ANL-24/38

2023 AI Testbed Expeditions Report

Laboratory Directed Research and Development (LDRD)

Advanced Computing Expedition Leads:
Valerie Taylor, Ian Foster, Salman Habib, and Michael E. Papka

Computing, Environment and Life Sciences Directorate

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

ANL-24/38

2023 AI Testbed Expeditions Report

Laboratory Directed Research and Development (LDRD)

Prepared by:
Venkat Vishwanath, Murali Emani, Varuni Sastry, William Arnold, Rajeev Thakur

Computing, Environment and Life Sciences Directorate, Argonne National Laboratory

December 2023

Authors

Bryce Allen, Henry Chan, Rodrigo Ceccato de Freitas, Mathew J. Cherukara, Miaoqui Chu, Jose M. Monsalve Diaz, Neil Getty, Ross Harder, Kyle Hippe, Saugat Kandel, Antonino Miceli, Suresh Narayanan, Oleksandr Narykov, Alexander Partin, Nesar Ramachandra, Arvind Ramanathan, Esteban Rangel, Siddhisanket Raskar, Andrew Siegel, John Tramm, Thomas Uram, Azton Wells, Leighton Wilson, Fangfang Xia, Kazutomo Yoshii, Ruoxi Zhao, Tao Zhou

Table of Contents

Real-time Analysis for X-ray Photon Correlation Spectroscopy with Cerebras Wafer Scale Engine

Miaoqi Chu
Software Engineering 2

Suresh Narayanan
Physicist

Thomas Uram
Software Engineering 5

October 2023

1 Introduction to the Science problem

The dynamics in condensed materials systems play an essential role in their functions and properties. Such examples include the process of drying paint and the diffusion of functional proteins to deactivate a virus. Thus, a technique with high temporal sensitivity to the dynamics is valuable in the design of better materials and drugs.

At the Advanced Photon Source (APS), a new feature beamline is being built that specializes in probing dynamics in materials with X-ray photon correlation spectroscopy (XPCS). XPCS uses coherent X-rays, which will increase by 500x with the ongoing APS-U project, to penetrate the sample and obtain the dynamics information by collecting and correlating a time series of scattering images. As shown in Figure 1.a, the dynamics of nano/microparticles will result in the fluctuation of the scattering speckles, which can be used to extract both the dynamics and structural information. A typical XPCS measurement can take a thousand to a million images, each one with more than a million pixels, at frequencies up to 56 kHz. In addition, the data needs to be processed in a timely manner so that scientists can steer the direction of the next steps to maximize the opportunity for scientific discovery. The high data rate, large dataset size, and real-time processing requirements create a considerable challenge for conventional data processing pipelines.

This research tries to tackle the challenge with ALCF's AI-testbed system, Cerebras CS-2. CS-2 is equipped with more computation units and memory/cache than a CPU or a GPU, a potential candidate to enable high-performance correlation algorithms for XPCS

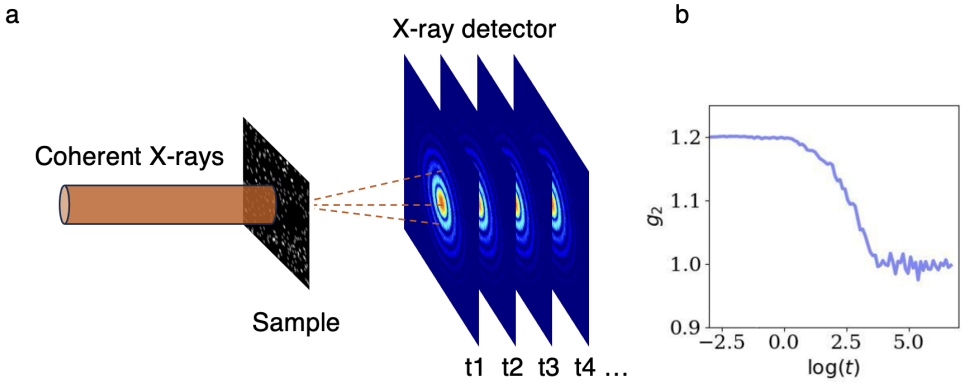


Figure 1: a. The schematic diagram for a typical XPCS measurement: a fast X-ray detector takes a time series of pictures of the scattering pattern, which originates from the nano/micro-particle dynamics illuminated by coherent X-rays. b. A typical plot of g2 correlation (defined Section 2) can be used to infer the dynamics of the material.

2 Description of the AI model and implementation

Our focus in this study is to implement the multi-tau algorithm for XPCS, one that is especially useful for studying equilibrium or steady-state systems. This algorithm computes the autocorrelation with constant delays, which can be summarized as,

$$g2 = \left\langle \frac{\langle I_{\mathbf{q},t} I_{\mathbf{q},t+\tau} \rangle}{\langle I_{\mathbf{q},t} \rangle \langle I_{\mathbf{q},t} \rangle} \right\rangle_{\mathbf{q},t}, \tag{1}$$

in which I, $\mathbf{q}$, t, τ are the scattering intensity, momentum transfer, time, and delay, respectively. The autocorrelation is averaged over the equivalent $\mathbf{q}$ and over all possible time t to increase the signal-to-noise (SNR) ratio. In addition, for longer delays, the raw signal is averaged first to improve the SNR and the robustness of the algorithm.

In order to run this algorithm on the CS-2, which is designed for deep learning (DL) models, we implement the algorithm in a layered structure that resembles a DL model, which is dubbed as **CorrModel**, as shown in Figure 2. The **CorrModel** consists of several correlation layers (**CorrLayer** shown in Figure 2 which is analogous to the standard layers used in DL such as **Linear** and **Conv2d**). Each layer defines persistent data structures, including *buffer*, *corr_result*, *frame_info*, along with other variables required for the correlation.

The XPCS scattering images are streamed sequentially into the first **CorrLayer** and stored in the buffer. When the buffer accumulates a certain number of frames, operators (OPs) are performed on the data: the correlation OP computes the multi-tau correlation and exports the partial result to *corr_result*; the sum OP computes the sum of frames which is used to normalize the correlation result in the final step; the average OP computes the averaged data which serves as the input for the next *CorrLayer*. After all images are processed, the correlation results are gathered from all **CorrLayer**s. Post-processing is then performed on the gathered data to compile the correlation curve (an example shown in Figure 1).

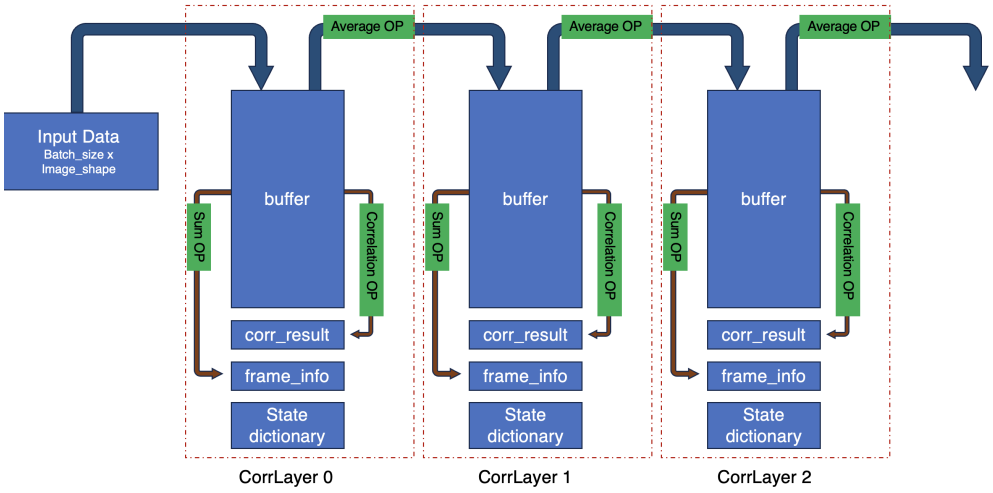


Figure 2: **CorrModel**, the multi-tau correlation algorithm implementation in a layered model (three layers shown) to make use of the DL software architecture available on CS-2. Each correlation layer consists of a buffer region to keep track of the previous frames and other data structures to enable the correlation computation. The time-series scattering data from an XPCS measurement can be wrapped in a *DataLoader*.

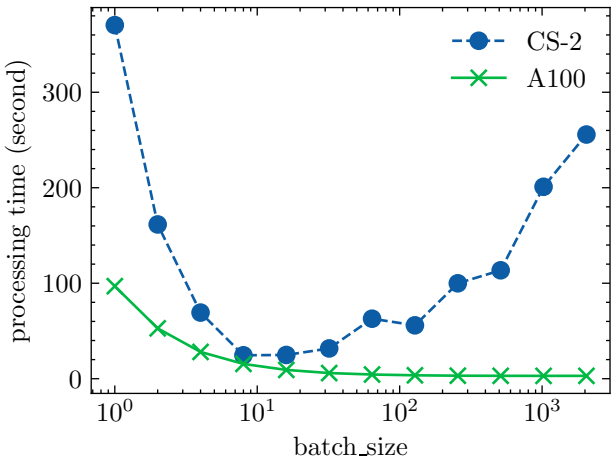


Figure 3: Performance of **CorrModel** running on a CS-2 and a Nvidia-A100 GPU on processing a reduced-size XPCS dataset as a function of *batch_size*.

3 What was needed to get the model running on the AI Accelerator

In addition to the layered-model implementation for the multi-tau algorithm described in the previous section, more modifications are needed to get it running on CS-2.

- While **CorrModel** resembles a DL model, its layers are not standard DL layers. Some operations in the layers need to be modified to meet the requirements of the target systems. For example, **CorrModel** does not return any output until the final step, which makes it impossible to compute the loss. To circumvent this problem, we perform the computation using the *evaluation* mode rather than the *training* mode, and a dummy loss function is provided to satisfy the platform's requirement on the target system.
- Some of the *pytorch* modules behave differently on CS-2, during this project. For example, the register buffer on CS-2 can't be accessed using the tensor/array slicing method. We have to convert the register buffer variables in a way that they're accessed as a whole. Standard layers such as *AvgPool1d* and *AvgPool2d* are not fully supported, which forced us to convert the average layer to a dense *Linear* layer.
- The experiment dataset is stored in a sparse representation in binary format. A customized *Dataset* class is needed to convert the dataset to a dense representation and eventually be wrapped in a *DataLoader* class.

4 Performance Evaluation

- We managed to run **CorrModel** on both the CS-2 and a GPU (Nvidia-A100) on a reduced XPCS dataset (65,536 channels out of 524,288 on the whole detector), as shown in Figure 3. While both devices finish the analysis within a reasonable time frame, CS-2 performs similarly to a GPU at $batch\_size = 8$, but is overall slower than an A-100. It's noted that the result is achieved with CorrModel implemented in an unoptimized way because of some missing features in CS-2, as discussed in the previous section. For example, the average OP is implemented as a dense *Linear* layer, which is very inefficient. In addition, since we only used 10 layers in **CorrModel**,
- As shown in Figure 4, **CorrModel** can perform the correlation accurately, yielding results that are almost identical to the standard implementation of the multi-tau algorithm. There are some finite

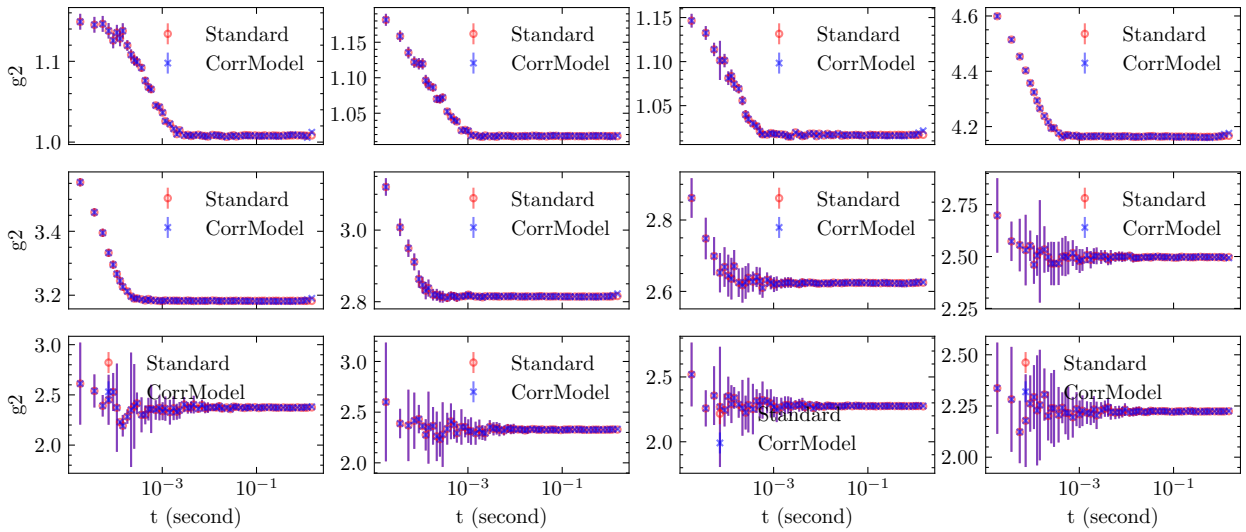


Figure 4: Correlation result comparison between **CorrModel** and our standard multi-tau algorithm. Overall, **CorrModel** generates accurate results of both the correlation values as well as their error bars, except a few data points at the end of the curve, which is caused by a different normalization procedure used in **CorrModel**.

differences between **CorrModel** and the standard result for the last few points. The origin is the use of a different normalization method in **CorrModel**. It's possible to fix this problem in future implementations.

5 Conclusion and next steps

In this project, we successfully implemented the multi-tau algorithm used at the APS with a layered model and performed accurate analyses on ALCF's CS-2 platform. The performance is on par with a professional GPU due to the compromise in the implementation with inefficient layers, despite the best of our efforts so far. We plan to explore other ALCF's testbeds, such as the Graphcore system, which is more flexible in creating customized layers. In addition, we also want to continue the development of **CorrModel** initialized in this research. With its modular design and easy portability to AI accelerators, **CorrModel** has excellent potential for future applications at XPCS beamlines.

