In [1]:
```
#%%capture
!pip freeze > requirements.txt
!pip freeze | grep -v -f requirements.txt - | grep -v '^#' | grep -v '^-e ' | xargs pip uninstall -y
```

Abstract

This parallel processing work pertains to performing Fast Fourier Transforms (FFTs) and a Finite-Difference Method (FDM) implementation of the 1D/2D Poisson equation for a pn-diode; on a multicore CPU and a CUDA-enabled GPU (hardware accelerator) using MathWork®'s '**MATLAB Parallel Computing Toolbox**'.

Table of Contents

Part I: FFT

Parallel vs serial execution

Algorithm 1 Adapted FFT pseudocode for standardised hardware execution and benchmarking using MATLAB

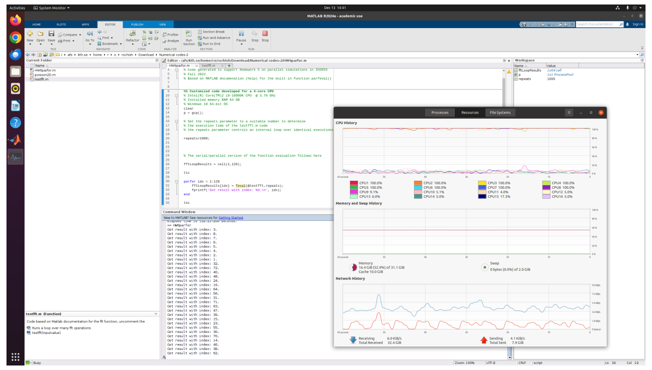
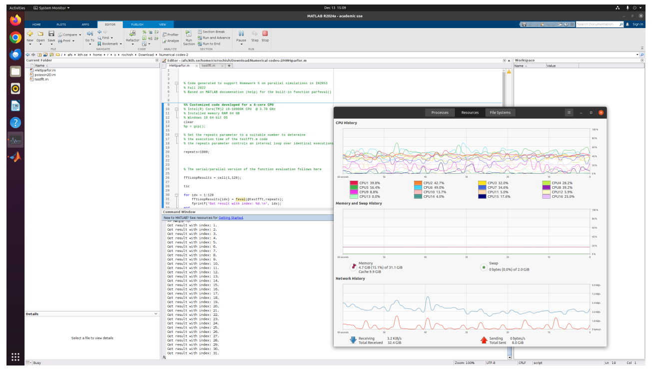
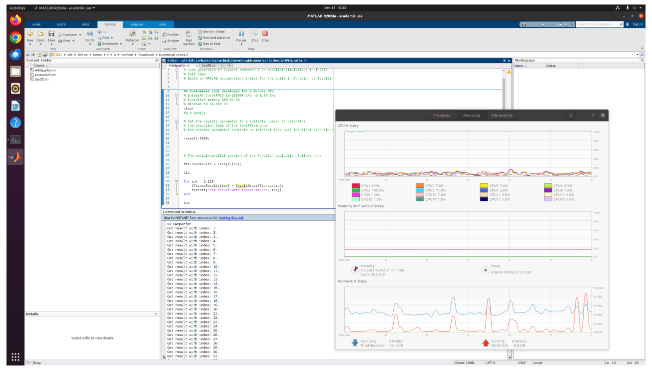
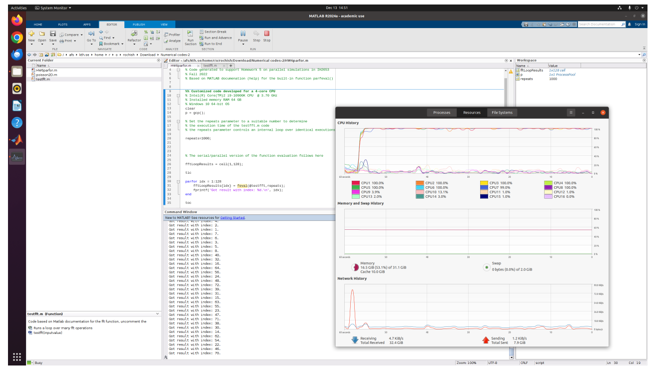
```
Require: p = gcp()                                      ▷ Should be commented for serial CPU execution
Require: tasks = 128
Require: repeats = 1000
Ensure: fftLoopResults = |fft(X,n)/n|
  fftLoopResults ← array(1, tasks)
  Start timer
  for i ← 1 to tasks do                                 ▷ Replaced with parfor for parallel CPU execution
    for j ← 1 to repeats do
      F_s ← 2^18                                        ▷ Default is 2^16
      t ← array(−0.5 : 1/Fs : 0.5)                      ▷ Replaced with t ← gpuArray(−0.5 : 1/Fs : 0.5) for parallel GPU execution
      L ← length(t)
      X ← 1/(0.4·√(2·π)) · exp(−t.^2/(2·0.01))          ▷ t.^2 evaluates positive next powers of 2, for the gaussian pulse X with σ = 0.1s
      n ← 2^nextpow2(L)                                 ▷ nextpow2(L) evaluates exponent (x) ∋ 2^x ≥ |L|, useful when |L| ≠ 2^x
      Y ← fft(X, n)                                     ▷ fft(X, n) evaluates n-point DFT
      f ← F_s · array(0:n/2)/n
      P ← |Y/n|
    end for
    fftLoopResults[i] ← P
    Print('Got result with index: %d.', i)
  end for
  End timer
```

