

Displayed in Fig. 1 is the resulting convergence data recorded under successive refinements of the grid.

h	$\ u - u_h\ $	$\log_2(e_h/e_{\frac{h}{2}})$
0.10000	0.000780	\emptyset
0.05000	0.000194	2.00698
0.02500	0.000048	2.00174
0.01250	0.000012	2.00044

Fig. 1: Convergence of approximate solution u_h computed on a GPU under successive mesh refinements.

Then, the GPU code is tested against the original serial code. We record the time it takes to complete all time stepping. Fig. 2 shows the result of the run time tests and we can see that the GPU code, though it slows down on larger inputs, performs better than the CPU code on large inputs.

h	CPU run time (s)	GPU run time (s)
0.005	0.004179153	0.003549657
0.001	0.119557278	0.016512015
0.0002	2.922218388	0.087218712s

Fig. 2: Run time comparison for GPU and CPU codes.

my_forward __EulerGPU!()

```

1  using SparseArrays
2  using CuArrays
3  using CUDAnative
4  using CUDAdrv: synchronize
5  include("addVectors.jl")
6  include("rates.jl")
7
8
9  # compute  $y = Ax + b$ 
10 function knl_gemv_naive!(A, x, b)
11     N = size(A, 1)
12
13     @assert length(x) == N
14     @assert length(b) == N
15
16     y = zeros(N)
17
18     for i = 1:N
19         for j = 1:N
20             y[i] += A[i, j] * x[j] # + b[i]/N
21         end
22         # which is faster, including  $b_i$  in the j loop or appending it
23         # just outside the j loop by
24         y[i] += b[i]
25     end
26
27     return y
28
29
30 end
31
32 function knl_gemv!(b, A, x)
33     N = size(A, 1)
34
35     @assert length(x) == N
36     @assert length(b) == N
37
38     bidx = blockIdx().x # get the thread's block ID
39     tidx = threadIdx().x # get my thread ID
40     dimx = blockDim().x # how many threads in each block
41
42     bidy = blockIdx().y # get the thread's block ID
43     tidy = threadIdx().y # get my thread ID

```

```

44     dimy = blockDim().y # how many threads in each block
45
46
47     # figure out what work i need to do
48     i = dimx * (bidx - 1) + tidx
49     j = dimy * (bidy - 1) + tidy
50
51     if i <= size(A, 1) && j <= size(x, 2)
52         for k = 1:N
53             b[i, j] += A[i, k] * x[k, j]
54         end
55     end
56
57
58 end
59
60 #=
61 let
62
63     M = 10
64     N = M
65     Q = 1
66
67     A = rand(M, N)
68     x = rand(N, Q)
69     b = rand(N, Q)
70
71     d_A = CuArray(A)
72     d_x = CuArray(x)
73     d_b = CuArray(b)
74     d_bcopy = similar(d_b)
75
76
77
78     num_threads_per_block_x = 32
79     num_threads_per_block_y = 32
80     thd_tuple = (num_threads_per_block_x, num_threads_per_block_y)
81
82     num_blocks_x = cld(M, num_threads_per_block_x)
83     num_blocks_y = cld(Q, num_threads_per_block_y)
84
85     #matmul!(C, A, B)
86     #fake_knl_matmul!(C, A, B, num_threads_per_block_x, num_threads_per_block_y, num_blocks_x, num_blocks_y)
87
88     @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_bcopy, d_A,