6 Acknowledgements

The authors would like to thank Nicholas Schwartz, William (Bill) Allcock, Varuni Sastry, and Mathew Cherukara for their help on this project. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory, and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. This research also used resources of the Advanced Photon Source, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Basic Energy Sciences, under Contract No. DE-AC02-06CH11357.

Using AI accelerators for real-time training and feedback in x-ray ptychography experiments

Saugat Kandel (Postdoctoral appointee, APS), Tao Zhou (Assistant Scientist, CNM), Antonino Miceli (Physicist, APS), Mathew J. Cherukara (Computational Scientist, APS)

October 2023

1 Introduction to the Science problem

X-ray ptychography is a powerful lensless coherent diffraction imaging (CDI) technique that is widely used for nanoresolution imaging (2D or 3D) in the materials and life sciences, and is even a cornerstone of the APS scientific strategy post-upgrade. In this technique, a small coherent beam is used to illuminate a large sample sequentially in overlapping patches, with the diffraction pattern collected for each scan position. Subsequently, the entire set of diffraction patterns is used together for a simultaneous numerical reconstruction (phase retrieval) of the full sample at a high resolution. The traditional ptychography reconstruction procedure is computationally expensive and is difficult to use for real-time analysis in experiments with high data throughput. In contrast, machine learning techniques like PtychoNN can provide single-shot estimate for the object at each illumination position, and have the potential to provide real-time analysis capability that can keep up with the data acquisition rates[2]. We have recently demonstrated the use of HPC resources in conjunction with edge computing devices to achieve real-time object visualization in a ptychography experiment with data acquisition rates of 2 Khz for 128×128 pixel diffraction images (or >2 GBps)[1]. However, we expect the APS upgrade to enable a $100\times$ increase in the coherent flux, such that we will have diffraction images of size 512×512 pixels and with sufficient photon counts to enable data acquisition rates of >2 KHz.

One of the primary challenges in the use of NN for real-time data interpretation is to ensure that the NN model accurately represents the experimental conditions. In the PtychoNN context, this means that the object and illumination structure have to be similar to the experimental data used to train the PtychoNN model. However, it is often the case that we do not possess any prior data related to the object under study, and that the object structure can change either when we scan new regions of the object, or when we study dynamic phenomena. In these settings, if we want to make real-time decisions about the experiment (autonomously or manually), we cannot separate the training and inference into two distinct stages: the acquired data has to be continuously used to update the model, and the model for real-time inference and decision-making. Therefore, we want the training to be as fast as possible, and we expect that dedicated AI accelerators like the Cerebras CS2 will provide much faster training for large NN models than general compute hardware. Our prior work on the CS2 hardware has shown that a single CS2 can provide faster training than NVIDIA A100 GPUs for 128×128 pixel images. In this project, we explore the use of CS2 hardware for online training of larger PtychoNN models (for up to 512×512 pixels) and to map out a full integration of the accelerator into real-time ptychography workflow once the APS upgrade is complete.

2 Description of the AI model and implementation

For this work, we use a modified version of the PtychoNN2 model described in [1], which consists of a convolutional autoencoder architecture that uses a ptychographic diffraction image input to predict the phase of the illuminated object slice. In our version here, we adapt the model to 512×512 pixel images by increasing the number of blocks of convolutional layers. The desired model architecture is shown in Figure 1. The model contains $\approx$11.8 million trainable parameters. For mixed precision training with a single 512×512 pixel input image (or a batch size of 1) of size 1.05 MB, the model has a computational complexity of 10.5 GMACs and an estimated total size of 265 MB.

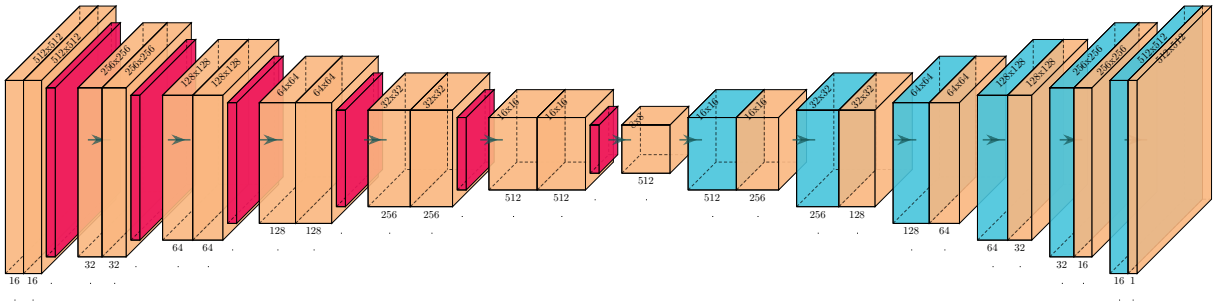


Figure 1: Schematic of the neural network structure. The orange boxes represent a convolutional layer, with the output image size indicated at the top and the number of convolutional filters at the bottom. The red boxes represent a maxpool layer. The blue boxes represent a transpose convolution with a stride of 2, which serve both as a convolutional layer and to increase by 2× the dimensionality of the input image. The model uses the Leaky ReLU activation in all the layers except the final layer that uses a tanh activation.

3 What was needed to get the model running on the AI Accelerator

Getting the model running on the CS2 accelerator required two separate steps: preparing the training dataset and dataloaders, and building the model architecture. The training datset consisted of 114 ptychographic scans, each with 963 diffraction images and the corresponding object slices, where each pair of diffraction image and object slice makes up the training input and output data. This makes up ≈220 GB of training data. Since practical implementations could require even larger training datasets, we cannot expect to load the full dataset into the computer memory, and therefore have to design a data loading procedure that can efficiently access the data from long-term storage. Instead of developing our own data loaders, we chose to adapt the U-Net HDF5 data loader provided in the Cerebras modelzoo. This overall required adapting the data loader and converting our data into HDF5 format with the appropriate structure. We tested this data loading paradigm on smaller NN models in the CS2 hardware and also with the full model (from Figure 1) in the LCRC A100 GPUs, and it performed well in both settings.

To test the model architecture, we started from a small PtychoNN model designed for 64×64 pixel input and outputs, then successively increased the model complexity to handle larger input images. We found that we were not able to compile the 128×128 pixel model with a batch size of > 8, the 256×256 model with a batch size of > 1, and not at all for the 512×512 model — the compilation failed with an internal error during the "Exploring data layouts" step. Furthermore, a 128×128 pixel model with a batch size of 16 would only require ≈210 MB memory, which is orders of magnitude less than that required for the Large Language Models that CS2 supports. It is therefore not clear why the compilation fails with the CS2 1.9.1 software. .

4 Performance Evaluation

We ran into three challenges in our attempt to use the CS2 hardware for performant continual learning with PtychoNN. First, as discussed in Section 2, we were unable to the larger PtychoNN models with sufficiently large batch sizes. Second, while CS2 is able to run Pytorch models in "eval" mode, switching between "train" and "eval" modes requires a full device reinitialization, which can take up to ≈10 minutes or more. This means that we cannot follow the standard training scheme, where we set aside a small amount of data to periodically check the model for overfitting — accomplishing this would require running a parallel instance of the model just for the validation checks. Third, a continual learning scheme assumes that we have to periodically add new data to the training dataset. The run scripts provided by Cerebras, however, seem to assume that the dataset remains unchanged; it is not clear how to reinitialize only the data loader and optimizer without reinitializing the full device. However, the simple performance benchmark

reported in (Table 1), which compares the CS2 performance with the performance of a NVIDIA A100 GPUs for compiled Distributed DataParallel models, shows that there is potential for significant acceleration in continual learning with the CS2 hardware if we can address these challenges in the future.

Device	**Batch size**	**Frames/s**
Cerebras	8	≈450
A100	64	≈300 per GPU

Table 1: Comparison of throughput for 128×128 pixel PtychoNN model on Cerebras and NVIDIA A100 GPUs. The training on the A100 GPUs used a compiled Distributed DataParallel model.

5 Conclusion and next steps

The CS2 shows great potential to accelerate workflows for x-ray microscopy which require fast, online training. Next steps:

- Continue working with ALCF and Cerebras teams to address compilation and eval mode issues.
- Implement the full online training workflow where the PtychoNN model is continuously updated using the CS2 during the experiment and trained models are pushed to the x-ray microscope for inference on edge computing devices[1].

6 Acknowledgements

This work was performed, in part, at the Advanced Photon Source, a U.S. Department of Energy (DOE) Office of Science User Facility, operated for the DOE Office of Science by Argonne National Laboratory under Contract No. DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We thank Varuni K. Sastry for her guidance and expertise in using the testbeds.

References

[1] Anakha V Babu, Tao Zhou, Saugat Kandel, Tekin Bicer, Zhengchun Liu, William Judge, Daniel J Ching, Yi Jiang, Sinisa Veseli, Steven Henke, et al. Deep learning at the edge enables real-time streaming ptychographic imaging. *arXiv preprint arXiv:2209.09408*, 2022.

[2] Mathew J Cherukara, Tao Zhou, Youssef Nashed, Pablo Enfedaque, Alex Hexemer, Ross J Harder, and Martin V Holt. Ai-enabled high-resolution scanning coherent diffraction imaging. *Applied Physics Letters*, 117(4), 2020.

Diffusion-Based Generative Model for Gene Expression Samples

Oleksandr Narykov (Postdoctoral Appointee), Alexander Partin (Computational Scientist 3)

October 2023

1 Introduction to the Science problem

Genetics-informed translational studies in the medical field are challenging because of the significant variability between individual organisms. In the case of pre-clinical studies, the problem is even more pronounced as there is a need to fill the gap between different biological models that vary in purity and availability. Previous studies addressed the problem of augmenting tumor gene expression data in the limited context of cancer sample classification using Generative Adversarial Networks (GANs). However, training this class of Neural Networks (NNs) is known to be challenging due to vanishing gradients, model collapse, and failures to converge. Diffusion models (DMs) are a cutting-edge advancement in the area of generative AI.

It is succeeding adversarial approaches, such as GANs and DMs, that rely on an iterative process of degrading data and denoising the result in advanced image, audio, and video generation areas. The performance of these models is robust. However, the generative process is known to be computationally intensive and slow. Recent OpenAI work addresses DMs' convergence speed issue by introducing a new class of DM – Consistency Models. Ultimately, we aim to leverage this architecture and AI Testbed resources to construct a generative model for the multiple biological models – cell lines, single-cell RNA-Seq samples from patients, and patient-derived xenografts (PDX). Generating robust biological data is an open problem that is plagued by challenges that are much less in traditional AI domains, e.g., image and audio, - lack of training samples, data inconsistency, high dimensionality, and poor interpretability. Overcoming those limitations is an undertaking that requires developing new strategies and approaches, so it is vital to be able to test them promptly. This project provides life science researchers with a way to create synthetic data based on samples cell line samples and obtain in silico samples for complex, realistic settings for further analysis and refinement.

2 Description of the AI model and implementation

This project focused on adopting CMs to generate synthetic RNA-Seq gene expression profiles and was tested in the cancer cell lines context. The main idea behind CMs is to learn a consistency function $f(x_t, t) \to x$, where x_t corresponds to the noisy data at the arbitrary timepoint t. This allows us to generate a ground-truth sample x from any point along the diffusion process in one step.

CMs portray a generative process as continuous in time instead of sticking to discrete steps that can be described with the stochastic differential equation (SDE)

$$dx_t = \mu(x_t, t)dt + \sigma(t)dw_t, \tag{1}$$

where $t \in [0, T], T > 0$ is a fixed timepoint, and $\mu(\cdot, \cdot)$ with $\sigma(\cdot)$ represent drift in the diffusion coefficients, and w_t is a Wiener process (Brownian motion). The model uses this continuous process to define diffusion over input data as

$$p_t(x) = p(x) \otimes \mathcal{N}(0, t^2 I) \tag{2}$$

The key point for this process is that we can obtain samples that are distributed according to $p_t(x)$ from the solution trajectories of the closed-form ordinary differential equation (ODE)

$$dx_t = [\mu(x_t, t) - \frac{1}{2}\sigma(t)^2 \bigtriangledown \log p_t(x_t)]dt \tag{3}$$

Originally, authors proposed to substitute $\bigtriangledown \log p_t(x_t)$ with the score that can be obtained from some other pre-trained generative model and use CMs to speed up computationally expensive diffusion models training process using the following equation:

$$\frac{dx_t}{dt} = -ts_\phi(x_t, t) \tag{4}$$

However, a much more interesting practical application of the algorithm comes from a standalone training process. The authors demonstrate that it is possible to estimate the score in the following way:

$$\bigtriangledown \log p_t(x_t) = \mathbb{E}\left[\frac{x_t - x}{t^2}|x_t\right] \tag{5}$$

This equation lies at the core of the current adaptation of CM to the biological field.

Original CMs have a UNet architecture and use ResNet blocks with convolutional layers as basic building steps and insert attention blocks that operate over standard keys, queries, and values entries. This design allows to chain outputs of the consistency models at multiple time steps, thus improving sample quality at the cost of increasing compute. Another critical point in CMs that separates them from regular UNets is the usage of the timestep embedding to keep track of the diffusion process.

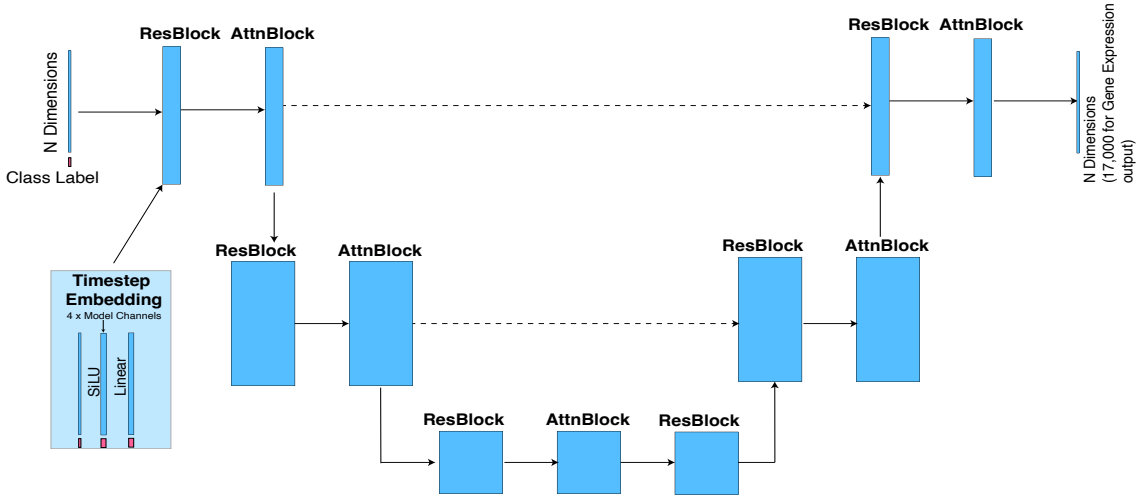


Figure 1: General Consistency Model structure. CM utilizes UNet structure to denoise data at multiple stages of the diffusion process.