FFT pseudocode adapted from **[1] MATLAB fft() Documentation**

Summary

1. Computer system configuration used in SAL 207 at KTH Kista:
   - CPU: `11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz (8 cores, 2 threads per core)`
   - Installed RAM: `32 GB (31.1 GiB usable)`
   - GPU: `NVIDIA T1000 (GDDR6: 4 GB)`
   - OS: `Ubuntu 20.04.6 LTS, 64-bit`
2. Table summarising standardised FFT evaluation speed comparison on various hardware processes:

	Parallel execution with CPU 128 tasks, 1000 repeats, $F_s = 2^{18}$	**Serial execution with CPU** 128 tasks, 1000 repeats, $F_s = 2^{18}$	**Serial execution with GPU** 128 tasks, 1000 repeats, $F_s = 2^{18}$	**Parallel execution with GPU** 128 tasks, 1000 repeats, $F_s = 2^{18}$
Average memory allocated	~ 16.4 GiB (with ~ 1.3 GiB in each of 8 operational cores)	~ 4.8 GiB (with ~ 2.4 GiB in 1 operational core)	~ 4.6 GiB (with ~ 2.2 GiB in 1 operational core)	~ 17.7 GiB (with ~ 1.4 GiB in each of 8 operational cores)
Loop execution order	Unpredictable (follows generic increasing index)	Serial increase	Serial increase	Unpredictable (follows generic increasing index)

Elapsed time	~ 369 seconds	~ 703 seconds	~ 264 seconds	~ 253 seconds
Screenshot of system monitor				

1. From the table above -
   - Parallel execution with CPU refers to multiprocessing on CPU across 8 operational cores / 16 threads, as alloted by the OS.
     - Since multiple cores are being utilised, average memory consumption is higher than in serial execution, to allot and aggregate data from the CPU cores. While memory used per core is lowest than in all cases, due to multiprocessing and no interfacing requirement with GPU.
   - Serial execution with CPU refers to sequential flow of data processing in one CPU core in a queue-like fashion, and can be multi-threaded as alloted by the OS.
     - Since a single core is being utilised, average memory consumption is lower than in parallel execution, for data allocation and aggregation. While memory used per core is highest than in all cases, due to entire workload on the single operational core.
     - This results in the highest execution time than in all cases.
   - Serial execution with GPU refers to sequential flow of data processing in one CPU core while interfacing the CUDA-enabled GPU for data-parallel applications (matrix multiplies, linear solvers). Since serial GPU acceleration requires data flow from a single CPU process it is single-threaded.
     - The slight decrease in average memory consumption and memory used by each core (and a major decrease in execution time) when compared to serial execution with CPU, is emblematic of GPU acceleration.
   - Parallel execution with GPU refers to a combination of CPU multiprocessing and interleaved CUDA-enabled GPU processing, for maximum degree of parallelisation.
     - Since multiple cores are being utilised while interfacing GPU, average memory consumption is highest than in all cases, to allot and aggregate data from CPU and GPU cores. While memory used by each core is slightly higher when compared to Parallel execution with CPU, due to the interfacing requirement with GPU.
     - This results in the lowest execution time than in all cases.

