High Performance Computations for Random Network Models of Parental Vaccine Acceptance and Disease Spread using CuPy

Thalia Juarez Tamer Oraby Andras Balogh

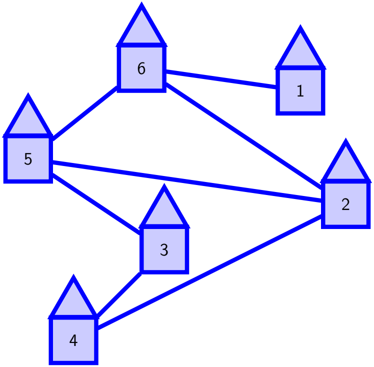
University of Texas Rio Grande Valley

July 3, 2020

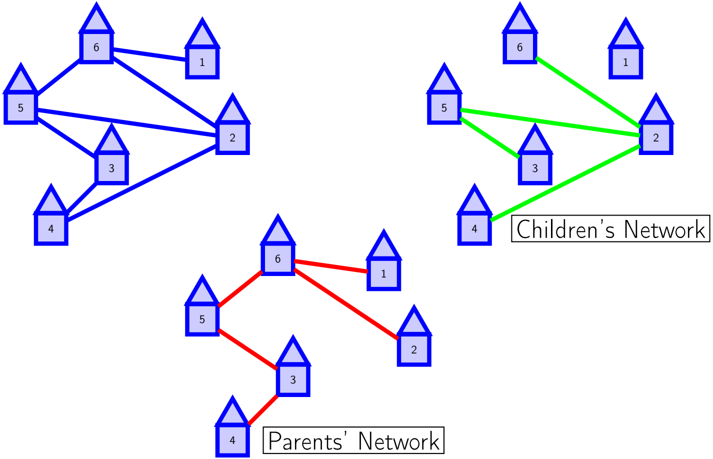
Previous work

- Computational implementation of coupled social and biological networks to examine
  - the influence of imperfect information on the diffusion of vaccination opinion
  - spread of vaccine-preventable pediatric diseases.
- Random network considered: Erdős-Rényi network
- High performance parallel stochastic simulations using Graphical Processing Units (GPUs) (2496 CUDA cores per GPUs).
- CuPy: CUDA and Python combined.

Network of Households (two overlapping networks)



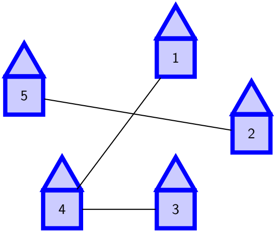
Children Spreading Infection, Parents Spreading Information about Disease and Vaccination



Network Settings (programmed previously)

- N – households
- A maximum number of M_c children placed into each household
- $0 \leq Children(i) \leq M_c$, for $1 \leq i \leq N$ (binomial distribution, $P_c = 0.5$ probability of having a child)
- Initially all these children are susceptible
- I_0 infected children are distributed among households randomly
- Erdős-Rényi model for the children's biological network: household i and j are connected with probability $P_{link}\sqrt{Children(i)Children(j)}$
- Parent's (Social) Network
  - Children's connections are retained with probability P_{ret}
  - New connections are added with probability P_{add}
- Separate epidemic/vaccination/birth processes go through the networks.

Adjacency Matrices (Children and Parents)



- $x_{ij} = 1$ if and only if households i and j are connected.
- Symmetric matrix with zero diagonal

$$\begin{bmatrix} \color{blue}{0} & 0 & 0 & \color{red}{1} & 0 & \dots \\ 0 & \color{blue}{0} & 0 & 0 & \color{red}{1} & \dots \\ 0 & 0 & \color{blue}{0} & \color{red}{1} & 0 & \dots \\ 1 & 0 & 1 & \color{blue}{0} & 0 & \dots \\ 0 & 1 & 0 & 0 & \color{blue}{0} & . \end{bmatrix}$$