CMs demonstrate spectacular performance in the field of computer vision. However, direct usage of convolutional layers is not appropriate for the RNA-Seq samples, as the proximity of two genes in the feature vector does not imply a relationship between their values. Because of this, we opted to adopt a feature transformer similar to the one used in TabNet - a deep learning model specializing in learning from tabular data. We incorporated timestep embedding into this architecture and added an additional connected layer at the start of the block to better capture relationships that can be learned from gene expression values.

3 What was needed to get the model running on the AI Accelerator

The project included multiple challenging details. First, the architecture of the CMs had to be adapted to the tabular data. Instead of following the convolutional architecture suitable for the image data, we adopt feature transformer elements of TabNet to navigate tabular data. Even though the basic layers that model requires were implemented, the architecture was quite different from Large Language Models (LLMs), which are the primary target of Cerebras. While the vision transformers are the upcoming focus of the release 2.0, the current process to make it work was challenging.

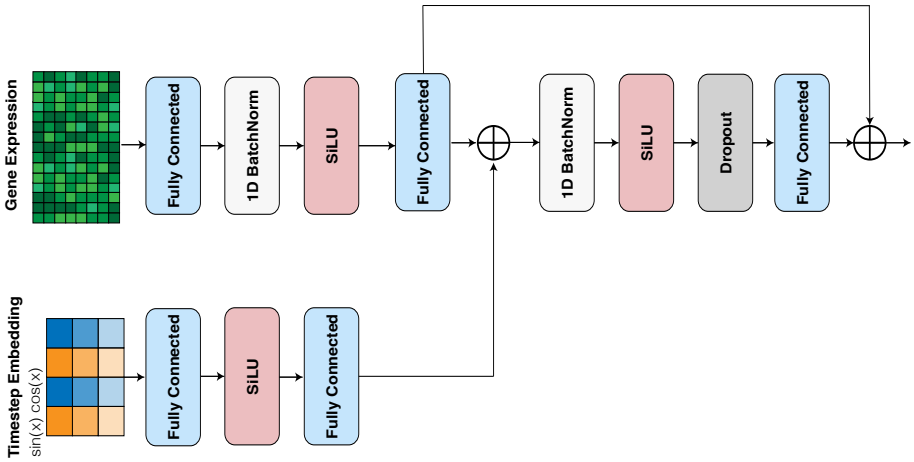


Figure 2: Structure of ResBlock feature transformer. The basis for the transformer is adopted from the TabNet model. However, it maintains positional encoding and dropout regularization from the Consistency Model.

First, the Cerebras builds a complete, static compute graph for compilation and execution that returns tensor values without JIT or CPU fallback. All PyTorch code has to operate entirely on torch.tensors and avoid calling functions that explicitly or implicitly convert data to a Python scalar, print tensor contents, or use a tensor as a Python conditional. Identifying all such instances in the code is not a trivial task, as compiler's messages are often low-level and usually are challenging.

Second, the debugging process was complicated as the Python Debugger's (PDB) usability was severely restricted due to compilation for different architecture, particularly access to the data. A large amount of errors was not detectable while using the CPU to run the model. The Cerebras compilation process is a multistage one, and PDB can be used only during the first phase. When encountering challenges in the later ones, the only approach that I found feasible was isolating parts of code and using surrogate data.

Third, not all common functionality is currently implemented on the Cerebras. For example, there is no problem generating tensors of zeros and ones. However, filling tensors with random numbers is not supported, which prompted me to create a separate dataloader for pre-generated random data.

4 Performance Evaluation

Several factors were considered for evaluating the generative model:

- Clustering algorithms' ability to distinguish between real-world and generated data
- UMAP embedding of the results
- Computation speed on GPU and Cerebras system

To ensure that our sampled data points are not easily distinguishable from the real-world counterparts, we perform multiple runs of clustering. The idea is to check whether an unsupervised algorithm is able to identify generated samples based solely on data structure in a dataset that combines 15xxx ground truth cancer samples and 10000 datapoints produced by our CM. We consider two algorithms - K-Means, Spectral Clustering, and DBSCAN. As K-Means and Spectral Clustering require to a pre-defined number of clusters, we explore outcomes for different values of this parameter - 2, 4, and 8. DBSCAN determines the number of clusters automatically.

Our evaluation strategy assigns ground truth labels to two possible memberships - real-world and sampled. Then we access the quality of clustering regarding the task of separating those memberships. Good clustering performance corresponds to the poor quality of sampling, and vice versa - poor clustering indicates that algorithms cannot pickup differences between real data and CM output. We compute the following metrics

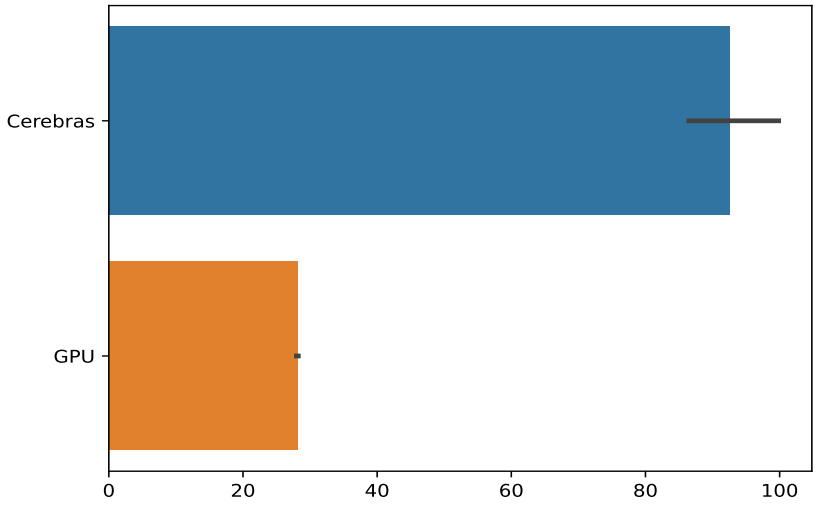


Figure 3: The sample rate for the batch size four for the Cerebras system and NVIDIA Tesla V100.

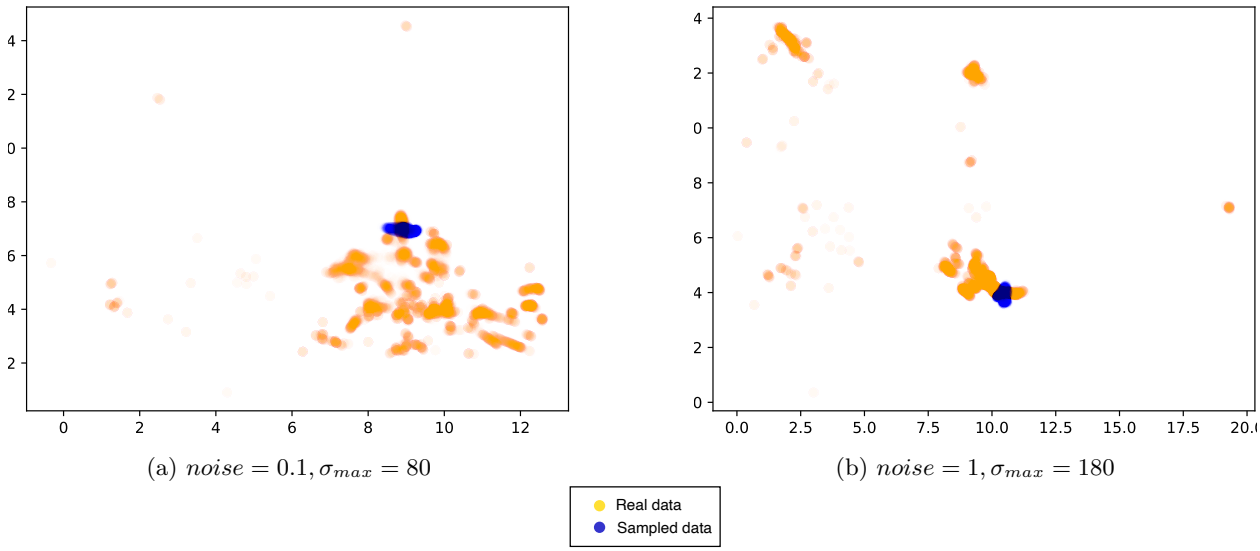
- Rand index, adjusted Rand index, mutual information-based score, homogeneity, completeness, and V measure. Rand index measures the similarity between two cluster assignments while ignoring permutations. The expected value for random split is 0.5. Adjusted rand index corrects for chance. Adjusted mutual information similarly estimates agreement between two assignments. High homogeneity indicates that each cluster contains only a member of a single ground truth class. High completeness corresponds to the scenario where all given class members are assigned to a single class. V-measure is a harmonic mean between homogeneity and completeness. As we can see from Table 1, even the highest score achieved by the K-Means with eight initial clusters is considerably low.

UMAP is an algorithm that attempts to map data points into lower-dimensionality space (most commonly, 2D). Our results from Fig.4 suggest that only a portion of actual biological variation was captured, and even adjusting diffusion parameters does not provide us with sufficient diversity.

We compare the performance of the PyTorch implementation of the Consistency Model on NVIDIA Tesla V100 and the synthetic data on the Cerebras system. Current results of samples rate (Fig.3) comparison demonstrates that adopting the Cerebras system gives us 3.3x time speed boost with 93.97 samples per second being processed on Wafer-scale cluster against 28.16 samples per second on GPU. This is a result for basic configuration; better performance gains are expected when increasing number of workers.

Table 1: Clustering scores over the combination of synthetic and real-world data

Algorithm	K-Means			Spectral Clustering			DBSCAN
Number of clusters	2	4	8	2	4	8	Auto
Rand Score	0.51	0.53	0.70	0.52	0.52	0.52	0.52
Adj Rand Score	-0.01	0.06	0.42	0.0	0.0	0.0	0.0
Adj Mutual Information	0.16	0.25	0.54	0.0	0.0	0.0	0.0
Homogeneity Score	0.13	0.31	0.93	0.0	0.0	0.0	0.0
Completeness Score	0.20	0.22	0.38	1.0	0.05	0.05	1.0
V Measure Score	0.16	0.25	0.54	0.0	0.0	0.0	0.0



(a) $noise = 0.1, \sigma_{max} = 80$

(b) $noise = 1, \sigma_{max} = 180$

Figure 4: UMAP embedding for the noise generated with two different sets of diffusion parameters.

5 Conclusion and next steps

Current work adapted a novel class of diffusion models - consistency models - to the gene expression data from cell lines. However, generating realistic biological samples remains a challenging task. We can see from results in Fig.4 that the current model is able to capture only part of biological variability and is heavily skewed towards cancer types prevalent in data. Intensive tuning may be required to figure out appropriate noise parameters for the random process.

The next step to alleviate this issue is to add class conditioning to the model so it is able to provide samples for multiple cancer types and augment data based on other important factors, such as gender and race, in order to achieve more equitable outcomes in healthcare.

Another challenge to address common to generative models in the life science field is the small dataset sizes. And in our case, it is exacerbated by the high dimensionality of the data that contains information on 17743 genes. There is a need for exploring various regularization strategies based either on expert knowledge (e.g., pathway information in our case) or automatic inference (e.g., random masking) that helps the model to efficiently factorize overly complex joint distribution of all features and generate samples that reflect these substructures in data space.

6 Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory, and is based on research supported Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory, provided by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

Efficient Algorithms for Monte Carlo Particle Transport on AI Accelerator Hardware

John Tramm[a,*], Bryce Allen[a,b], Kazutomo Yoshii[a], Andrew Siegel[a], Leighton Wilson[c]

[a] *Argonne National Laboratory, 9700 S. Cass Ave., Lemont, 60439, IL, USA*
[b] *University of Chicago, 5801 S. Ellis Ave., Chicago, 60637, IL, USA*
[c] *Cerebras Systems Inc., 1237 E. Arques Ave., Sunnyvale, 94085, CA, USA*

Abstract

The recent trend in computing toward deep learning has resulted in the development of a variety of highly innovative AI accelerator architectures. One such architecture, the Cerebras Wafer-Scale Engine 2 (WSE-2), features 40 GB of on-chip SRAM, making it an attractive platform for latency- or bandwidth-bound HPC simulation workloads. In this study we examine the feasibility of performing continuous energy Monte Carlo (MC) particle transport by porting a key kernel from the MC transport algorithm to Cerebras's CSL programming model. We then optimize the kernel and experiment with several novel algorithms for decomposing data structures across the WSE-2's 2D network grid of approximately 750,000 user-programmable distributed-memory compute cores and for flowing particles (tasks) through the WSE-2's network for processing. New algorithms for minimizing communication costs and for handling load balancing are developed and tested. The WSE-2 is found to run 130 times faster than a highly optimized CUDA version of the kernel run on an NVIDIA A100 GPU—significantly outpacing the expected performance increase given the relative number of transistors each architecture has.

