 //         a[i] = a[i] + 1;
729 // }
730 //
731 // int main()
732 // {
733 //     cudaDeviceProp prop;
734 //     cudaGetDeviceProperties(&prop, 0);
735 //     printf("GPU Device: %s\n", prop.name);
736 //
737 //     int n = N;
738 //     int *a = (int *) malloc(n * sizeof(int));
739 //     for (int i = 0; i < n; i++)
740 //         a[i] = 0;
741 //
742 //     cudaStream_t stream;
743 //     cudaStreamCreate(&stream);
744 //
745 //     dim3 blockDim(1024);
746 //     dim3 gridDim(1);
747 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
748 //
749 //     for (int i = 0; i < n; i++)
750 //         a[i] = a[i] + 1;
751 //
752 //     printf("Sum: %d\n", a[n-1]);
753 //
754 //     free(a);
755 //     cudaStreamDestroy(stream);
756 //
757 //     return 0;
758 // }
759 //
760 // The code is 150 lines long, which is a reasonable size for a
761 // simple algorithm like this.
762 //
763 // The code is as follows:
764 //
765 // #include <cuda_runtime.h>
766 // #include <device_launch_parameters.h>
767 // #include <stdio.h>
768 // #include <string.h>
769 // #include <math.h>
770 //
771 // #define N 1024
772 //
773 // __global__ void reduce_kernel(int *a, int n)
774 // {
775 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
776 //     if (i < n)
777 //         a[i] = a[i] + 1;
778 // }
779 //
780 // int main()
781 // {
782 //     cudaDeviceProp prop;
783 //     cudaGetDeviceProperties(&prop, 0);
784 //     printf("GPU Device: %s\n", prop.name);
785 //
786 //     int n = N;
787 //     int *a = (int *) malloc(n * sizeof(int));
788 //     for (int i = 0; i < n; i++)
789 //         a[i] = 0;
790 //
791 //     cudaStream_t stream;
792 //     cudaStreamCreate(&stream);
793 //
794 //     dim3 blockDim(1024);
795 //     dim3 gridDim(1);
796 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
797 //
798 //     for (int i = 0; i < n; i++)
799 //         a[i] = a[i] + 1;
800 //
801 //     printf("Sum: %d\n", a[n-1]);
802 //
803 //     free(a);
804 //     cudaStreamDestroy(stream);
805 //
806 //     return 0;
807 // }
808 //
809 // The code is 150 lines long, which is a reasonable size for a
810 // simple algorithm like this.
811 //
812 // The code is as follows:
813 //
814 // #include <cuda_runtime.h>
815 // #include <device_launch_parameters.h>
816 // #include <stdio.h>
817 // #include <string.h>
818 // #include <math.h>
819 //
820 // #define N 1024
821 //
822 // __global__ void reduce_kernel(int *a, int n)
823 // {
824 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
825 //     if (i < n)
826 //         a[i] = a[i] + 1;
827 // }
828 //
829 // int main()
830 // {
831 //     cudaDeviceProp prop;
832 //     cudaGetDeviceProperties(&prop, 0);
833 //     printf("GPU Device: %s\n", prop.name);
834 //
835 //     int n = N;
836 //     int *a = (int *) malloc(n * sizeof(int));
837 //     for (int i = 0; i < n; i++)
838 //         a[i] = 0;
839 //
840 //     cudaStream_t stream;
841 //     cudaStreamCreate(&stream);
842 //
843 //     dim3 blockDim(1024);
844 //     dim3 gridDim(1);
845 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
846 //
847 //     for (int i = 0; i < n; i++)
848 //         a[i] = a[i] + 1;
849 //
850 //     printf("Sum: %d\n", a[n-1]);
851 //
852 //     free(a);
853 //     cudaStreamDestroy(stream);
854 //
855 //     return 0;
856 // }
857 //
858 // The code is 150 lines long, which is a reasonable size for a
859 // simple algorithm like this.
860 //
861 // The code is as follows:
862 //
863 // #include <cuda_runtime.h>
864 // #include <device_launch_parameters.h>
865 // #include <stdio.h>
866 // #include <string.h>
867 // #include <math.h>
868 //
869 // #define N 1024
870 //
871 // __global__ void reduce_kernel(int *a, int n)
872 // {
873 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
874 //     if (i < n)
875 //         a[i] = a[i] + 1;
876 // }
877 //
878 // int main()
879 // {
880 //     cudaDeviceProp prop;
881 //     cudaGetDeviceProperties(&prop, 0);
882 //     printf("GPU Device: %s\n", prop.name);
883 //
884 //     int n = N;
885 //     int *a = (int *) malloc(n * sizeof(int));
886 //     for (int i = 0; i < n; i++)
887 //         a[i] = 0;
888 //
889 //     cudaStream_t stream;
890 //     cudaStreamCreate(&stream);