Sparse Symmetric Matrix Storage

- Symmetric sparse matrix with zero diagonal

$$\begin{bmatrix} \color{blue}{0} & 0 & 0 & \color{red}{1} & 0 & \dots \\ 0 & \color{blue}{0} & 0 & 0 & \color{red}{1} & \dots \\ 0 & 0 & \color{blue}{0} & \color{red}{1} & 0 & \dots \\ 1 & 0 & 1 & \color{blue}{0} & 0 & \dots \\ 0 & 1 & 0 & 0 & \color{blue}{0} & . \end{bmatrix} \xRightarrow{\text{indexing}} \begin{bmatrix} . & 1 & 2 & \color{red}{4} & 7 & \dots \\ . & . & 3 & 5 & \color{red}{8} & \dots \\ . & . & . & \color{red}{6} & 9 & \dots \\ . & . & . & . & 10 & \dots \\ . & . & . & . & . & . \end{bmatrix}$$

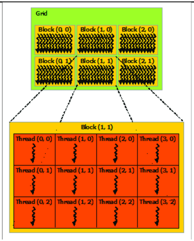
- Only the indices of nonzero elements in the upper part are stored: $k \in \{4, 6, 8, \dots\}$ (Children_mtx_indx, Parents_mtx_indx)
- We switch between indexing formulas as needed:

$$k = i + \frac{(j-2)(j-1)}{2}$$

$$\text{col index } j = \text{floor}\left(\frac{3+\sqrt{8k-7}}{2}\right)$$

$$\text{row index } i = k - \frac{(j-2)(j-1)}{2}$$

Parallel computations using CUDA on Graphical Processing Units

NVIDIA Tesla K40 Computing Accelerator 4.2 TFLOPS 12GB memory	2880 CUDA cores maximum size of a thread block $(1024, 1024, 64)$
	

CUDA/GPGPU

CUDA

- CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on its own GPUs.
- CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation.

GPGPU

- General-purpose GPU computing (GPGPU) is the use of a GPU to do general purpose computing.
- The model for GPU computing is to use a CPU and GPU together in a heterogeneous co-processing computing model.
    - The sequential part of the application runs on the CPU.
    - The computationally-intensive part is accelerated by the GPU.

CUDA with different programming languages

- Originally CUDA is written in C/C++. Problem: matrices and matrix operations
- CUDA Fortran: very convenient handling of matrices; object oriented, but not popular anymore on the job market
- Python: popular in both academia and business, easy to read
- Python with CUDA: CuPy
  - NumPy-compatible matrix library accelerated by CUDA
  - Example: cupy.flatnonzero() - indices of nonzero entries
  - Example: a + b - adding vectors/matrices in parallel
- Current CuPy code is based on previously written CUDA Fortran code
- Typically, every process is done simultaneously (parallel) for each household.
- CUDA C++ kernel functions are still useful.

Disease progression (daily process)

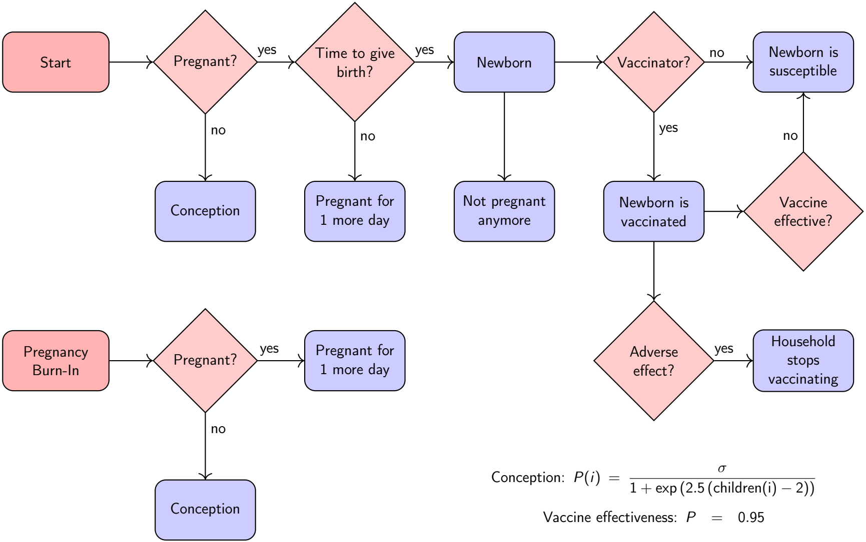
Immunized ← Susceptible → Infected → Recovered