1. Introduction

As performance gains become more difficult to attain with traditional CPU architectures, hardware accelerators have become increasingly commonplace in high-performance computing (HPC) systems. Much of this diversification has been driven

*Corresponding Author
Email address: jtramm@anl.gov (John Tramm)

by the need to support artificial intelligence (AI) training, resulting in the widespread adoption of graphics processing units (GPUs) and even more bespoke artificial intelligence (AI) accelerators. Some of these new architectures are highly specialized for deep learning workloads and are typically not designed with general-purpose HPC simulation in mind. However, some do have new and innovative characteristics that make them attractive for HPC simulation work. One such recent architecture, the Cerebras Wafer-Scale Engine 2 (WSE-2), is notable not only for its very large physical size but also for its 40 GB of on-chip SRAM available with 1-cycle latency. This characteristic makes it an attractive architecture for HPC simulation workloads that have traditionally been bandwidth- or latency-bound. In the present analysis we study a historically memory-bound HPC kernel from the field of Monte Carlo (MC) particle transport and its performance on the WSE-2 AI accelerator. The MC particle transport algorithm is an ideal candidate for this study because it is of great importance to both fission and fusion reactor simulation fields and because the MC algorithm has historically failed to achieve more than a few percent of theoretical peak FLOP performance due to its inherently stochastic memory access patterns [1].

To our knowledge, this is the first work on the topic of adapting a Monte Carlo particle transport algorithm for use on an AI accelerator. While several other works have adapted HPC simulations to Cerebras architectures [2, 3, 4, 5], prior work in the field has focused on algorithms whose inner loops tend to be composed mostly of dense matrix operations or regular stencil operations. The present work differs greatly in that it investigates a highly irregular stochastic algorithm that does not involve matrix operations, is subject to stochastic load imbalances, and requires complex multistage routing of particles through the accelerator's network. Given these added complexities, the present analysis is expected to be highly informative as to the potential for a wider variety of complex and irregular simulation methods to be mapped efficiently onto AI accelerators like the Cerebras WSE-2.

1.1. Monte Carlo Particle Transport and the Cross Section Lookup Kernel

Monte Carlo particle transport is a method for simulating the behavior of particles as they move through (and interact with) a medium. The MC process is notable in that it is a direct simulation method that simulates the histories of individual particles, rather than numerically integrating a partial differential equation that describes the process. This method is commonly deployed in a variety of scientific and engineering fields because it is both highly accurate and general-purpose, making a minimum of physical assumptions while still being capable of simulating a huge variety of problem types. The downside to the method is that it is both numerically costly (in that many particles must be run in order to reduce uncertainties to ac-

ceptable levels) and computationally inefficient (since the MC process is inherently stochastic, resulting in branchy control flow, random memory access patterns, and low natural vector efficiency). Given the method's great utility to several industries (in particular, the simulation of both fission and fusion reactors) and its optimization difficulty, recent years have seen significant research investment into developing new algorithms to allow for Monte Carlo to more efficiently map onto high-performance computing architectures such as GPUs.

In Monte Carlo neutron particle transport, the macroscopic neutron cross section lookup kernel typically is responsible for the majority of overall program runtime when simulating depleted fuel reactor cores. On CPUs, this single kernel has accounted for up to 85% of the overall runtime [1], with similarly high runtime percentages in GPU implementations (for instance, the PRAGMA GPU Monte Carlo code reported that 67% of runtime was spent performing cross section lookups [6]). Thus, optimization of this kernel is key to optimizing Monte Carlo particle transport in general, and consideration of the macroscopic lookup kernel in isolation is a valuable exercise that spares us the complexity of having to implement all lower-cost kernels in the MC transport algorithm (e.g., ray tracing, geometry representation, collision physics). In fact, the XSBench mini-app [1] that represents only the cross section lookup kernel has been commonly used as a stand-in for performance analysis of full-physics Monte Carlo. Thus, we limit the scope of this paper to consideration of only the macroscopic cross section lookup kernel, leaving implementation of other kernels within the Monte Carlo transport loop as tasks for future research.

The macroscopic cross section (XS) lookup kernel's function within the MC particle transport routine is to assemble statistical distribution data stored in a number of tables, which is then used to generate random samples for a particle's behavior as it moves through a simulated geometry and interacts with various materials (e.g., scattering, being absorbed, escaping the geometry, causing a fission). Thus, in order to facilitate stochastic sampling of each event the particle undergoes, cross section data must be looked up. This data has several dependencies, including the energy of the neutron, the isotopic composition of the material that it is traveling through, and the temperature of the material. Microscopic cross section data is typically stored in the form of a table of data for each nuclide, with several reaction channels of data stored for each of thousands of different energy levels per nuclide. The energy grids for each nuclide are typically unique, since they are generated to ensure that a maximum error threshold is not exceeded when interpolating values out of that table. Energy spacing within the various nuclide tables is also different, since spacing typically becomes very fine in locations that have sharp resonance features, which are located at different places for different nuclides. Thus, in order to assemble a

macroscopic cross section for a particle, a separate lookup operation must be done to locate and interpolate microscopic data from each nuclide's grid and then multiply each nuclide's microscopic cross section quantity by that nuclide's density within the material. These nuclide-wise quantities are then summed together to form the final macroscopic cross section value that can be used when sampling the particle's stochastic behavior, as

$$\Sigma_r(e) = \sum_{n}^{material} \sigma_{r,n}(e)\rho_n, \tag{1}$$

where Σ_t is the macroscopic cross section at energy e for reaction channel r, n is the nuclide within a material, σ_n is the microscopic cross section, and ρ_n is the density of that nuclide within the material. We note that the first character in the right-hand side of Equation 1 represents a summation, not the cross section value. Multiple cross section reaction channels are stored at each energy level (e.g., total, absorption, scattering, fission, nu-fission), and typically all are computed for each cross section lookup operation. One simplification we make here is that cross section data is also dependent on the temperature of the material (e.g., due to Doppler broadening), requiring another level of lookups and interpolation between energy levels, but we ignore that here for simplicity since it only adds another outer loop—temperature dependence does not affect the underlying inner loop of the computation. A simplified pseudocode of this process is given in Algorithm 1, which corresponds to a single lookup for a single particle at energy $e_{particle}$. In XSBench and in our implementation of the kernel for the WSE-2, we also add in an outer loop over many independent particles with randomized energy levels, each of which requires an energy lookup, as is typical of the way this kernel is expressed as part of the "event-based" formulation of the Monte Carlo particle transport loop [7].

The XSBench mini-app has historically served as a simple representation of this macroscopic cross section lookup kernel in the context of depleted fuel nuclear reactor simulations that typically feature hundreds of nuclides in the fuel material. While fresh UO_2 nuclear reactor fuel may carry only a handful of nuclides (oxygen-16, uranium-235, uranium-238), after a reactor is started and fission occurs, hundreds of fission products and actinides (as well as their subsequent decay chains) are produced and must be considered as part of a simulation. While Monte Carlo particle transport methods can be used to simulate a wide variety of problems beyond fission reactors (e.g., fusion reactor design, medical dosimetry, shielding), for the purposes of grounding our analysis, in this paper we will consider problem configurations typical of full-core nuclear reactor simulations featuring depleted fuel.

Algorithm 1 Simplified macroscopic cross section lookup kernel

$\vec{\Sigma} \leftarrow 0$ ▷ Macroscopic XS vector
for nuclide n in material **do**
    lower bound $\leftarrow$ binary search for $e_{particle}$ in $\vec{e}_n$
    $e_{n,low} \leftarrow \vec{e}_n$[lower bound]
    $e_{n,high} \leftarrow \vec{e}_n$[lower bound+1]
    $f \leftarrow \frac{e_{particle} - e_{n,low}}{e_{n,high} - e_{n,low}}$ ▷ Interpolation factor
    for channel r in $\vec{\Sigma}$ **do**
        $\sigma_{n,r,low} \leftarrow \vec{\sigma}_n$[lower bound][$r$]
        $\sigma_{n,r,high} \leftarrow \vec{\sigma}_n$[lower bound+1][$r$]
        $\sigma_{n,r,e_{particle}} \leftarrow \sigma_{n,r,high} - f(\sigma_{n,r,high} - \sigma_{n,r,low})$
        $\Sigma[r] \leftarrow \Sigma[r] + \rho_n \sigma_{n,r,e_{particle}}$
    end for
end for

1.2. Cerebras WSE-2 Hardware Architecture Overview

The Cerebras WSE-2 architecture differs from traditional HPC architectures in a number of ways. The first, and most notable, is the scale of the chip. While a modern GPU such as the NVIDIA A100 has a die area of 826 mm^2, the Cererbas WSE-2 is a "wafer-scale" chip that is scaled to utilize an entire silicon wafer at once, thus having a die area of 46,225 mm^2. In fact, one might argue that term "chip" does not well describe the WSE-2, since the etymology of this term stems from the process of breaking off a small chip from a large silicon wafer. In addition to its greatly increased scale, the WSE-2 differs greatly from CPU or GPU architectures in that it is custom built to handle deep learning AI computation tasks, featuring specialized instructions and hardware to handle the 16-bit matrix operations that are common to these workloads. While GPUs also feature specialized hardware for these tasks, GPUs also provision significant die space for resources devoted to graphics processing and more general-purpose workloads. The WSE-2 also does not not support 64-bit floating-point arithmetic; 32-bit floating-point (single-precision FP32) is the highest precision supported.

Another unique feature of the WSE-2 its lack of off-chip memory. The WSE-2 architecture does not utilize any dynamic random access memory (DRAM). Rather, all memory comes in the form of single-cycle latency static random access memory (SRAM), of which about 40 GB is available on the WSE-2. This singular feature makes the architecture a potentially good fit for HPC simulation kernels that are memory bandwidth or latency bound, since the theoretical bandwidth of the WSE-

2 is staggering at 20 petabytes/second (compared with the A100 GPU, which has 1.5–2.0 terabytes/second of bandwidth, depending on the specific model of A100). However, the low latency and high bandwidth come with a catch—namely, that the 40 GB of SRAM is distributed across 850,000 cores. Furthermore, the WSE-2 is not a shared-memory architecture. Each compute core (called a "processing element" (PE) on the WSE-2) has only 48 kB of local SRAM, with no ability to abstractly access any other memory spaces. Communication of data between PEs must be done manually by the application developer via a distributed-memory message-passing model. A simplified diagram of the WSE-2 architecture, shown in Figure 1, shows how the WSE-2's PEs are arranged in a 2D grid, with each neighbor interconnected with one another.

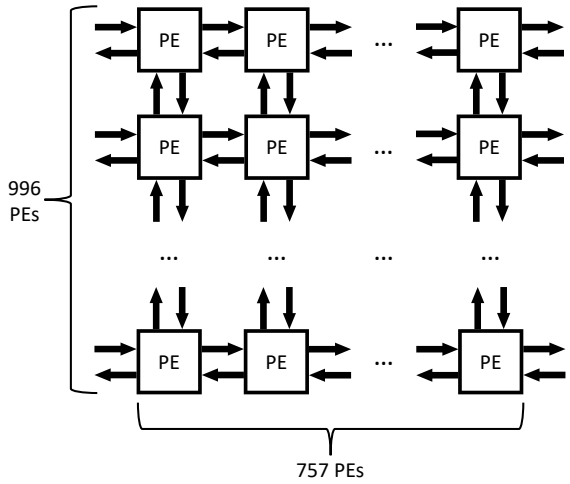


Figure 1: Diagram of WSE-2 architecture.

Another important characteristic of the architecture shown in Figure 1 is that the latency between PEs can be as low as 1 cycle. Additionally, communication between PEs is typically done asynchronously, allowing for natural pipelining of data between PEs, where the already low communication latency can theoretically be masked. Thus, while the architecture can be programmed as if it were a large 2D network of cores (much in the way HPC architectures have historically approached distributed-memory parallelism via MPI, though with major caveats), the architecture can also be approached as a dataflow architecture, where each PE is configured to perform only a single small task on an object before sending it to the next PE. We also note that unlike typical network architectures such as Ethernet or InfiniBand that are programmed with MPI, the WSE-2 network is strictly a neighbor-to-neighbor

network. Messages cannot be passed abstractly between any processor in the grid to any other processor. Rather, they must be routed manually by the programmer between neighbors. In the Cerebras Software Language (CSL) used to program the WSE-2, a few primitive operations are provided abstractly, such as reduction operations across rows or columns of the WSE-2, but in general most communication schemes must be programmed manually. Given the limited memory capacity of each PE (only 48 kB), sophisticated communication runtimes (e.g., as in MPI) are not practical. Rather, message passing is done in a very low-level manner, where the PE's router and the various limited hardware message queue resources on each PE must be explicitly configured and managed by hand.

Thus, while the architecture is highly attractive due to the 40 GB of 1-cycle latency SRAM, decomposing of data into 48 kB subdomains (and coordinating communication between subdomains using a low-level message-passing model) presents clear challenges. The adaptation of any computational HPC kernel onto the WSE-2 will therefore require that new communication algorithms and optimization techniques be developed in order to facilitate decomposition into kilobyte-scale subdomains.

1.3. Cross Section Lookup Kernel on the WSE-2

In general, our goal is to reproduce the cross section lookup kernel and parameters as defined in XSBench for the WSE-2 using the Cerebras SDK and the CSL programming model, with a few simplifications. XSBench represents a realistic lookup pattern that involves lookups from materials with few nuclides (such as the coolant) as well as fuel materials (with hundreds of nuclides) using realistic distributions of lookup frequencies taken from real Monte Carlo simulations. However, recent work in the field of GPU Monte Carlo has often found that it is optimal to partition lookups into two separate event-based kernels [8, 6, 9]. The first kernel handles only the expensive fuel lookups, while the second kernel handles the comparatively much cheaper lookups for all other materials in the simulation. Thus, for the sake of simplicity, for our implementation we consider only lookups in a single depleted fuel material.

Similar to XSBench, we also utilize randomly generated synthetic cross section data, which can be done because we care only about mimicking the computational patterns and are not concerned that the resulting macroscopic cross section data has no physical meaning. While the number of nuclides (250) in our implementation is realistic for depleted fuel, real cross section data will feature a variable number of energy gridpoints per nuclide, with some having only a thousand energy gridpoints, while others may have over 100,000 energy gridpoints. In XSBench and our implementation for the WSE-2, we simplify this variability by assuming that all nuclides

have about 10,000 gridpoints, which corresponds with the approximate average number of gridpoints per nuclide in a real depleted fuel problem [1]. We also utilize 32-bit data for both the energy grid and underlying cross section data, as is done in the 32-bit version of XSbench [10].