```

```

89     synchronize()
90
91     t_device = @elapsed begin
92         @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_b, d_A,
93             synchronize()
94     end
95     @show t_device
96
97     t_host = @elapsed begin
98         C0 = A * x + b
99     end
100
101
102     @show t_host
103
104
105     #@show norm(CuArray(C0) - d_C)
106     @assert CuArray(C0) ≈ d_b
107 end
108 =#
109
110 function F(x,t)
111     return 2*(pi^2 - 1)*exp(-2t)*sin.(pi*x)
112 end
113
114 function f(x)
115     return sin.(pi*x)
116 end
117
118 function exact(x,t)
119     return exp(-2t)*sin.(pi*x)
120     #return exp(-pi^2 * t)*sin.(pi*x)
121 end
122
123 function time_dependent_heat_GPU(k, Δx, Δt, T, my_source, my_initial)
124     N = Integer(ceil((1-0)/Δx)) # N+1 total nodes, N-1 interior nodes
125     x = 0:Δx:1
126     t = 0:Δt:T
127     M = Integer(ceil((T-0)/Δt)) # M+1 total temporal nodes
128
129     λ = Δt/Δx^2
130
131     # A is N+1 by N+1
132     A = (1-2*λ*k)*sparse(1:N+1,1:N+1,ones(N+1), N+1, N+1) +
133         (λ*k)*sparse(2:N+1,1:N,ones(N),N+1,N+1) +

```

```

134         (lambda*k)*sparse(1:N,2:N+1,ones(N),N+1,N+1)
135
136     # initial conditions
137     u = zeros(N+1,1)
138     u .= my_initial.(x)
139     u_serial = copy(u)
140
141     d_A = CuArray(A)
142     d_u = CuArray(u)
143     d_b = CuArray(zeros(Float64, N+1))
144
145     num_threads_per_block_x = 32
146     num_threads_per_block_y = 32
147     thd_tuple = (num_threads_per_block_x, num_threads_per_block_y)
148
149     num_blocks_x = cld(size(A,1), num_threads_per_block_x)
150     num_blocks_y = cld(size(u,2), num_threads_per_block_y)
151
152     F(x, t[1])
153     @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_b, d_A, d_u)
154
155     t_dev = @elapsed begin
156     for n = 1:length(t)
157         b = Δt * F(x, t[n])
158         d_b = CuArray(b)
159         @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_b, d_A,
160         synchronize()
161
162         d_u .= d_b
163
164         d_u[1,1] = 0
165         d_u[length(x),1] = 0
166     end
167     end
168     @show t_dev
169
170     t_serial = @elapsed begin
171     for n = 1:length(t)
172         b_serial = Δt * F(x, t[n])
173         u_serial .= A*u_serial + b_serial
174
175         u_serial[1] = 0
176         u_serial[length(x)] = 0
177     end
178     end

```

```

179     @show t_serial
180
181
182     # $\Delta x = 0.00125?$ 
183
184
185     return d_u
186 end
187
188
189 function my_forward_Euler!(Y,  $\Delta t$ , t, A, F, y, x, N, M)
190     #M = length(t)
191     #t = t1: $\Delta t$ :tf
192     #M = Integer(ceil((tf-t1)/ $\Delta t$ ))
193     Y[2:N,1] = y[:]
194
195     for n = 1:M
196         b =  $\Delta t$  * F(x[2:N],t[n])
197         y[:] = A * y[:] + b
198         Y[2:N,n] .= y[:]
199
200     end
201
202     #return (t, Y)
203
204 end
205
206
207 function ( $\Delta x$ )
208     k = 2
209      $\Delta x$  = 0.000125
210
211
212      $\lambda$  = 0.1
213      $\Delta t$  =  $\lambda * \Delta x^2$  / k
214     T = 3* $\Delta t$ 
215
216     x = 0: $\Delta x$ :1
217     t = 0: $\Delta t$ :T
218
219     u_h = time_dependent_heat_GPU(k,  $\Delta x$ ,  $\Delta t$ , T, F, f)
220
221     uexact = CuArray( exact(x, t[length(t)]))
222     #uexact = exact(x, t[length(t)])
223     error =sqrt.(  $\Delta x$  * (transpose(u_h-uexact) * (u_h-uexact)))

```

```
224     return error[1,1]
225 end
226
227 #convergence_rates(, 0.1, 4)
```