- New Infections in a household i
  - $\beta = 0.01$ – transmission probability <u>between</u> households
  - $\beta_h = h\beta = 2\beta$ – transmission probability <u>within</u> households

$$P_{\text{infection}}(i) = 1 - (1-\beta)^{\frac{\textbf{infected connections}(i)}{\textbf{Children}(i)}} (1-\beta_h)^{\text{Infected(i)}}$$

  - Infected connections are based on the children's network
- Recovery period: based on discretized gamma distribution. Mean: 22 days; Maximum: 28 days (Measles)
  - An infectious recovers on day $0 \leq j \leq 28$ with probability $P(j)$
  - $P(28) = 1$

Pregnancy and Newborns (daily process)

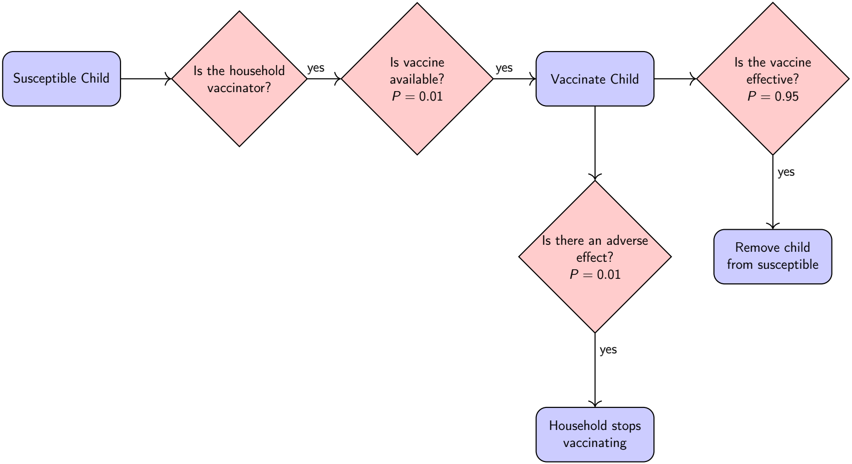


$$\text{Conception: } P(i) = \frac{\sigma}{1 + \exp\left(2.5\left(\text{children(i)} - 2\right)\right)}$$

$$\text{Vaccine effectiveness: } P = 0.95$$

$$\text{Vaccine adverse effect: } P = 0.01$$

Vaccination (daily process)



Probability of household i to vaccinate (daily process)

- Probability to vaccinate against the disease without any social influence, based on only the pay-off between the dangers of infection and the dangers of adverse effects:

$$p_0 = \frac{1}{1 + \exp\left(\gamma(\text{total adverse events}) - \alpha(\text{total infected})\right)}$$

- Probability to vaccinate:

$$P_V(i) = \frac{p_0 \cdot q^{n_V(i)} \cdot (1-q)^{n_N(i)}}{p_0 \cdot q^{n_V(i)} \cdot (1-q)^{n_N(i)} + (1-p_0) \cdot (1-q)^{n_V(i)} \cdot q^{n_N(i)}}$$

- q – probability that parents give the correct/honest signal on their stance/attitude toward vaccination
- $n_V(i)$ – number of socially connected vaccinating households
- $n_N(i)$ – number of socially connected non-vaccinating households

Description of CuPy Code: library imports

```
import numpy as np # numerical library
import cupy as cp # CUDA accelerated library
import os  # for reading random seed from OS
import sys # for stopping code if needed
import math # for ceiling functions
from scipy import stats
import matplotlib.pyplot as plt
from kernel_functions import * # our kernel functions
    (CUDA c++)