Table 1: Simplified cross section lookup parameters representing a depleted fuel material in a nuclear reactor simulation.

MC Cross Section Kernel Parameter	Value
Nuclides	250
Energy gridpoints per nuclide	10,000
Cross section reaction channels	5
Bytes per 32-bit value	4
Total cross section + energy data	60 MB

Given the parameters listed in Table 1, it is clear that cross section data cannot be replicated on each PE (which has only 48 kB of local memory); it must be decomposed across many PEs of the WSE-2.

2. Compute Kernel Optimization

Before we discuss cross section data decomposition methods, we begin by optimizing the basic cross section lookup computation itself, assuming temporarily that all needed data could fit within a single processing element. The naive implementation into the Cerebras CSL programming model is simple and can be done in about the same number of lines of code as if implemented in the C programming language. While additional boilerplate code in CSL is required for defining where within the WSE-2's grid the kernel will launch and for moving data between the host and device, in general the code complexity is similar to that of most other device-oriented programming models (e.g., CUDA, SYCL, HIP, OpenMP). Listing 1 gives an example of the basic kernel definition in CSL.

However, a number of potential optimizations are possible at this scale. Several of these optimizations focus on the linear interpolation operation. This interpolation operation is needed because the cross section data is stored exactly at each energy gridpoint location. In order to determine the correct cross section values between energy gridpoints, a simple linear interpolation operation is performed, as in Equation 2, where f is the computed interpolation factor, e_{low} and e_{high} are the lower and higher bounding energy grid levels, respectively, and e is the particle's current energy

```
fn calculate_xs() void {
  for (@range(i16, n_particles)) |p| {
    var e: f32 = particle_e[p];
    for (@range(i16, n_nuclides)) |n| {
      // Perform binary search
      var lower: i16 = bsearch(n, e);

      // Compute Interpolation factor
      var e_lower  : f32 = nuclide_energy[n, lower];
      var e_higher : f32 = nuclide_energy[n, lower + 1];
      var f : f32 = (e_higher - e) / (e_higher - e_lower);

      // Interpolate and store XS to particle
      for (@range(i16, n_xs)) |xs| {
        var xs_lower  : f32 = nuclide_xs[n, lower, xs];
        var xs_higher : f32 = nuclide_xs[n, lower+1, xs];
        particle_xs[p, xs] += densities[n] * (xs_higher - f *
        ↪  (xs_higher - xs_lower) );
      }
    }
  }
}
```

Listing 1: Simplified Monte Carlo cross section lookup kernel implemented in CSL.

level. With f computed, it can then be used to interpolate each of the microscopic cross section reaction channels.

$$f = \frac{e_{high} - e}{e_{high} - e_{low}} \tag{2}$$

This is an expensive operation on the WSE-2 because 32-bit floating-point division operations take around 50 cycles to complete. We consider several optimizations. The first potential optimization is to simply replace the 32-bit division operation with a 16-bit division operation. While this will result in some small loss of accuracy of the interpolation factor, given that the cross section data has associated uncertainties already, the loss of precision in this operation is not expected to impact the accuracy of a real simulation. One complication of this optimization is that the operands

and result will need to be converted between FP16 and FP32 formats, which is not free, but may still be cheaper than the savings from the reduced-precision division operation.

The second potential optimization we developed was the use of a stochastic treatment for the interpolation operation. Rather than doing a true linear interpolation, we instead sample a random energy level, s, from a uniform distribution between the two bounding energy gridpoints (e_{low} and e_{high}). We then compare this sample with the particle energy e in order to determine which of the bounding datasets to pick. If the particle's energy is above the sample, we select the higher gridpoint's cross section data to use. If the particle's energy is below the sample, we select the lower gridpoint's data. This operation is statistically identical to that of performing linear interpolation, although it does have the downside of adding in a very small amount of additional variance into the overall simulation. Typically, this is not considered to be a useful optimization, however, because, on CPU and GPU architectures, both data points are typically located on the same (or adjacent) cache lines, such that accessing both sets is not likely to result in a cache miss. Furthermore, usage of a pseudorandom number generator (PRNG) introduces additional overhead, which depending on the PRNG algorithm may itself involve a floating-point division operation. However, the WSE-2 hardware actually has specialized PRNG hardware for producing random variates quickly, making stochastic interpolation potentially much more attractive.

We note that, while statistically similar to regular linear interpolation, the use of stochastic interpolation has the potential to result in slightly higher overall variance. Because of the use of randomized (synthetic) cross section data in our CSL implementation, we cwnnot accurately gauge the impacts of this change on variance in a realistic simulation using real data. To ensure that the use of stochastic interpolation is a valid optimization, we therefore implemented the stochastic interpolation scheme into the full-physics OpenMC Monte Carlo particle transport code [11] and tested each method on a simulation benchmark problem of a realistic depleted full-core small modular reactor. To amplify the impacts of this change, we disabled use of $S(\alpha, \beta)$ calculations in the thermal neutron regions as well as probability tables in the unresolved resonance range. For a simulation with 12.5 million active batch particles in total, we found that OpenMC produced a k-eff eigenvalue of 1.00498 ± 0.00026 using normal linear interpolation and 1.00504 ± 0.00027 using stochastic interpolation. Thus, both solutions were well within statistical uncertainty, and the difference in the magnitudes of the uncertainties themselves was negligible, meaning that the stochastic interpolation strategy can be considered a valid optimization in our CSL implementation.

A final optimization we considered targets a different part of the lookup algorithm. This optimization leverages the vector units on the WSE-2 PE hardware to perform the inner loop over reaction channels in Algorithm 1. This is accomplished by utilizing intrinsic vector functions in CSL, which the compiler does not currently utilize if operations over vectors are expressed more plainly in the form of typical iterative `for` loops. We implement this optimization only for the case where stochastic interpolation is used, since this can be done easily using only a single fuse multiply-add (FMA) vector instruction (`@fmacs`, in CSL).

All three of the proposed optimizations were implemented into a basic single-PE implementation in CSL for testing on a single PE of a WSE-2 machine. Since a realistic problem size (featuring hundreds of nuclides and thousands of gridpoints per nuclide) cannot fit onto a single PE, we select a problem size that corresponds to the subdomain a single PE might possess if run in a domain-decomposed manner. Our test problem features a single nuclide, with 161 energy gridpoints, 5 cross section reaction channels, and with 100 particles. Thus, this portion of our analysis does not account for communication costs, as will be discussed later in the paper.

We collect results by running on a Cerebras CS-2 Wafer-Scale Cluster that features several CPU nodes, along with two CS-2s, each with a single WSE-2 chip. All runtime measurements in this paper are made by running on a single WSE-2 of a CS-2. Using the Cerebras SDK, each WSE-2 exposes up to 750×994 user programmable PEs (i.e., 745,500 PEs). While the WSE-2 hardware actually contains about 850,000 PEs in total, some additional rows and columns around the user-space PEs are reserved for memory movement operations (to facilitate abstractions for moving data to/from the host) and other system functions. CSL allows a programmer to define smaller grids than the maximum allowed by the hardware, or even for running on just a single PE at a time. To determine the runtime of the kernel, the CSL language exposes hardware clock cycle timer data that can be queried and saved over the runtime of a kernel and reported back to the host. Thus, the total wall time of a kernel can be computed by determining the maximum number of cycles used by any PE during the kernel and dividing it by the clock rate of the WSE-2 (850 MHz). While some degree of thermal throttling will occur, the WSE-2 implements throttling by injecting "nop" commands rather than by adjusting the clock speed itself, such that any thermal "nop" cycles are included when measuring kernel cycle counts. This method of measuring kernel runtime performance by recording clock cycles is used throughout this paper whenever runtime data is reported and is the standard method for recording performance data on Cerebras machines [2].

Performance results for our single-PE optimization strategies are given in Table 2. We found that the stochastic interpolation idea was indeed highly impactful,

resulting in about a 65% overall kernel speedup. This is an interesting result because it leverages the unique PRNG feature of the WSE-2 hardware. The WSE-2's PRNG hardware might also be expected to be useful for a more fully featured implementation of the Monte Carlo particle transport algorithm, given that it uses PRNGs in a number of places to sample from various random distributions (e.g., distance to collision, scattering direction and energy, fission spectrum sampling).

We also found that the FP16 division option did offer some speedup (about 14%), but given that the stochastic interpolation optimization was much faster, we chose to use the latter approach for the final implementation of our kernel. Thus, there was no need to validate the potential for accuracy loss stemming from this method, since it was not used in our final code. Use of vector intrinsic operations for the inner loop over reaction channels also netted a small benefit, reducing cycle counts from 281 down to 250, so this optimization (in addition to stochastic interpolation) was selected for our final implementation. With these optimizations in place, we found that the majority of kernel cycles were spent performing binary search operations, such that any further significant optimizations to the kernel would likely need to stem from lower-level optimizations of our binary search routine.

Table 2: Cross section lookup kernel optimizations for single WSE-2 PE. Cycle counts are per-particle.

	Cycle Count	Speedup Over Baseline
Baseline	463	-
FP16	405	14%
Stochastic Interpolation (Software)	399	16%
Stochastic Interpolation (Hardware)	281	65%
Stochastic Interpolation (Hardware) + Vectorization	250	85%

3. Cross Section Data Decomposition

With our single-PE kernel implementation well optimized, we now consider simulation with a realistically sized cross section dataset decomposed across many PEs of the WSE-2's grid. Since our target simulation requires at least 60 MB of data (as shown in Table 1) and since each PE has only 48 kB of local memory, decomposition to at least $\mathcal{O}(1000)$ PEs is required. Cross section data in our simplified kernel has

three dimensions (nuclide, energy, reaction channel) that are available for decomposition. With this in mind, there are two high-level strategies for decomposing data. In the first strategy, data is decomposed in a static manner, with particles flowing through the PEs as needed. The second strategy is to leave the particles in place and flow the cross section data through the network instead. Both methods have their strengths and weaknesses. In general, fixing cross section data and moving particles will be advantageous when particle object sizes are small and/or when a small number of particles per PE are being simulated. Conversely, it is advantageous to fix the particles in place and flow the cross section data through when we have many particles and/or particle object sizes are very large, such that the total size of particle objects distributed across the network becomes less than the total size of cross section data.

Given the simplicity of our present analysis and that particle objects here are small (being composed only of an energy field and five cross section reaction fields), we limit our algorithm development and communication pattern development to consider only the case where cross section data is statically decomposed and particles are moved through the network. Analysis of cross section data movement communication patterns is left for future work.

The concept of decomposition of cross section data has also been considered for improving locality in CPU-based MC simulations (e.g., via "energy banding" [12]), which functioned by limiting particles to certain energy ranges of data held by a processor at once. When particles fell below the cutoff for that energy range, they were either transmitted to another processor that had that data, or the particles were buffered until the next energy range of cross section data could be loaded. The decomposition of data across PEs of a WSE-2 is similar in nature, although it requires an even finer-grained approach given the extremely limited memory resources (48 kB) of each WSE-2 PE.

To this end, we have developed a scheme wherein each row in the WSE-2 PE grid is dedicated to an energy band, while each column in the PE grid is dedicated to a single nuclide. We do not decompose reaction channels because access to these fields is often contiguous and the binary search cost of locating the correct energy grid point is amortized over these contiguous accesses, so it is not likely to be advantageous (or necessary) to decompose into this third dimension of phase space.

This sort of decomposition is expected to be reasonably efficient for usage with a fully featured Monte Carlo particle transport application, provided that an event-based mode is used. Under these conditions, a kernel invocation would represent a single cross section lookup event for all particles that are located within a fuel material within a reactor geometry.

4. Communication Patterns

With a data decomposition scheme in place, we now must solve the problem of how particles (tasks) will traverse through the WSE-2's 2D network so that they reach the needed data given a particle's current energy level. Various assumptions might be made regarding the starting distribution of particles. In the most trivial case, we might assume that particles have already been sorted into the appropriate energy bands (perhaps either on the host or by some other kernel invocation that is launched before the cross section lookup kernel is invoked). However, the assumption of particles already being sorted into energy bands is rather weak, since the cost of sorting may be nontrivial, so performance results may not be illustrative of actual performance if the kernel was implemented inside a full MC transport application for the WSE-2. Thus, we consider the more realistic case where particles begin dispersed across the WSE-2's PEs with a random energy distribution. While particles will have some level of locality associated with their energy between kernel calls (i.e., the physics of neutron scattering dictate that a neutron can lose at most half of its energy when colliding with another nucleus), given the fine width of energy bands required to decompose the energy space particles will almost always need to move to another energy band after each scattering. Therefore, for our CSL kernel implementation, we make a more conservative assumption that particles will have a fully random energy distribution, such that they may need to travel to any of the other bands to find their data.

We also assume that particles are uniformly distributed among all the PEs of the WSE, such that each PE begins the kernel holding the same number of particles. While we assume that all PEs have the same number of starting particles, because of the randomization of energy this will result in some variance in the workload (total number of particles) that each energy band will experience. Thus, even if this assumption were relaxed and some noise was added to the starting distribution of particles among PEs, it would not be particularly impactful given that the particle energy distribution is already randomized. Given this random energy assumption, the inevitable result is that there will be some level of stochastic load imbalance among‘ PEs that will impact performance on the WSE-2 if not addressed.

These assumptions are reasonable if we assume we are in the midst of a well-developed event-based simulation, where particles have already been sampled and undergone several events and are now randomly distributed in energy space. This assumption would be violated during the first kernel invocation of each Monte Carlo batch in a real transport simulation because the fission particle distribution follows the Watt spectrum and is therefore clustered in the high-energy bands. However, as has been shown with many GPU event-based implementations [8], typically several

thousand events will be undergone by the particle within its lifetime; and when fresh particles are mixed in after some die early, we would expect that after the first few kernel calls the thousands of subsequent kernel calls will experience a generally random particle energy distribution. Thus, we believe that optimization of the kernel given these starting particle parameters would provide an accurate performance picture for the kernel if it were implemented within a full Monte Carlo particle transport loop.