Part II: FDM

CPU vs GPU execution

Algorithm 2 Adapted FDM pseudocode for 1D/2D Poisson solution for RHS of pn-junction, using MATLAB

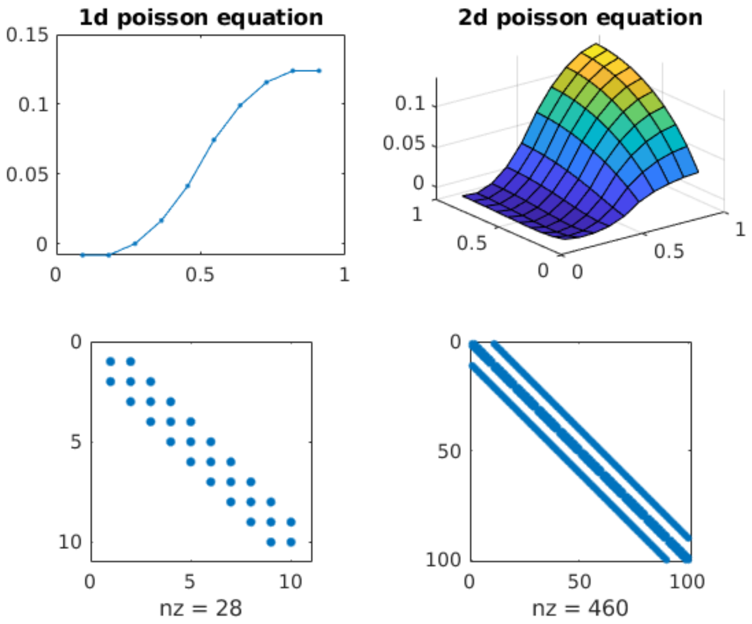
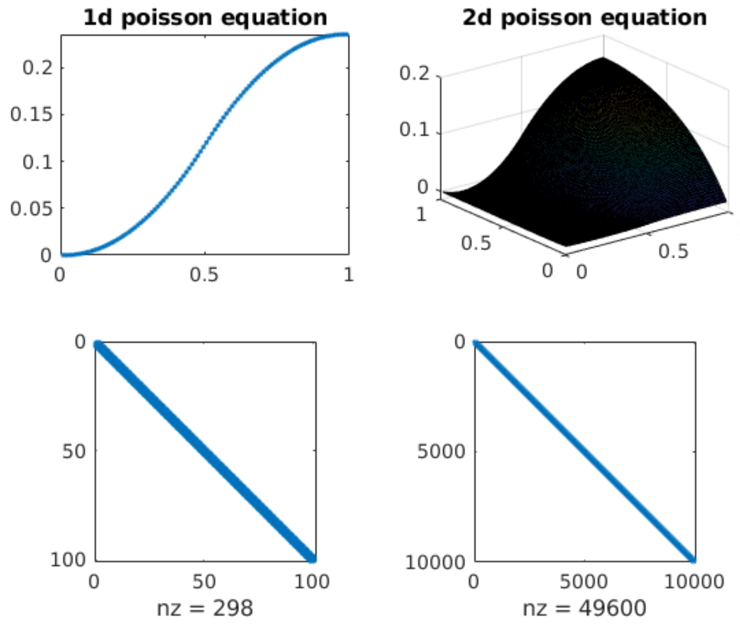
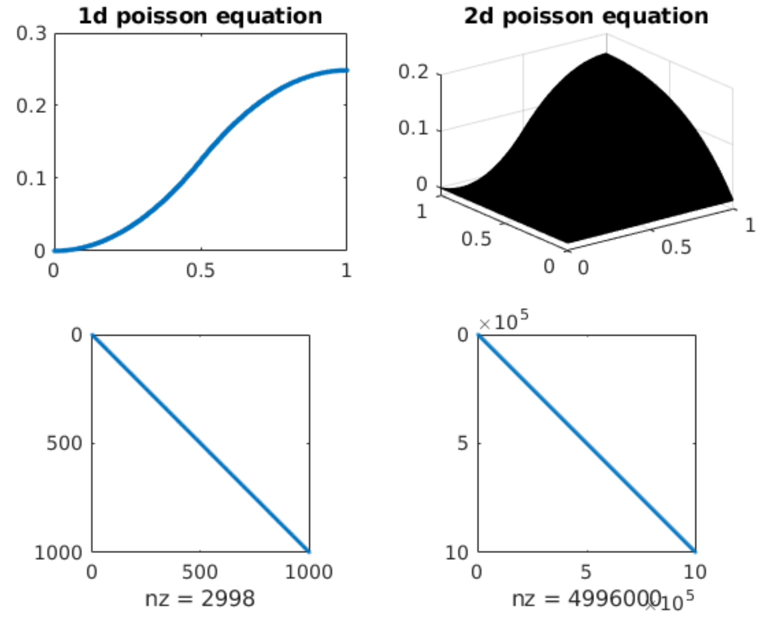
Require: n ▷ # of even discretisation nodes
  $h \leftarrow 1/(n+1)$
Ensure: $u2D = K2D \backslash f2D$, $u1D = K1D \backslash f1D$
  $K1D \leftarrow spdiags(ones(n,1) \cdot [-1, 2, -1], -1:1, n, n)$ ▷ 1D Poisson matrix
  $K1D[end, end] \leftarrow 1$
  $I1D \leftarrow speye(size(K1D))$ ▷ 1D Identity matrix
  $f1D \leftarrow h^2 \cdot ones(n,1)$ ▷ 1D RHS
  $f1D[1:end/2] \leftarrow f1D[1:end/2] \cdot -1$ ▷ for constant 1D pn-junction RHS
  $f1D[end] \leftarrow 0$
  $u1D = K1D \backslash f1D$ ▷ Poisson solution for linear equation system: $K1D = f1D \cdot u1D$