```

Reading network data of children and parents

```
data=np.load("erndata.npz")

N=np.asscalar(data['N'])
NC=np.asscalar(data['NC'])
NP=np.asscalar(data['NP'])
Plink=np.asscalar(data['Plink'])
Pret=np.asscalar(data['Pret'])
Padd=np.asscalar(data['Padd'])
I0=np.asscalar(data['I0'])
Children_mtx_indx=cp.asarray(data['Children_mtx_indx'
    ],dtype=cp.int64)
Parents_mtx_indx=cp.asarray(data['Parents_mtx_indx'],
    dtype=cp.int64)
AllInfected=cp.asarray(data['Infected'],dtype=cp.int32
    )
Susceptible=cp.asarray(data['Susceptible'],dtype=cp.
    int32)

Children=cp.add(AllInfected,Susceptible)
```

Setting the parameters

```
q=0.5     # Probability of signal matching opinion
rho=0.01 # Probability of vaccination access
Padv=0.01  # Probability of adverse effect
aalpha=10**(-4) # Household view of infection
ggamma=0.1  # Household view of adverse effect
bbeta=0.01  # Transmission rate between households
bbetah=2*bbeta  # Transmission rate within households
NV0=0.05 # Proportion of all-time never-vaccinator
    households
Peff=0.45  # Probability of vaccine effectiveness
ssigma=0.005 # Birth rate
gestation=280
MaxDays=1000
ip=28    # Days it takes to recover

Infected=cp.zeros((N,ip),dtype=cp.int32)
```

Recovery process follows gamma distribution

```
shape = 22; disc=np.arange(ip+1) # ip=28; Measles
#shape = 11.5;  ip=16; disc=np.arange(ip+1); Flu
cumprob=stats.gamma.cdf(disc, shape, scale=1)
Pincubtrans=cp.asarray((cumprob[1:ip+1]-cumprob[0:ip])
    /(cumprob[ip]-cumprob[0:ip]), dtype=cp.float32)
```

Arrays

```
Pregnancy=cp.zeros(N, dtype=cp.int32)
Vaccinator_yesnonever=cp.zeros(N, dtype=cp.int32)
Nbrvacc_yes=cp.zeros(N, dtype=cp.int32)
Nbrvacc_no=cp.zeros(N, dtype=cp.int32)
Adverse=cp.zeros(N, dtype=cp.int32)
Vaccinated_new=cp.zeros(N, dtype=cp.int32)
Vaccinated=cp.zeros(N, dtype=cp.int32)
Recovered=cp.zeros(N, dtype=cp.int32)
InfNeighb=cp.zeros(N, dtype=cp.int32)
Infected_Total=cp.zeros(MaxDays, dtype=cp.int32)
Daily_Incidence=cp.zeros(MaxDays, dtype=cp.int32)
Daily_Vaccinations=cp.zeros(MaxDays, dtype=cp.int32)
Daily_Suscep=cp.zeros(MaxDays, dtype=cp.int32)
Daily_Vaccinators=cp.zeros(MaxDays, dtype=cp.int32)
Daily_NonVaccinators=cp.zeros(MaxDays, dtype=cp.int32)
Daily_Recovered=cp.zeros(MaxDays, dtype=cp.int32)
Daily_P0=cp.zeros(MaxDays, dtype=cp.float32)
Daily_Children=cp.zeros(MaxDays, dtype=cp.int32)
PV_info=cp.zeros(N, dtype=cp.float32)
P_infection=cp.zeros(N, dtype=cp.float32)
Infected[:,1]= AllInfected
```

Pregnancy Burn-In

```
blocksize_x = 1024 # maximum size of 1d block is 1024 threads
blocks=(blocksize_x,1,1) # number of blocks in the grid to cover all indices see 'grids' later
grids=(math.ceil(N/blocksize_x),1,1) # set grid size N

seed=int.from_bytes(os.urandom(4),'big') # Random seed from OS
# Uses the number of existing children and birth rate (ssigma) to create pregnancies at different stages in each household.
# Pregnancy[i] = 0 - no pregnancy
# Pregnancy[i] = j, 0 < j < gestation - jth day of pregnancy
pregnancy_burn_in(grids, blocks, (N, cp.float32(ssigma), seed, Children, Pregnancy, gestation))
```