With particle starting conditions in mind, we now must consider how to move particles from their starting locations through the WSE-2 PE network first to their appropriate energy bands and then later to accumulate data from all nuclides within the material.

4.1. Energy Sorting

With a random distribution of particles in energy space assumed as the starting condition for any given column in our grid, our first task is therefore to sort all the particles in the column into the correct energy band (row). Particles do not need to be perfectly sorted into their exact order; rather, the sorting operation is more akin to construction of a histogram. To accomplish this communication task, we implement a routine wherein each column of PEs will form a 1D communication pattern, with each stage of communication involving a PE transmitting particle data above and below it while also receiving particle data from above and below it. For the highest and lowest PEs in the column, particles will not be transmitted above or below them, respectively, since these rows represent the highest and lowest energy bands possible. While we assume each PE will start with the same number of particles, the randomized energy distribution means that message lengths between PEs in the column must be variably sized, because after several hops, particle buffers will naturally diminish in length as PEs claim particles from buffers into their own energy band.

To determine when communication is complete, we break this pattern into communication iterations, wherein particles are sent and received between nearest neighbors once per iteration. While the exact number of iterations needed to move all particles to their energy band rows is unknown, we can bound this value by considering the worst case where a particle starting at the bottom row needs to migrate to the top row, which for a column of height h will require $h - 1$ communication iterations. This inevitably results in unnecessary communication overhead. A more optimized strategy might be to utilize the CSL "tally kernel module," which is a library-based method for coordinating termination criteria between many PEs. For the present case, however, we use the simplified communication model that performs

$h-1$ communication iterations and otherwise does not involve any coordination between PEs to determine whether termination criteria are met. A simplified diagram depicting this process is shown in Figure 2.

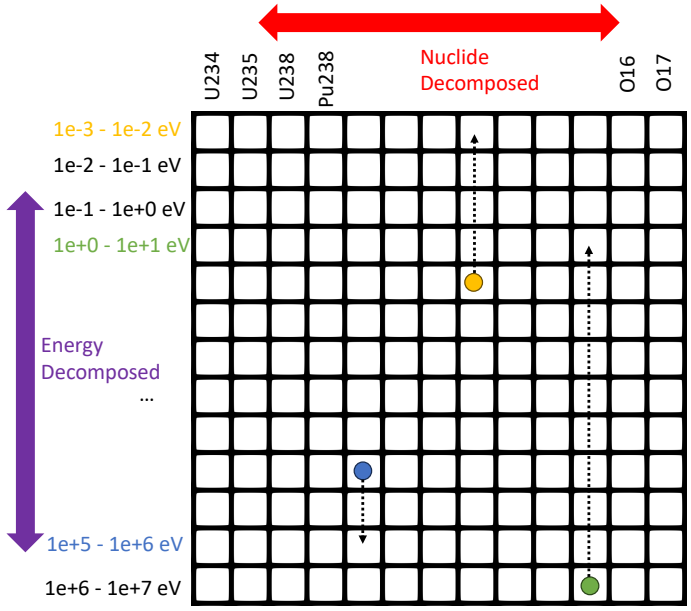


Figure 2: Diagram showing the energy column sorting process. Each square represents a single PE in the WSE-2 grid, with each dot representing a particle. Particles are color coded according to their energy band. While only three particles are shown, in reality each PE may be processing many particles at once. Particles move to their destination rows by moving one row at a time, being evaluated to determine whether they are in the correct energy band and then being either claimed or transmitted again.

We begin our column-sorting performance analysis first with a weak scaling study along just a single column within the WSE-2, wherein the problem size per PE is fixed. We study just a single column here because in this stage each column will be operating independently. While there is theoretically the opportunity for adjacent columns to affect the performance of a neighbor column because of thermal effects, we will study such effects later in this paper when performing our final full-machine runs in section 5.

Our baseline weak scaling problem size is for 1 nuclide for the column, with the nuclide having 1,000 energy gridpoints and 5 cross section reaction channels. Since our communication pattern requires that each PE have a task to do (e.g., processing of at least a single particle for a single nuclide), we pin the problem size per PE

as the number of starting particles per PE, n. The sensitivity in performance to n is also investigated. Weak scaling efficiency is then calculated separately for each study of n by comparing the number of cycles per PE per particle against the case where only 1 PE is used for that value of n. Our results for this single-column study, shown in Figure 3, show that the communication costs are significant past 10 PEs in a column for all cases and can become costly when even just two PEs are used if the particle count per PE is low. While these results may at first appear to indicate impractically high communication costs for this decomposition scheme to work, we note that the sorting costs in this study are amortized over a lookup only for a single nuclide. While adding more nuclides to the problem will not reduce the absolute communication costs of this sorting operation, it may better amortize them such that the relative cost of sorting may appear small compared with the cost of doing useful cross section lookup work (as will be studied later in the paper in section 5 when 2D decomposition results are presented).

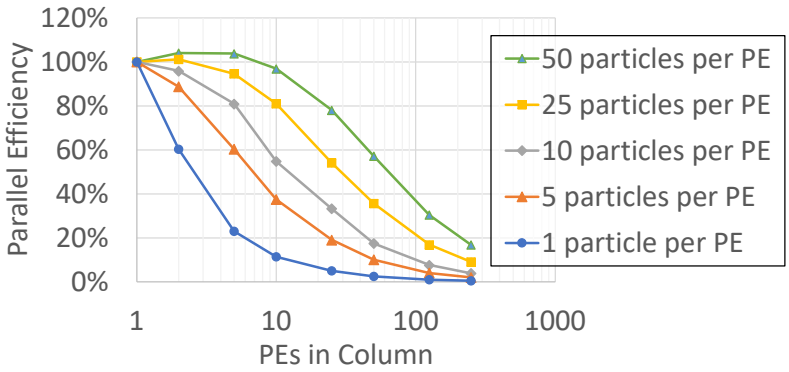


Figure 3: Weak scaling energy decomposition study across a column of PEs. The problem size involves one nuclide with 1,000 energy gridpoints per energy level (row) and five cross section reaction channels. Problem size per PE is fixed with n particles, with several studies run for various quantities of n.

We also performed a strong scaling study for the column-sorting algorithm. In strong scaling, the global problem size is fixed; and as we add PEs to the row, we spread out the static load (particles) among more processors. This type of scaling is fairly unnatural on the WSE-2, given that it is difficult to create a meaningful problem with enough work for hundreds of PEs that can also be fit within only the memory resources of a single PE. We fix the number of particles at 100, the number of nuclides at 1, the energy gridpoints per nuclide at 800, and the cross section reaction channels at 5. Note that as we add rows to the column, the number of gridpoints

stored for that nuclide per PE will also decrease due to energy decomposition, which has the positive effect of slightly reducing the number of binary search operations that must be performed as more PEs are used.

The results of our strong scaling analysis, given in Figure 4, show that after there are 10 PEs in the column, the communication costs involved with sorting begin to become too expensive for further performance gains. Notably, the sorting costs are non-zero when only a PE is present, since the PE must analyze all 100 starting particles to determine whether they are in the local energy band and copy them into a separate buffer, which has a significant startup cost associated with it. As a few additional PEs are added, the cost of this initial sorting operation is diminished due to each PE starting with fewer particles—in this regime is a bigger win than the added costs of transmitting particles between PEs. Once approximately 10 PEs are used, however, the communication costs tend to surpass the time it takes to do the actual lookups, after which point use of more PEs results in an increase in runtime. This strong scaling analysis is useful in that it makes clear the benefit to decomposing to at least 10 energy bands, even in the case of a very small problem size. It also makes clear that decomposition past 10 energy bands is not efficient, although we will investigate empirically in section 5 whether the costs of sorting end up being small enough that overall performance is not impacted.

4.2. Nuclide Accumulation

With particles sorted into their appropriate bands, we now must solve the problem of ensuring that each particle is able to access data for every nuclide within the material. As described in section 3, our strategy is to decompose nuclides over columns. Once a particle has arrived in the correct energy band (row) after sorting, the particle must accumulate a contribution to its macroscopic cross section value from each nuclide by traveling to each PE in its row. During its visit to each PE, the particle will perform a lookup and accumulation kernel for all nuclides that the PE holds. We implement this communication pattern in terms of a "round-robin" 1D neighbor exchange, where at each communication step a PE will send outgoing particles to its right neighbor and receive incoming particles from its left. Periodic boundary conditions are manually implemented such that particles being transmitted from the rightmost boundary PE will be received by the leftmost boundary PE to continue on its round-robin traversal. For a row of width w, each particle will therefore make $w - 1$ hops until it has visited every PE in the row. A simplified diagram of this process is shown in Figure 5.

We will not go into full detail on the mechanics of how the communication pattern was implemented in CSL, since this would involve a full explanation of the intricacies

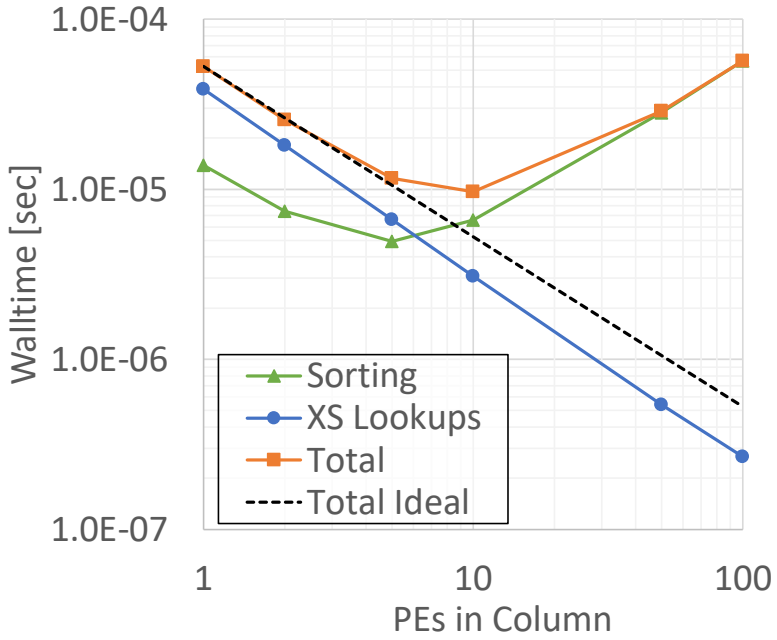


Figure 4: Strong scaling energy decomposition study across a column of PEs. The global problem size is fixed with 100 particles and 1 nuclide, with 800 energy grid points and 5 cross section reaction channels per nuclide.

of the CSL programming model, which is beyond the scope of this paper. However, we will give a cursory overview of some of the hardware characteristics and the basic concepts of message passing on the WSE-2 using CSL. Figure 6 shows the basic strategy we used, where each PE in the row had its router configured to pass messages of different colors in order to accomplish the needed one-dimensional and one-direction message-passing scheme. Each WSE-2 PE is composed of two disjoint elements: the router and the processor. The router operates independently from the processor, and the processor cannot view any messages unless they are sent down the "ramp" interconnecting the two elements first. As can be seen in the diagram, interior PEs have "even" and "odd" configurations such that adjacent PEs match colors. For example, the second PE passes to its right on color orange and receives from the left on color green, while the third PE does the opposite (sending on green and receiving on orange). This diagram also shows how the periodic boundary conditions were implemented, with interior PEs simply forwarding the periodic color (purple) from right to left and the boundary PEs having unique shapes and directions to complete the message pathway. Another important aspect of this diagram to note

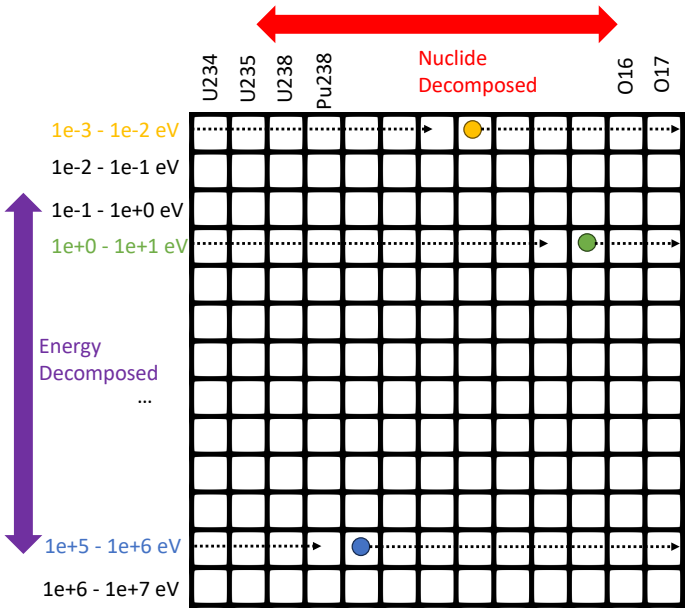


Figure 5: Diagram showing the "round-robin" row exchange process for accumulating nuclide data into each particle. Each square represents a single PE in the WSE-2 grid, with each dot representing a particle. While only three particles are shown, in reality each PE may be processing many particles at once. Particles visit each PE in the row once, accumulating nuclide cross section information for one or more nuclides at each visit.

is that messages transiting between two adjacent routers are passed with 1-cycle latency, with each message carrying a 32-bit data payload. While adjacent routers communicate at 1-cycle latency, the latency between each processor and its router within the PE is much higher, taking 7 cycles.