891 //
892 //     dim3 blockDim(1024);
893 //     dim3 gridDim(1);
894 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
895 //
896 //     for (int i = 0; i < n; i++)
897 //         a[i] = a[i] + 1;
898 //
899 //     printf("Sum: %d\n", a[n-1]);
900 //
901 //     free(a);
902 //     cudaStreamDestroy(stream);
903 //
904 //     return 0;
905 // }
906 //
907 // The code is 150 lines long, which is a reasonable size for a
908 // simple algorithm like this.
909 //
910 // The code is as follows:
911 //
912 // #include <cuda_runtime.h>
913 // #include <device_launch_parameters.h>
914 // #include <stdio.h>
915 // #include <string.h>
916 // #include <math.h>
917 //
918 // #define N 1024
919 //
920 // __global__ void reduce_kernel(int *a, int n)
921 // {
922 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
923 //     if (i < n)
924 //         a[i] = a[i] + 1;
925 // }
926 //
927 // int main()
928 // {
929 //     cudaDeviceProp prop;
930 //     cudaGetDeviceProperties(&prop, 0);
931 //     printf("GPU Device: %s\n", prop.name);
932 //
933 //     int n = N;
934 //     int *a = (int *) malloc(n * sizeof(int));
935 //     for (int i = 0; i < n; i++)
936 //         a[i] = 0;
937 //
938 //     cudaStream_t stream;
939 //     cudaStreamCreate(&stream);
940 //
941 //     dim3 blockDim(1024);
942 //     dim3 gridDim(1);
943 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
944 //
945 //     for (int i = 0; i < n; i++)
946 //         a[i] = a[i] + 1;
947 //
948 //     printf("Sum: %d\n", a[n-1]);
949 //
950 //     free(a);
951 //     cudaStreamDestroy(stream);
952 //
953 //     return 0;
954 // }
955 //
956 // The code is 150 lines long, which is a reasonable size for a
957 // simple algorithm like this.
958 //
959 // The code is as follows:
960 //
961 // #include <cuda_runtime.h>
962 // #include <device_launch_parameters.h>
963 // #include <stdio.h>
964 // #include <string.h>
965 // #include <math.h>
966 //
967 // #define N 1024
968 //
969 // __global__ void reduce_kernel(int *a, int n)
970 // {
971 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
972 //     if (i < n)
973 //         a[i] = a[i] + 1;
974 // }
975 //
976 // int main()
977 // {
978 //     cudaDeviceProp prop;
979 //     cudaGetDeviceProperties(&prop, 0);
980 //     printf("GPU Device: %s\n", prop.name);
981 //
982 //     int n = N;
983 //     int *a = (int *) malloc(n * sizeof(int));
984 //     for (int i = 0; i < n; i++)
985 //         a[i] = 0;
986 //
987 //     cudaStream_t stream;
988 //     cudaStreamCreate(&stream);
989 //
990 //     dim3 blockDim(1024);
991 //     dim3 gridDim(1);
992 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
993 //
994 //     for (int i = 0; i < n; i++)
995 //         a[i] = a[i] + 1;
996 //
997 //     printf("Sum: %d\n", a[n-1]);
998 //
999 //     free(a);
1000 //     cudaStreamDestroy(stream);
1001 //
1002 //     return 0;
1003 // }
1004 //
1005 // The code is 150 lines long, which is a reasonable size for a
1006 // simple algorithm like this.
1007 //
1008 // The code is as follows:
1009 //
1010 // #include <cuda_runtime.h>
1011 // #include <device_launch_parameters.h>
1012 // #include <stdio.h>
1013 // #include <string.h>
1014 // #include <math.h>
1015 //
1016 // #define N 1024
1017 //
1018 // __global__ void reduce_kernel(int *a, int n)
1019 // {
1020 //     int i = blockIdx.x * blockDim.x + threadIdx.x;
1021 //     if (i < n)
1022 //         a[i] = a[i] + 1;
1023 // }
1024 //
1025 // int main()
1026 // {
1027 //     cudaDeviceProp prop;
1028 //     cudaGetDeviceProperties(&prop, 0);
1029 //     printf("GPU Device: %s\n", prop.name);
1030 //
1031 //     int n = N;
1032 //     int *a = (int *) malloc(n * sizeof(int));
1033 //     for (int i = 0; i < n; i++)
1034 //         a[i] = 0;
1035 //
1036 //     cudaStream_t stream;
1037 //     cudaStreamCreate(&stream);
1038 //
1039 //     dim3 blockDim(1024);
1040 //     dim3 gridDim(1);
1041 //     cudaLaunchKernel(reduce_kernel, gridDim, blockDim, a, n, stream);
1042 //
1043 //     for (int i = 0; i < n; i++)
1044 //         a[i] = a[i] + 1;
1045 //
1046 //     printf("Sum: %d\n", a[n-1]);
1047 //
1048 //     free(a);
1049 //
```



# Flexible parallel reduction in TNL

Take a look at scalar product:

```

1 float result( 0.0 );
2 for( int i = 0; i < size; i++ )
3     result += a[ i ] * b[ i ];

```

Let us rewrite it using C++ lambda functions as:

```

1 float a[ size ], b[size ];
2
3 ...
4
5 auto fetch = [=] (int i)->float { return a[i]*b[i];};
6 auto reduce = [] (float& x, const float& y) { x += y;};
7
8 float result( 0.0 );
9 for( int i = 0; i < size; i++ )
10     reduce( result, fetch( i ) );

```

# Flexible parallel reduction in TNL

To perform the same on GPU in TNL just add  
`__cuda_callable__` to lambdas...

```

1 auto fetch = [=] __cuda_callable__ (int i)->bool {
2     return ( a[i] * b[i] ); };
3 auto reduce = [] __cuda_callable__ (float& x, const float& y) {
4     x += y; };

```

... and for certain reasons, deliver volatile version of reduce:

```

1 auto volatileReduce = [] __cuda_callable__ (volatile float& x,
2     volatile const float& y) {
3     x += y; };

```

This could be avoided when CUDA compiler supports C++17  
 better. Now call

```

1 Reduction< Devices::Cuda >::
2     reduce( size , reduce , volatileReduce , fetch , zero );

```

# Expression Templates in TNL

Algebraic vector expressions are handled by **expression templates**:

```
1 x = a + 2 * b + 3 * c;
```

- simple
- works for both CPU and GPU
- efficient
  - **one** loop on CPU
  - specialized **one** CUDA kernel for each expression on GPU

# Expression Templates & Parallel Reduction in TNL

Example:

```
1 using Vector = Vector< float , Devices::Cuda, int >;
2 Vector a( 100 ), b( 100 ), c( 100 ), d( 100 );
3 ...
4 float scalarProduct = ( a, b + 3 * c );
5 d = a + b * c + sin( d );
6 a = min( b, c );
7 float min_a = min( a );
8 float total_min = min( min( a, b ) );
```

# Performance comparison

Performance was tested on:

- GPU Nvidia P100
  - 16 GB HBM2 @ 732 GB/s
  - 3584 CUDA cores, 4.7 TFlops in double precision
- CPU
  - Intel Core i7-5820K, 3.3GHz, 16MB cache

# Expression Templates in TNL

Scalar product:  $r = (x, y)$ .

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	17.4	5.0	<b>0.3</b>	49.3	69.9	<b>1.41</b>
200k	17.4	5.0	<b>0.3</b>	90.1	108.6	<b>1.20</b>
400k	17.7	5.0	<b>0.3</b>	142.2	159.1	<b>1.11</b>
800k	13.7	4.8	<b>0.3</b>	207.4	233.4	<b>1.12</b>
1.6M	12.6	4.8	<b>0.4</b>	313.6	333.3	<b>1.06</b>
3.2M	12.8	4.6	<b>0.4</b>	381.0	403.7	<b>1.05</b>
6.4M	12.7	4.6	<b>0.4</b>	417.1	431.8	<b>1.03</b>

BW = effective memory bandwidth in GB/s

# Expression Templates in TNL

Vector addition:  $x += a$ .

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	34.0	42.0	<b>1.2</b>	152.2	174.8	<b>1.14</b>
200k	35.1	43.0	<b>1.2</b>	196.6	216.1	<b>1.09</b>
400k	31.7	36.7	<b>1.2</b>	277.6	294.4	<b>1.06</b>
800k	19.7	19.3	<b>0.97</b>	326.2	333.6	<b>1.02</b>
1.6M	18.1	17.3	<b>0.95</b>	362.5	374.2	<b>1.03</b>
3.2M	18.4	17.6	<b>0.95</b>	422.4	436.8	<b>1.03</b>
6.4M	18.3	17.5	<b>0.95</b>	456.6	469.8	<b>1.02</b>

BW = effective memory bandwidth in GB/s

# Expression Templates in TNL

Vector addition:  $x += a + b$ .

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	23.6	42.0	<b>1.8</b>	188.3	190.9	<b>1.01</b>
200k	23.5	41.8	<b>1.8</b>	218.0	230.7	<b>1.05</b>
400k	20.9	37.4	<b>1.8</b>	243.1	305.9	<b>1.25</b>
800k	13.7	18.7	<b>1.4</b>	263.8	353.0	<b>1.33</b>
1.6M	12.2	16.8	<b>1.4</b>	285.9	389.4	<b>1.36</b>
3.2M	12.3	17.5	<b>1.4</b>	312.9	442.8	<b>1.41</b>
6.4M	12.2	17.3	<b>1.4</b>	327.3	471.9	<b>1.44</b>