Pregnancy Burn-In kernel function

```
import cupy as cp

pregnancy_burn_in = cp.RawKernel(r'''
    #include <curand_kernel.h>
    extern "C" __global__
    void pregnancy_burn_in(const int N, const float ssigma,
    int seed, const int* Children,
                    int* Pregnancy, const int gestation){
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        int j, seq, offset;
        seq = 0;
        offset = 0;
        curandState h;
        if(i<N){
        curand_init(seed+i,seq,offset,&h);
          for(j=1; j < gestation; j++){
              if(Pregnancy[i]>0) { ++Pregnancy[i]; }
              else if( curand_uniform(&h) < ssigma/(1.0+exp
    (2.5*(Children[i]-2)))) { Pregnancy[i]=1; }
          }
        }
    }
    ''', 'pregnancy_burn_in', backend='nvcc')
```

Vaccinators_Init

```
seed=int.from_bytes(os.urandom(4),'big')
# Initial set up of vaccinators and non-vaccinators, based
    on the initial infected (I0)
# Vaccinator_yesnonever[i] = 0/1, family i doesn't vaccinate
    /does vaccinate. Can change, see Vaccinator_update
Vaccinators_Init(grids, blocks, (N, cp.float32(1.0/(1.0+np.
    exp(-aalpha*I0))),seed,Vaccinator_yesnonever))
```

Vaccinators_Init kernel function

```
Vaccinators_Init = cp.RawKernel(r'''
    #include <curand_kernel.h>
    extern "C" __global__
    void Vaccinators_Init(const int N, const float VProb,
   int seed, int* Vaccinator_yesnonever){
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        int seq, offset;
        seq = 0;
        offset = 0;
        curandState h;
        if(i<N){
            curand_init(seed+i,seq,offset,&h);
            if(curand_uniform(&h)<VProb) {
   Vaccinator_yesnonever[i] = 1; }
            else { Vaccinator_yesnonever[i] = 0; }
        }
    }
    ''', 'Vaccinators_Init', backend='nvcc')
```

```
# Vaccinator_yesnonever[i] = -1
# NV0 proportion of non-vaccinators

# Number of vaccinators
Nvacc=np.asscalar(cp.count_nonzero(Vaccinator_yesnonever).
    get())

 # Calculates the number of never-vaccinators
 N_nevervacc=int(round(NV0*(N-Nvacc)))

 # Indices of non-vaccinators, min value of
    Vaccinator_yesnonever is 0
idx_Nonvacc=cp.argmin(Vaccinator_yesnonever)

# From the number of non-vaccinators (N-Nvacc), choose
    randomly N_nevervacc indices (never vaccinators)
idx_Nevervacc=cp.random.choice(N-Nvacc, size=N_nevervacc,
    replace=False, p=None)

# Place never-vaccinators (-1) into Vaccinator_yesnonever
cp.put(Vaccinator_yesnonever, idx_Nevervacc, -(cp.ones(
    N_nevervacc,dtype=cp.int16)))
```

```
while (Infected_Total[day].get() > 0):
    day = day + 1
    # Marks households NbrNbrvacc_yes=0/1 & Nbrvacc_no=0/1
    grids=(math.ceil(N/blocksize_x),1,1) # set grid size N
    Vaccinators_Separate(grids, blocks, (N,
    Vaccinator_yesnonever, Nbrvacc_yes, Nbrvacc_no))

    # Counts vaccinating/non-vaccinating neighbors
    grids=(math.ceil(NP/blocksize_x),1,1) # set grid size NP
    Pressure_Update(grids, blocks, (NP, Parents_mtx_indx,
    Vaccinator_yesnonever, Nbrvacc_yes, Nbrvacc_no))

    # p0 - probability to vaccinate without social influence
    p0=1.0/(1.0+np.exp(ggamma*np.asscalar(cp.sum(Adverse).
    get())-aalpha*np.asscalar(cp.sum(Infected_Total).get()))
    )

    grids=(math.ceil(N/blocksize_x),1,1) # set grid size N
    # PV_info - probability to vaccinate with social
    influence
    pv_info_update(grids, blocks, (N, PV_info, cp.float32(p0
    ), cp.float32(q), Nbrvacc_yes, Nbrvacc_no,
    Vaccinator_yesnonever))
```