We also implement two different methods for performing the round-robin message passing in our code. The first method assumes perfect post-sorting load balancing. That is, after sorting in energy, every PE in the grid will contain the same number of particles. This is not a very realistic assumption given the stochastic nature of Monte Carlo, but it is nonetheless interesting as it allows us to use a simpler communication pattern. In this simplified "fixed message length" communication pattern, we can simply transmit a compile-time known number of particles between each PE at each communication phase. Doing so eliminates the need to send or receive any particle buffer length data between PEs before actual particle data is sent, potentially reducing communication costs. We also implement a more realistic communication scheme that allows for variable message sizes between neighbors, with

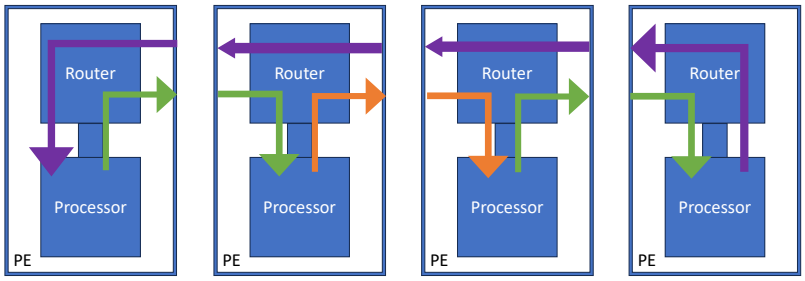
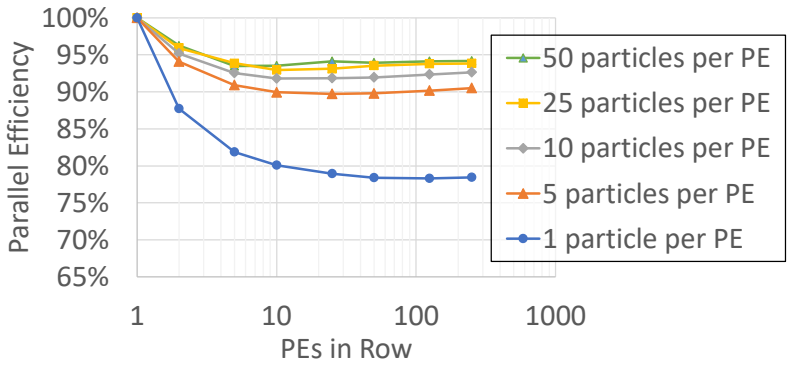


Figure 6: Diagram showing how the 1D "round-robin" row exchange process (with a periodic boundary condition) is implemented. Routers are preconfigured to apply specific directional information to messages with specific color tags. Particles visit each PE in the row once, accumulating nuclide cross section information for one or more nuclides at each visit.

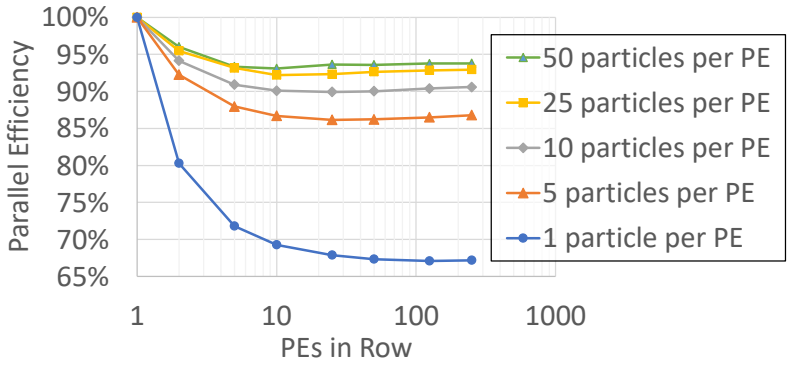
buffer lengths communicated on the fly between neighbors before actual particle data is transmitted at each communication phase. This more capable implementation allows for realistic load imbalances between PEs, but it also likely increases the latency of communication, although the cost may potentially be amortized if many particles are passed per communication phase.

We begin our analysis with a weak scaling study, wherein the problem size per PE is fixed. Our baseline problem size is for 1 nuclide per PE, with a nuclide having 1,000 energy gridpoints and 5 cross section reaction channels. Since our communication pattern requires that each PE have a task to do (e.g., processing of at least a single particle for a single nuclide), we also pin the problem size per PE in a second dimension—the number of particles per PE, n. The sensitivity in performance to n is also investigated. Weak scaling efficiency is then calculated separately for each study of n by comparing the number of cycles per PE per nuclide per particle against the case where only 1 PE is used for that value of n. Both variable- and fixed-length message-passing strategies are evaluated. Our results, shown in Figure 7, show impressive weak scaling efficiency. With higher numbers of starting particles per PE, weak scaling out to a row width of 250 PEs can remain as high as 93%. For the smallest problem sizes possible (where each PE has only a single nuclide and particle to operate on per communication round), the kernel was still reasonably efficient, at 67% efficiency for the variable-sized message case and 78% efficiency for the fixed-width case. Thus, the use of fixed-size messages was not necessary to achieve high performance.

We also performed a strong scaling row study. In strong scaling, the global problem size is fixed; and as we add PEs to the row, we spread the static load out



(a) Weak scaling with fixed-size messages determined at compile time.



(b) Weak scaling with variable-sized messages determined on the fly.

Figure 7: Weak scaling for nuclide and particle decomposition across a row of PEs. The problem size per PE is fixed to hold one nuclide with 1,000 energy gridpoints and 5 cross section reaction channels and starts n particles, with a variety of values for n compared.

among more processors. This type of scaling is fairly unnatural on the WSE-2, given that it is difficult to fit a meaningful problem for 250 PEs using only the memory resources of a single PE. Thus, we use fairly minimal parameters for our global problem size, fixing the number of particles at 250, the number of nuclides at 250, and the energy gridpoints per nuclide at 10, and with only a single cross section reaction channel. While the weak scaling analysis is likely to be more relevant in practice, a strong scaling analysis is nonetheless interesting given the added costs of communication. Similar to our weak scaling analysis, we consider both the variable- and fixed-sized message-passing schemes. The results of our strong scaling analysis, given in Figure 8, show that the kernel performs surprisingly well in the strong scaling regime, with communication costs remaining trivial until about 125 PEs of

width, after which point the communication costs begin to dominate. Given the results of the weak scaling study that indicated significant reductions in relative communication costs as the particle count was increased, we expect that a strong scaling analysis using more starting particles (which is not possible to fit into a single PE's memory for this problem) would theoretically show improved strong scaling performance, likely allowing good performance all the way to the needed 250 PE width scale.

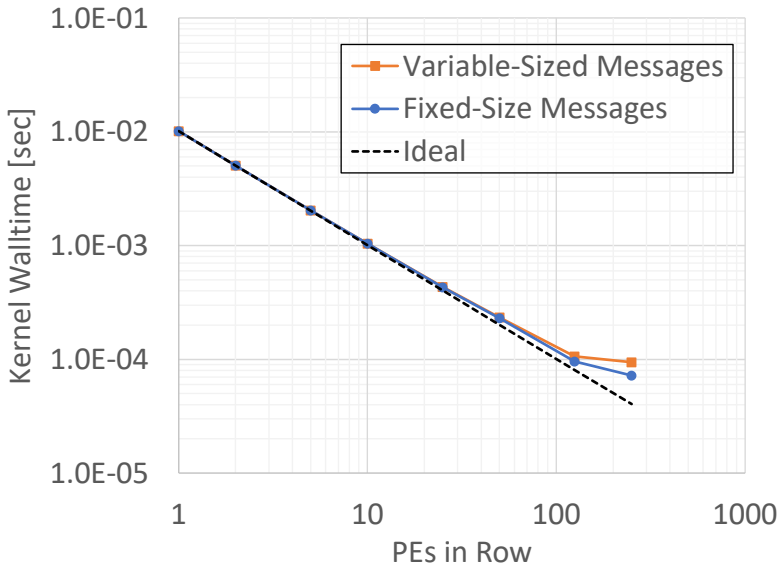


Figure 8: Strong scaling nuclide decomposition study across a row of PEs. The global problem size is fixed with 250 particles and 250 nuclides, with 10 energy grid points and one cross section reaction channel per nuclide.

4.3. Load Balancing

A potentially major implication of our decomposition scheme is the potential for large stochastic load imbalances between the PEs of the WSE-2 grid. In particular, because of the random distribution of particle energy levels, the number of particles in each energy band (and within each PE of each energy band) will be subject to fluctuations. While one can reasonably assume that the distribution will be approximately statistically uniform (assuming appropriate energy bands boundaries are selected), random variations in this distribution may greatly inhibit the overall performance of the kernel. Overall kernel wall time will be dictated by the PE that starts with the largest number of particles after energy sorting. While these particles will be passed

between each PE in the row as part of the round-robin exchange, in our current communication scheme this "workgroup" of particles will travel together and so will form a bottleneck regardless of which PE they travel to. Critically, this bottleneck is defined by the worst-case load peak out of any PE out of the approximately 750k usable PEs on the WSE-2. In other words, even for a particle distribution with fairly low noise, there are enough PEs that a high particle count outlier is inevitable. Therefore, we found that it was necessary to develop some mechanism for mitigating these load imbalance issues.

The goal for our load balancing technique is to ensure that peak load experienced by any PE is as close as possible to the average load. There are two dimensions upon which we can generally perform load balancing. The first is the number of particles that will be sampled to be within a given energy band (row). This determines the minimum load imbalance between entire rows. In a real-world MC simulation, we cannot bias this sampling directly, so we do not have full freedom to load balance between rows since moving high-energy particles into a low energy row would mean that the data they need is not available in that row. In theory, however, we could adjust the bounds of the energy band that a row holds on the fly after particle sorting, for instance by dynamically transferring nuclide gridpoint data and associated particles between rows based on the number of particles present in that row. This would be a difficult communication pattern to implement, however, particularly given that it would involve some sort of full-row communication and synchronization to determine how to restructure the row's energy band.

The second dimension we can load balance is the number of starting particles held by the various PEs within a given row (energy band) after sorting. This is a much easier dimension to load balance over, because the particles within a row do not need to maintain any special ordering, so we are free to move the particles around as is desirable for load balancing. This would not be difficult to manage if we had a single process on each row that contained a full view of the sorted particle distribution across all PEs of the row, but the expense of accumulating this sort of data across all PEs (and then transmitting a remapping pattern back out to all PEs) may be prohibitively expensive. Rather, a more ideal pattern would not involve any global synchronization between PEs in a row, instead relying only on neighbor-to-neighbor communication. A final (and more practical) characteristic of a good load balancing pattern is that it should be simple to implement, given the complexity of implementing message-passing routines in the CSL language.

With these ideas in mind, we decided to forgo any between-row load balancing given the high degree of difficultly of reshuffling both particles and cross section energy band data between rows. Instead, we propose a "diffusion-based" load bal-

ancing technique to improve the distribution with each row independently. While focusing only on load imbalances within a row will mean that attainment of fully ideal load balancing will not be possible, it can at least transfer the bottleneck from the PE with the highest load (which is extremely sensitive to outliers in the starting distribution) to the row with the highest aggregate load (which, given that there are typically going to be tens or hundreds of PEs within each row, is a value that is likely to be much less noisy).

Our proposed "diffusion-based" load balancing technique applies an iterative process to the particle distributions within each 1D row. Similar to the round-robin row exchange phase, the diffusion phase is broken up into a number of communication iterations, where at each iteration each PE will transfer half of its current particles to its neighbor(s) and receive particles from its neighbors as well. At the asymptotic limit of many communication iterations, this diffusion operator will ensure that each row has a maximally uniform distribution of particles between its PEs. A great benefit of this strategy is that even just a few diffusion iterations should have a significant impact in the overall load balance, since the diffusion process tends to flatten peaks quickly. While this diffusion process would optimally be implemented in a bidirectional manner, with PEs in a row able to send particles to both their left and right, for the sake of simplicity, we implement it in only a single direction so as to make full reuse of the round-robin communication pattern we have already implemented to handle the nuclide accumulation phase of the simulation. Thus, with only a few additional lines of code and no extra communication pattern development, we were able to add additional functionality into the round-robin row exchange routine so as to handle the particle diffusion phase as well. This makes the diffusion process a little less efficient (since PEs can diffuse particles only to the right, with periodic boundary conditions), but nonetheless we expect that this will still greatly improve load balancing.

To demonstrate this process abstractly, we present an example that takes the case of a single row of PEs with 10 columns, with a randomized starting particle distribution. Each PE begins with a random number of particles between 0 and 20, and the single-direction diffusion process is then simulated, with the results of this process shown in Figure 9. In this example, the distribution begins with a PE holding 19 particles, creating a peaking load factor of 1.9x (meaning that a subsequent cross section lookup round-robin kernel would take about 1.9x longer than ideal to complete). After just 4 diffusion iterations, the particle distribution peak load has reduced to 14 (about 1.4x), and after 8 iterations, it has improved to a peak load of 12 particles (about 1.2x).

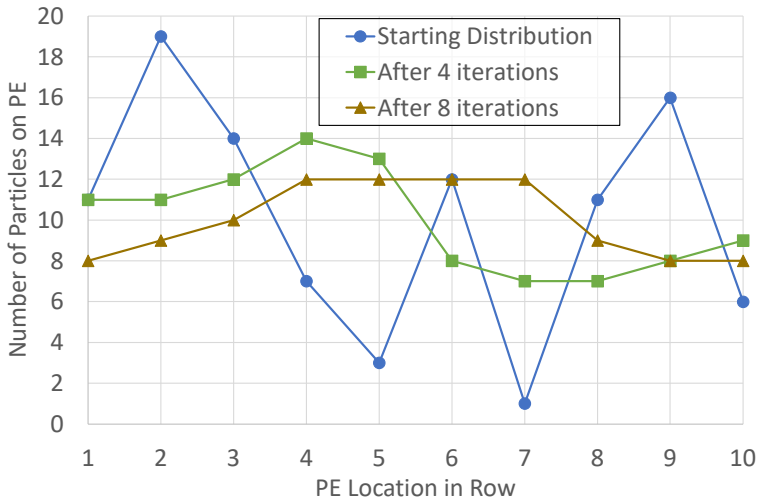


Figure 9: Simplified example of diffusion-based iterative load balancing process for a single row of 10 PEs.

4.4. Tiling

One natural limitation to our overall communication strategy is that our nuclide decomposition across PEs is naively limited by the number of physical nuclides being simulated in the problem. This would preclude utilization of the full resources of a WSE-2 (which is composed of a grid of 994×750 PEs), given that the highest number of nuclides present in a single material of a reactor simulation will be in the range of 250–300, well below the physical width of the WSE-2. Additionally, as we found in our column-scaling studies, the communication costs associated with sorting particles along a column length of 994 may be prohibitively costly. Given these considerations, it is more efficient to decompose cross section data and particles into a PE subgrid of dimensions far smaller than the overall WSE-2 grid. This smaller subgrid can then be "tiled" and replicated to run concurrently (and independently) so as to fill up the entire WSE-2. Given that particle histories are independent, there is no need for communication across tile boundaries. We also note that only 60 MB of aggregate data is needed to represent our target problem nuclide cross section dataset (while the WSE-2 has 40 GB of total memory), such that its replication across multiple tiles will not quickly exhaust the memory resources of the system. This tiling capability was added to our CSL implementation of the kernel, allowing the user to adjust the PE and tile configurations as inputs for the program.