BW = effective memory bandwidth in GB/s



# Expression Templates in TNL

Vector addition:  $x += a + b + c$ .

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	19.3	41.5	<b>2.2</b>	194.7	236.5	<b>1.21</b>
200k	19.7	41.7	<b>2.1</b>	228.3	277.6	<b>1.21</b>
400k	17.3	35.9	<b>2.1</b>	218.3	330.9	<b>1.51</b>
800k	11.7	19.3	<b>1.6</b>	233.3	370.6	<b>1.58</b>
1.6M	10.4	17.0	<b>1.6</b>	249.6	403.4	<b>1.61</b>
3.2M	10.2	17.3	<b>1.7</b>	266.6	444.8	<b>1.66</b>
6.4M	10.2	17.3	<b>1.7</b>	276.6	471.3	<b>1.70</b>

BW = effective memory bandwidth in GB/s

# Matrix formats

TNL supports the following matrix formats (on both CPU and GPU):

- dense matrix format
- tridiagonal and multidiagonal matrix format
- Ellpack format
- CSR format
- SlicedEllpack format
- ChunkedEllpack format

Oberhuber T., Suzuki A., Vacata J., *New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA*, Acta Technica, 2011, vol. 56, no. 4, pp. 447-466.

Heller M., Oberhuber T., *Improved Row-grouped CSR Format for Storing of Sparse Matrices on GPU*, Proceedings of Algoritmy 2012, 2012, Handlovičová A., Minarechová Z. and Ševčovič D. (ed.), pages 282-290.

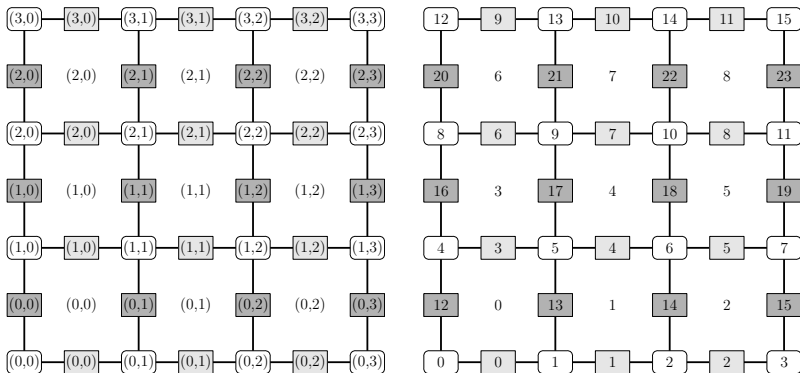
# Numerical meshes

TNL supports

- structured orthogonal **grids** – 1D, 2D, 3D
  - mesh entities are generated on the fly
- unstructured **meshes** – nD
  - mesh entities are stored in memory

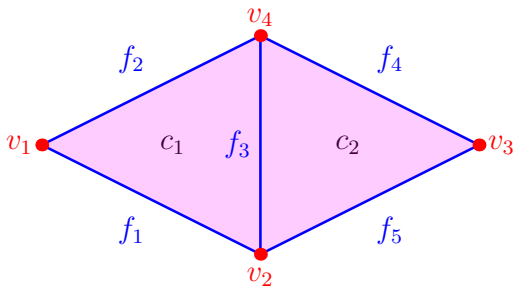
# Structured grids

TNL::Meshes::Grid< Dimensions,Real,Device,Index >



Grid provides mapping between coordinates and global indexes.

## Unstructured meshes



$$I_{0,1} = \left( \begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline v_1 & 1 & 1 & & & \\ v_2 & 1 & & 1 & & 1 \\ v_3 & & & & 1 & 1 \\ v_4 & & 1 & 1 & 1 & \end{array} \right) \quad I_{0,2} = \left( \begin{array}{c|cc} & c_1 & c_2 \\ \hline v_1 & 1 & \\ v_2 & 1 & 1 \\ v_3 & & 1 \\ v_4 & 1 & 1 \end{array} \right)$$

# Unstructured meshes