Daily Process (continued)

```python
    # Update vaccinators based on PV_info. Never-vaccinators will not change
    seed=int.from_bytes(os.urandom(4),'big')
    Vaccinator_update(grids, blocks, (N, PV_info, seed, Vaccinator_yesnonever))

    # Vaccinates susceptibles if the household is a vaccinator and the vaccine is available
    # The vaccine is effective with probability Peff and causes adverse effects with probability Padv
    seed=int.from_bytes(os.urandom(4),'big')
    Vaccinated_new.fill(0) # Resets newly vaccinated
    Vaccinate_Susceptibles(grids, blocks, (N, Vaccinated_new, Vaccinated, Vaccinator_yesnonever, Susceptible, seed, cp.float32(rho),cp.float32(Padv), Adverse, cp.float32(Peff)))

    seed=int.from_bytes(os.urandom(4),'big')
    Pregnancy_Newborns(grids, blocks, (N, Vaccinated_new, Vaccinated, Vaccinator_yesnonever, Susceptible, seed, cp.float32(Padv), Adverse, cp.float32(Peff), Pregnancy, gestation, Children, cp.float32(ssigma)))
```

Pregnancy_Newborns kernel function

```
if(i<N){     // in parallel for each household i simultaneously
    curand_init(seed+i,seq,offset,&h); // initialize random seed for each thread
    if (Pregnancy[i]==0){  // no pregnancy, might get pregnant
        if(curand_uniform(&h) < ssigma/(1.0+exp(2.5*(Children[i]-2.0)))) { Pregnancy[i]= 1; }
    }
    // else, the household is pregnant for another day
    else if(Pregnancy[i]<gestation){ Pregnancy[i]=Pregnancy[i]+1; }
    else if(Pregnancy[i] == gestation) { // else, newborn
        Children[i]=Children[i]+1;
        Pregnancy[i]=0;
        if (Vaccinator_yesnonever[i]==1) {  // household vaccinates
            Vaccinated_new[i]=Vaccinated_new[i]+1;
            Vaccinated[i]=Vaccinated[i]+1;
            // the vaccine is NOT effective
            if(curand_uniform(&h)>Peff) { Susceptible[i]= Susceptible[i]+1; }
            // adverse effect of the vaccine
            if (curand_uniform(&h)<Padv){
                Adverse[i]=Adverse[i]+1;
                Vaccinator_yesnonever[i] = -1;
            }
        }
        // otherwise, if the household does not vaccinate
        else { Susceptible[i]= Susceptible[i]+1; }
    }
}
```

Daily Process (continued)

```
    seed=int.from_bytes(os.urandom(4),'big')
    Recover_Infected(grids, blocks, (N, Infected, Recovered, seed, Pincubtrans, AllInfected, ip))

    grids=(math.ceil(NC/blocksize_x),1,1) # set grid size NC
    InfNeighb.fill(0)
    Infected_Neighbors(grids, blocks, (NC, Children_mtx_indx, AllInfected, InfNeighb))

    grids=(math.ceil(N/blocksize_x),1,1) # set grid size N
    Pinfection_update(grids, blocks, (N, P_infection, InfNeighb, Children, AllInfected, cp.float32(bbeta), cp.float32(bbetah)))

    seed=int.from_bytes(os.urandom(4),'big')
    New_Infected(grids, blocks, (N, P_infection, Infected, Susceptible, seed, AllInfected, ip))
```