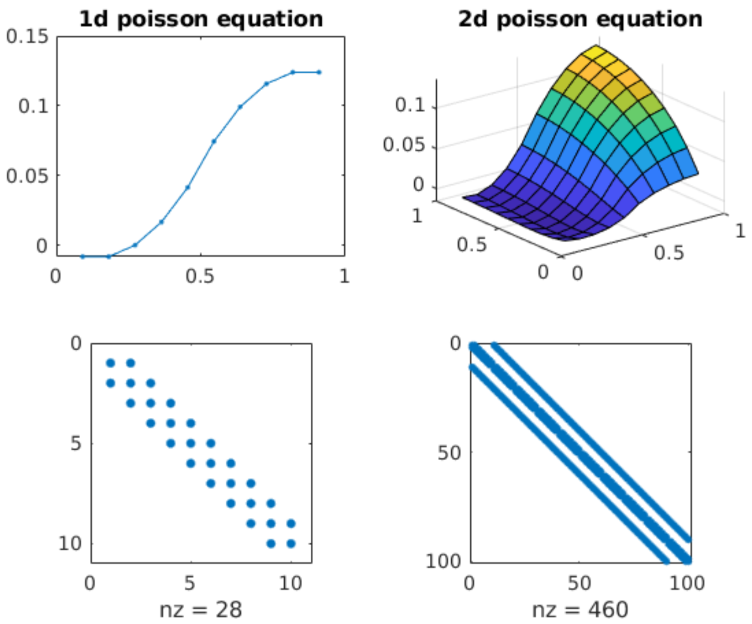
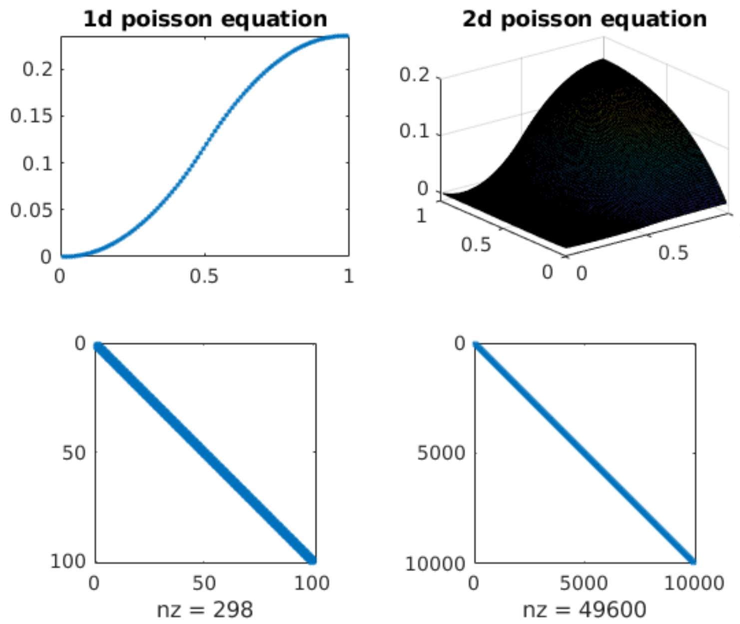
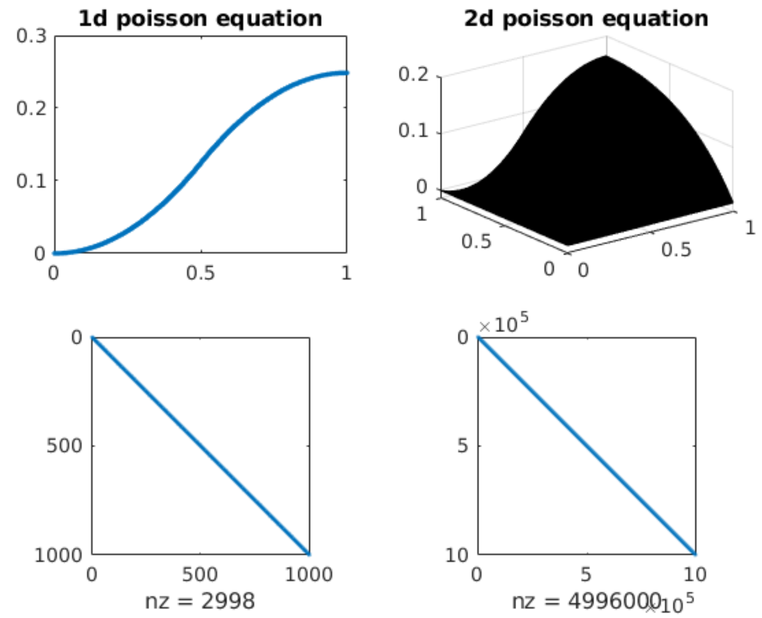
  if CPU execution case **then**
    $K2D \leftarrow kron(K1D, I1D) + kron(I1D, K1D)$ ▷ 2D sparse Poisson matrix, constructed by 1D Kronecker products
    $f2D \leftarrow h^2 \cdot ones(n^2, 1)$ ▷ 2D RHS
  else if GPU execution case **then**
    $K2D \leftarrow gpuArray(kron(K1D, I1D) + kron(I1D, K1D))$
    $f2D \leftarrow h^2 \cdot gpuArray.ones(n^2, 1)$
  end if
  $f2D[1:end/2] \leftarrow f2D[1:end/2] \cdot -1$ ▷ for constant 2D pn-junction RHS