```
TNL::Meshes::Mesh< MeshConfig, Device >
```

- can have arbitrary dimension
- MeshConfig controls what mesh entities and links between them are stored
- it is done in the compile-time thanks to C++ templates

**Based on MeshConfig, the mesh is fine-tuned for specific numerical method in compile-time.**

# Solvers

## ODEs solvers

- Euler, Runge-Kutta-Merson

## Linear systems solvers

- Krylov subspace methods (CG, BiCGSTab, GMRES, TFQMR)
- highly parallel CWYGMRES method

Klinkovský J., Oberhuber T., Fučík R., *Performance evaluation of distributed MGSR- and CWY- based GMRES variants of MHFEM*, submitted to International Journal of High Performance Computing Applications.

Oberhuber T., Suzuki A., Žabka V., *The CUDA implementation of the method of lines for the curvature dependent flows*, Kybernetika, 2011, vol. 47, num. 2, pp. 251–272.

Oberhuber T., Suzuki A., Vacata J., Žabka V., *Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods*, Journal of Math-for-Industry, 2011, vol. 3, pp. 73–79.

# Multiphase flow in porous media

We consider the following system of  $n$  partial differential equations in a general coefficient form

$$\sum_{j=1}^n N_{i,j} \frac{\partial Z_j}{\partial t} + \sum_{j=1}^n \mathbf{u}_{i,j} \cdot \nabla Z_j + \nabla \cdot \left[ m_i \left( - \sum_{j=1}^n D_{i,j} \nabla Z_j + \mathbf{w}_i \right) + \sum_{j=1}^n Z_j \mathbf{a}_{i,j} \right] + \sum_{j=1}^n r_{i,j} Z_j = f_i$$

for  $i = 1, \dots, n$ , where the **unknown vector function**  $\vec{Z} = (Z_1, \dots, Z_n)^T$  depends on position vector  $\vec{x} \in \Omega \subset \mathbb{R}^d$  and time  $t \in [0, T]$ ,  $d = 1, 2, 3$ .



# Multiphase flow in porous media

Initial condition:

$$Z_j(\vec{x}, 0) = Z_j^{ini}(\vec{x}), \quad \forall \vec{x} \in \Omega, \quad j = 1, \dots, n,$$

Boundary conditions:

$$\begin{aligned} Z_j(\vec{x}, t) &= Z_j^{\mathcal{D}}(\vec{x}, t), \quad \forall \vec{x} \in \Gamma_j^{\mathcal{D}} \subset \partial\Omega, \quad j = 1, \dots, n, \\ \vec{v}_i(\vec{x}, t) \cdot \vec{n}_{\partial\Omega}(\vec{x}) &= v_i^{\mathcal{N}}(\vec{x}, t), \quad \forall \vec{x} \in \Gamma_i^{\mathcal{N}} \subset \partial\Omega, \quad i = 1, \dots, n, \end{aligned}$$

where  $\vec{v}_i$  denotes the conservative velocity term

$$\vec{v}_i = - \sum_{j=1}^n \mathbf{D}_{i,j} \nabla Z_j + \mathbf{w}_i.$$

## Numerical method

- Based on the mixed-hybrid finite element method (MHFEM)
  - one global large sparse linear system for traces of  $(Z_1, \dots, Z_n)$  (on faces) per time step
- Semi-implicit time discretization
- General spatial dimension (1D, 2D, 3D)
- Structured and unstructured meshes

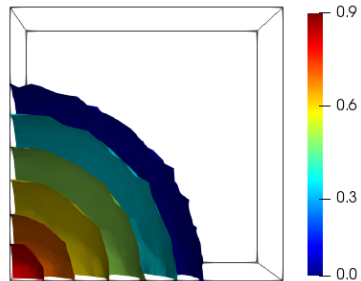
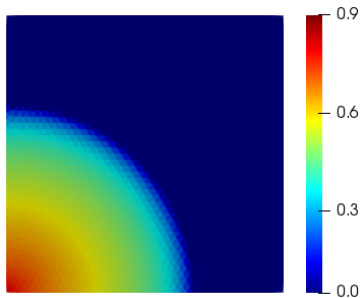
R. Fučík, J. Klinkovský, T. Oberhuber, J. Mikyška, *Multidimensional Mixed–Hybrid Finite Element Method for Compositional Two–Phase Flow in Heterogeneous Porous Media and its Parallel Implementation on GPU*, Computer Physics Communications 238, pp. 165–180, 2019.

# McWhorter–Sunada problem

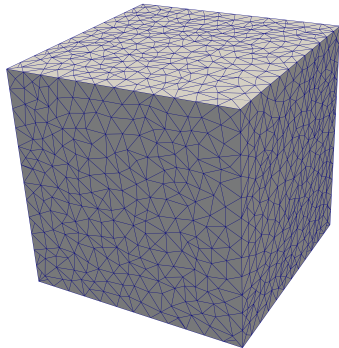
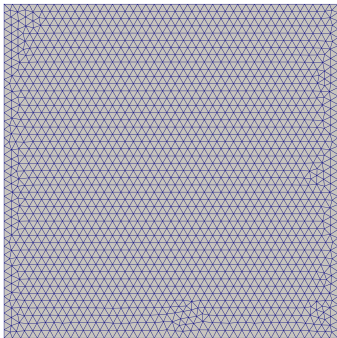