Daily Process Data Collection for Output

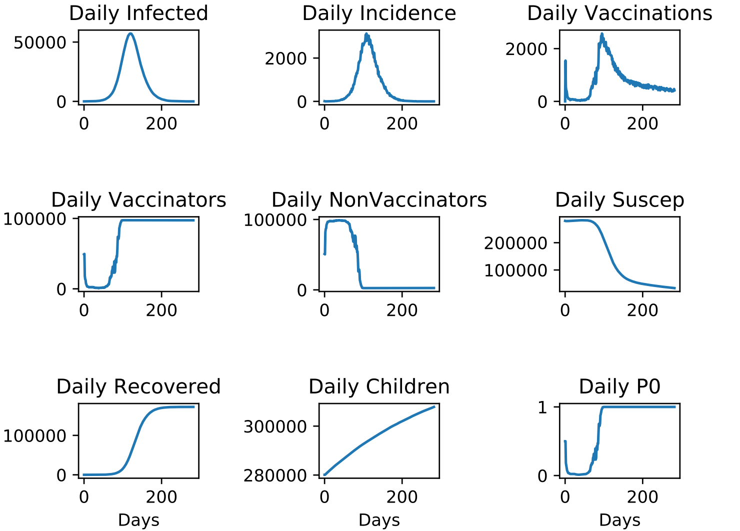
cupy.sum - summation is done in parallel on the GPU

```
    Infected_Total[day] = cp.sum(AllInfected)
    Daily_Incidence[day] = cp.sum(Infected[:,1])
    Daily_Vaccinations[day] = cp.sum(Vaccinated_new)
    Daily_Vaccinators[day] = cp.sum(Vaccinator_yesnonever > 0)
    Daily_NonVaccinators[day] = cp.sum(Vaccinator_yesnonever <= 0)
    Daily_Suscep[day] = cp.sum(Susceptible)
    Daily_Recovered[day] = cp.sum(Recovered)
    Daily_Children[day] = cp.sum(Children)
    Daily_P0[day] = p0
```

Plotting Results

```
Days=np.arange(day+1)
fig = plt.figure()
fig.subplots_adjust(hspace=1.5, wspace=1)

ax=fig.add_subplot(3,3,1)
ax.plot(Days,Infected_Total[0:day+1].get())
ax.set_title("Daily Infected")

ax=fig.add_subplot(3,3,2)
ax.plot(Days,Daily_Incidence[0:day+1].get())
ax.set_title("Daily Incidence")

...

ax=fig.add_subplot(3,3,9)
ax.plot(Days,Daily_P0[0:day+1].get())
ax.set_title("Daily P0")
ax.set_xlabel("Days")

plt.show()
fig.savefig("fig.pdf")
```

Sample Results



Timing Results

- For the epidemic process the speed of the CuPy code is comparable to the speed of the previous CUDA Fortran code, despite Python being an interpreted language.

N	Cuda Fortran	CuPy
$100,000$	$1.4s$	$4.7s$
$500,000$	$14.7s$	$16.5s$
$1,000,000$	$31.9s$	$37.4s$

- For the network generation the CuPy code is significantly faster than previous CUDA Fortran code. This is very important, as the stochastic computations require the calculation repeated thousands of time.

N	Cuda Fortran	CuPy	Speed up
$100,000$	$26s$ $(7.8GB)$	$18s$ $(9.1GB)$	44%
$500,000$	$626s$ $(8,7GB)$	$420s$ $(9.7GB)$	49%
$1,000,000$	$2500s$, $(8.7GB)$	$1700s$ $(10GB)$	47%

- The CuPy code is half as long as the CUDA Fortran code due to many of the NumPy-like library functions available in CuPy.

Summary

- We used the CuPy (CUDA + Python) open-source Python library with accelerated matrix and vector operations on NVIDIA GPUs for the stochastic simulation of an agent-based random network model.
- Two overlapping networks are considered, where the nodes represent households with parents and children in them.
- One network represents the children's physical connections through which disease spread.
- The other network represents the parents' social network though which information is exchanged about the disease and the vaccine.
- Initial results show a realistic disease process.

Future Plans

- Modify the code to run on multiple GPUs for population sizes larger than a million.
- Write code to calculate reproduction number R_0.
- Parameter analysis.
- Modify the code to run thousands of times for statistical purposes.
- Examine different diseases (measles, flu).
- Rewrite some of the kernel functions into CuPy code.

References

[1] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis.
Cupy: A numpy-compatible library for nvidia gpu calculations.
In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.