  Start timer
  $u2D \leftarrow K2D \backslash f2D$ ▷ Poisson solution for linear equation system: $K2D = f2D \cdot u2D$
  End timer

Poisson solution pseudocode adapted from original code provided by **[2] Benjamin Seibold**

Summary

	n = 10	n = 100	n = 1000
CPU execution			
Elapsed time (s)	0.000162	0.010136	1.683117

	n = 10	n = 100	n = 1000
GPU execution			
Elapsed time (s)	0.002004	0.062535	11.056630

1. The solution for system of linear equations $B = A * x$ (i.e. K2D\F2D in the pseudocode above) is found using the `mldivide` algorithm in MATLAB **[3]** (which gives more accurate solutions than matrix inversion algorithms). `mldivide` algorithm is partially parallelisable albeit on CPU (exploited through vector processing units operating on instruction-level parallelism / SIMD **[4])** and it is a memory-intensive algorithm (hence A needs to be a sparse matrix).
2. GPU execution of `mldivide` however cannot provide any meaningful speed-up due to negligible data-level parallelism in the `mldivide` matrix decomposition algorithms, hence GPU execution happens iteratively while communicating among various GPU cores to get to an acceptable solution. Hence elapsed time is higher for GPU executions than in CPU, while the calculated poisson solutions are identical in both CPU and GPU executions.

References

- [1] **MATLAB fft() Documentation**
- [2] **Benjamin Seibold**
- [3] **MATLAB mldivide() Documentation**
- [4] **Computer Architecture, Fifth Edition: A Quantitative Approach - Hennessy, John L. and Patterson, David A. [Pages 261-333 for SIMD]**

Additional information

Created by: Rochish Manda, MSc KTH

IH2653 Examiner: Dr. Gunnar Malm, Professor KTH

Data and config file at: **Github**