Benchmark problem – generalization of the McWhorter–Sunada problem

- Two phase flow in porous media
- General dimension (1D, 2D, 3D)
- Radial symmetry
- Point injection in the origin
- Incompressible phases and neglected gravity
- Semi-analytical solution by McWhorter and Sunada (1990) and Fučík *et al.* (2016)

# McWhorter–Sunada problem



# McWhorter–Sunada problem



# McWhorter–Sunada problem

Numerical simulations were performed on:

- 6-core CPU Intel i7-5820K at 3.3 GHz with 15 MB cache
- GPU Tesla K40 with 2880 CUDA cores at 0.745 GHz

# McWhorter–Sunada problem 2D

DOFs	GPU			CPU								
	<i>CT</i>	1 core		2 cores			4 cores			6 cores		
		<i>CT</i>	<i>GSp</i>	<i>CT</i>	<i>Eff</i>	<i>GSp</i>	<i>CT</i>	<i>Eff</i>	<i>GSp</i>	<i>CT</i>	<i>Eff</i>	<i>GSp</i>
Orthogonal grids												
960	1,5	0,7	<b>0,45</b>	0,4	0,79	<b>0,28</b>	0,3	0,52	<b>0,22</b>	0,3	0,41	<b>0,18</b>
3720	11,0	13,2	<b>1,20</b>	7,6	0,87	<b>0,69</b>	4,8	0,68	<b>0,44</b>	4,0	0,55	<b>0,37</b>
14640	46,3	197,0	<b>4,25</b>	107,5	0,92	<b>2,32</b>	65,7	0,75	<b>1,42</b>	52,6	0,62	<b>1,14</b>
58080	380,0	4325,7	<b>11,38</b>	2360,6	0,92	<b>6,21</b>	1448,1	0,75	<b>3,81</b>	1195,8	0,60	<b>3,15</b>
231360	4449,9	91166,3	<b>20,49</b>	49004,3	0,93	<b>11,01</b>	29182,1	0,78	<b>6,56</b>	24684,0	0,62	<b>5,55</b>
Unstructured meshes												
766	1,5	0,4	<b>0,27</b>	0,3	0,60	<b>0,22</b>	0,2	0,45	<b>0,15</b>	0,2	0,32	<b>0,14</b>
2912	8,9	6,2	<b>0,70</b>	3,7	0,84	<b>0,42</b>	2,3	0,66	<b>0,26</b>	2,0	0,52	<b>0,23</b>
11302	51,1	122,0	<b>2,39</b>	67,7	0,90	<b>1,32</b>	40,3	0,76	<b>0,79</b>	32,5	0,63	<b>0,64</b>
44684	396,1	2695,6	<b>6,80</b>	1480,7	0,91	<b>3,74</b>	855,2	0,79	<b>2,16</b>	671,7	0,67	<b>1,70</b>
178648	4008,3	57404,2	<b>14,32</b>	32100,5	0,89	<b>8,01</b>	18814,1	0,76	<b>4,69</b>	16414,0	0,58	<b>4,09</b>

# McWhorter–Sunada problem 3D

DOFs	GPU		CPU										
	CT	1 core		2 cores			4 cores			6 cores			
		CT	GS <sub>p</sub>	CT	Eff	GS <sub>p</sub>	CT	Eff	GS <sub>p</sub>	CT	Eff	GS <sub>p</sub>	
Orthogonal grids													
21 600	2,1	15,2	<b>7,30</b>	8,0	0,96	<b>3,82</b>	4,4	0,86	<b>2,13</b>	3,4	0,75	<b>1,62</b>	
167 400	30,8	564,3	<b>18,33</b>	319,5	0,88	<b>10,38</b>	186,7	0,76	<b>6,07</b>	150,3	0,63	<b>4,88</b>	
1 317 600	828,0	20 569,5	<b>24,84</b>	12 406,1	0,83	<b>14,98</b>	7 092,6	0,73	<b>8,57</b>	5 533,7	0,62	<b>6,68</b>	
10 454 400	31 805,6	(not computed on 1, 2 and 4 cores)									234 066,0	7,36	
Unstructured meshes													
5 874	1,4	2,0	<b>1,48</b>	1,2	0,85	<b>0,88</b>	0,7	0,68	<b>0,54</b>	0,6	0,54	<b>0,46</b>	
15 546	2,6	8,7	<b>3,30</b>	4,9	0,89	<b>1,85</b>	2,9	0,75	<b>1,10</b>	2,3	0,64	<b>0,86</b>	
121 678	23,9	330,9	<b>13,87</b>	184,8	0,90	<b>7,75</b>	107,9	0,77	<b>4,53</b>	93,4	0,59	<b>3,92</b>	
973 750	566,2	12 069,5	<b>21,32</b>	6 506,3	0,93	<b>11,49</b>	3 771,0	0,80	<b>6,66</b>	3 306,2	0,61	<b>5,84</b>	
7 807 218	37 695,3	(not computed on CPU)											



# Conclusion

Currently we are working on:

- MPI
- nd-arrays ( $\Rightarrow$  nd-grids)
- adaptive grids
- documentation

## More about TNL ...

TNL is available at

`www.tnl-project.org`

under MIT license.