5. Full Cerebras WSE-2 Performance

With communication schemes defined for sorting particles in energy, load balancing the particle load, moving particles between PEs for nuclide accumulation, and replicating these processes within independent tiles so as to saturate the entire WSE-2, we now have all the elements needed to execute a fully decomposed cross -ection lookup kernel at scale. For filling the full WSE-2, we consider a variety of tile sizes that can accommodate the realistic target benchmark nuclide data size listed in Table 1 so as to determine the optimal configuration. In all cases, however, the simulated target problem is configured to consistently represent a problem with 250 nuclides, 10,000 energy gridpoints per nuclide, 5 cross section reaction channels, and 30 starting particles per PE, regardless of decomposition dimensions. The number of gridpoints per row and the number of nuclides per column were adjusted to conserve the global problem size for each configuration.

For each tile configuration, we also ran three separate studies representing different assumptions regarding load balancing: ideal starting load distribution, realistic (random) starting load distribution, and realistic load balancing with our dynamic diffusion-based load balancing technique enabled. The purpose of the three studies is to quantify the performance impacts due to the load imbalance when using realistic sampling conditions as compared with the ideal case with perfect load balancing and to see how effective our load balancing technique was as compared with the ideal. In the ideal case, particles begin in a (seemingly) unsorted and random manner in each column, but we bias the sampling process such that we ensure that, after sorting, particles will end up in an ideally load balanced distribution where every PE on the WSE-2 has the exact same number of particles. This biasing is accomplished by sampling particles for each row within that row's energy band and then shuffling the columns of particles independently on the host before transferring the particle data and launching the kernel on the device. Thus, this method still accurately captures the expense of sorting but excludes the possibility of load imbalance during the round-robin phase. The second case simply removes the biased sampling so that, after sorting, particles will end up such that some PEs will have more load than others. The third case is identical to the second except that here we enable the diffusion-based load balancing scheme (as discussed in subsection 4.3) as well.

The results of our full-machine analysis are shown in Table 3. The first key finding is that the kernel tends to perform well on all of the three tested tile configurations, with the maximal and minimal ideal performance levels varying only by about 10%. The second major finding is that, without use of a dynamic load balancing strategy, allowing for particles to have a realistic random starting distribution does indeed result in a major performance loss. All three tested configurations resulted in a

Table 3: Full-machine results on a Cerebras WSE-2.

WSE-2 Grid Configuration			Ideal (Uniform) Particle Distribution		Realistic (Random) Particle Distribution		Realistic (Random) Particle Distribution with Diffusion Stage	
Tile Dimensions	Grid Dimensions	Total PEs	Performance [Lookups/s]	Peak Load	Performance [Lookups/s]	Peak Load	Performance [Lookups/s]	Peak Load
124 PE x 25 PE	8 Tiles x 30 Tiles	744,000	9.13E+09	1	5.51E+09	1.8	7.76E+09	1.2
90 PE x 125 PE	11 Tiles x 6 Tiles	742,500	9.25E+09	1	5.51E+09	1.8	8.36E+09	1.1
62 PE x 250 PE	16 Tiles x 3 Tiles	744,000	8.42E+09	1	4.93E+09	1.8	7.63E+09	1.1

large load imbalance between PEs when using realistic starting distributions, with each grid having at least one PE holding 1.8x more particles than average after the sorting operation was complete, as indicated by the "Peak Load" field of Table 3 that measures the maximum number of particles starting on any PE to the average number of particles started per PE. This is highly problematic because the overall runtime of the round-robin nuclide accumulation phase will be dictated by whichever workgroup (i.e., a group of particles starting on one PE) is the largest, since this group will bottleneck the entire row as it migrates through the round-robin row exchange. We can validate that this is indeed a practical problem, and not merely a theoretical one, since the performance differential between the ideal case and the realistic randomized particle case follows a trend similar to what the load balance would indicate, as shown in Table 3.

The third key finding was the success of our dynamic diffusion-based load balancing scheme. For all three configurations we used 100 diffusion iterations. Our diffusion-based load balancing stage added minimal overhead and greatly reduced the load peaking factors from 1.8x all the way down to 1.1–1.2x (very close to the ideal of 1.0). For the optimal 90×125 subgrid configuration, this resulted in a whole machine performance improvement of about 52% and makes the kernel much less sensitive to variance in the starting particle distribution. While additional gains may be possible by adding bidirectional diffusion or allowing for on-the-fly load balancing between rows, these changes would be much more complex to implement and would result in only marginal gains because the current performance is only about 10% slower than the ideal case with perfect load balancing.

While our results in Table 3 show that the load imbalance between PEs can be successfully mitigated, it is unclear from this data alone what the overall communication costs were for the kernel as a whole. Thus, an important question to answer is what the overall communication overhead is for our decomposition, load balancing, and particle movement scheme combined. Are significant further gains in performance possible by developing more optimal communication patterns, or is

the kernel performing close to the ideal? Given the fine-grained decomposition of data that is required to fit each subdomain within 48 kB of local PE memory, were communication costs prohibitively expensive?

We can answer these questions by computing an idealized optimal cycle count based on extrapolating the performance values gathered when developing the single-PE optimized kernel. In Table 2 we found that our optimal single PE performance (with no communication routines) required 200 cycles per nuclide per particle. Thus, to process 30 particles with 250 nuclides (assuming 161 local energy gridpoints), we can compute the total cycle count as 250 cycles per nuclide per particle $\times$ 250 nuclides $\times$ 30 particles = 1,875,000 cycles.

Our optimized full-machine WSE-2 run for the 90 $\times$ 125 PE case in Table 3 consumed 2,264,078 cycles (the maximum cycle count out of all PEs on the WSE-2). We can compare this value with the optimal single-PE (per-nuclide, per-particle) value of 1,875,000 cycles. Thus, the full cost of our communication scheme adds only 21% overhead as compared with an ideal case (assuming each PE had infinite memory and hence could replicate the full cross section dataset locally) where no communication was required. Additionally, we note that most of this cost can likely be attributed to the remaining 10% load imbalance. In this light, the opportunity for further optimizations to our decomposition and communication strategies (as well as their implementation details in CSL) appears to be narrow given the excellent performance that has already been achieved with this scheme.

6. Comparison with GPU

In order to appraise the performance of our XS lookup kernel on the Cerebras WSE-2 machine, a baseline was needed. Recent work in Monte Carlo particle transport has shown that GPUs tend to be far more efficient for this algorithm than are CPUs [8, 9, 6]. We therefore decided that the A100 GPU would make an ideal representative for what is possible for performance of this algorithm on more mainstream HPC architectures. In particular, the A100 is a good candidate for comparison with the WSE-2 given that both chips were manufactured by Taiwan Semiconductor Manufacturing Company using the same 7 nm process. To this end, we have ported the kernel into CUDA and have considered a variety of GPU- and CUDA-specific optimizations to ensure that a fair comparison is made. In other words, we take a "gloves off" approach and consider all possible optimizations for the GPU (just as we did with the WSE-2), even if the optimizations do not translate between architectures.

Before any performance comparisons are made, it is important to set reasonable expectations given the significant resource disparities between the wafer-scale WSE-2 and the more traditionally sized A100 chip. This is an important comparison to

make because, given the trouble of developing code in Cerebras's proprietary CSL programming model and the difficulty of developing performant decomposition and communication schemes, we would hope that the WSE-2 would be able to offer some performance advantage over an equivalently scaled cluster of A100 GPUs. A few of the possible ways of making a fair comparison are given in Table 4. While comparison of transistor counts, die area, and power are straightforward, comparison of peak floating-point capability is more nuanced. A single WSE-2 PE can do up to one FP32 fused multiply-add per cycle or two FP32 floating-point adds per cycle, in either case resulting in two FP32 operations per cycle. The clock rate of the Argonne CS-2 installation used in this study is 850 MHz, although the WSE-2 is capable of clocking up to 1 GHz (though this may cause an increase in thermal throttling). If excluding system and memory operation reserved PEs, such that only 745,500 PEs are used out of the 850,000 total, then we compute the theoretical maximum performance of the WSE-2 as $750 \times 994 \times 2 \times 850 \times 10^6 = 1.267 \times 10^{15}$ FP32 FLOPS. If using a more liberal interpretation of theoretical performance (i.e., using all PEs and assuming a steady 1 GHz clock), the value would increase to 1.7 PFLOPS.

Table 4: Comparison of an NVIDIA A100 (SXM4 40GB) and Cerebras WSE-2 architectures.

	Transistor Count [Trillion]	Die Area [mm^2]	Peak Power [kW]	Theoretical FP32 Peak [TFLOPS]	Monte Carlo XS Lookup FOM [Lookups/s]
A100 GPU	0.0542	826	0.4	19.5	6.43E+07
Cerebras WSE-2	2.6	46,225	22.8	1,267	8.36E+09
WSE-2/A100	48	56	57	65	130

Since the A100 GPU we used for our testing featured 40 GB of high-bandwidth memory, there is no explicit need for cross section data decomposition. This greatly simplifies the implementation, although a few fine-grained optimizations are still considered. In particular, we tested several of the same optimization strategies used to optimize the kernel for the WSE-2, as well as other techniques that make sense only in the context of GPUs.

The first optimization for the GPU we considered was use of half-precision arithmetic for the division operation in the interpolation phase of the lookup kernel. The second optimization was to consider use of stochastic interpolation instead of linear interpolation. To facilitate this, we used two different methods for pseudorandom number generation—NVIDIA's first-party cuRAND library and a minimal linear

congruential generator (LCG). We note that NVIDIA's cuRAND library is simply an optimized software library; there are no special hardware units on the A100 for random number generation. The third optimization was to sort particles before running the lookup kernel. This optimization was easy to implement via the NVIDIA CUDA Thrust sorting library. Timing results presented for this optimization include the costs of sorting, which were fairly small compared with the cost of the cross section lookup kernel itself.

An additional optimization was also implemented that is not practical on the WSE-2 given per-PE memory constraints. This optimization, known as the "double indexing" or as the "unionized energy grid (UEG)" algorithm [13], is an acceleration technique that functions by reducing the number of binary searches required for each XS lookup. The key optimization that this scheme makes is that only one binary search is required, rather than one binary search for each nuclide's energy grid, as is typically needed. For example, for a material with 250 nuclides, 250 binary searches would normally be required, but with the UEG approach only a single binary search is needed. The downside to this optimization strategy is that a significant additional quantity of memory is required to build a large table of indices. In order to implement the UEG optimization, a "unionized" grid of all energy levels of all nuclides is generated and sorted. For real cross section data, some nuclides will have some overlap between energy points; however, this overlap is typically small ($< 20\%$), so for our synthetic data we will assume that all nuclides feature unique energy grid points. This means that the total number of energy gridpoints in the unionized energy grid will be equal to the number of nuclides times the number of gridpoints per nuclide. For each gridpoint on the unionized energy grid, we store an array with an index into each nuclide's energy grid corresponding to the energy level that is at or just below the unionized energy gridpoint. At kernel runtime, a thread will perform a single search on the unionized energy grid, and it will then have a map of where the corresponding energy level can be found in each nuclide's grid. Theoretically, all cross section data can be stored on the unionized grid as well, although this results in a lot of replicated data, and given that 5 or more reaction channels are typically stored, this can result in an impractically large amount of data. Overall, for a typical problem of 250 nuclides and 10,000 gridpoints per nuclide, the addition of the UEG acceleration structures adds about 2.5 GB of memory usage.

The UEG approach is theoretically possible to implement on the WSE-2, although it greatly increases per-PE memory requirements. When fully decomposed in energy such that each row of PEs on a WSE-2 holds only 10 energy gridpoints per nuclide, for 250 nuclides this would increase the per-PE storage requirements from needing to store only 10 nuclide energy gridpoints with 5 reaction channels (about 240 bytes) up

to needing to store $10 \times 250 = 2500$ energy gridpoints and an equivalent number of indices (about 15 kB in total). While this can be reasonably fit within the 48 KB of memory per PE, it would preclude the extensive use of tiling to reduce communication costs, since at most two to three tiles could be used in the energy dimension, which would significantly increase overall communication costs.

We also considered an "energy-banding" approach on the GPU where smaller bands of cross section data are imported sequentially into shared GPU memory for faster access. However, we found that particle sorting tended to make this sort of optimization unnecessary because, when sorted, all particles within a warp tended to access the same exact global memory data. In this case, movement of global memory into local memory would not be expected to be amortized with any reuse.

The results of our optimization studies on the GPU are shown in Table 5. The primary optimization on the GPU was clearly the sorting of particles, which allows threads within a warp to access the same cross section data. Once particles were sorted, other optimizations had little impact. For instance, use of stochastic interpolation and the unionized energy grid improved sorted performance by only 1%. Overall, the relevant figure of merit for a single A100 GPU with a maximally optimized kernel implementation is 64.3 million lookups/sec. We compare this value with the measured full-machine value from the Cerebras WSE-2 in Table 4 and find that the WSE-2 achieved a rate of 8.36 billion lookups/sec with realistic load balancing. Therefore, the WSE-2 was about 130x faster than a single A100 GPU.

Table 5: Cross section lookup kernel optimizations for an A100 (40 GB SMX4) GPU/

	Lookups per sec	Speedup over Baseline
Baseline	2.94E+07	-
Sorting	6.33E+07	2.16
Sorting + FP16 Division	6.31E+07	2.15
Sorting + Stochastic Interpolation (LCG)	6.37E+07	2.17
Sorting + Stochastic Interpolation (cuRAND)	6.22E+07	2.12
Sorting + Unionized Energy Grid	6.41E+07	2.18
Sorting + Unionized Energy Grid + LCG	6.43E+07	2.19

Our analysis of GPU performance has so far allowed each architecture to be run in an architecture-specific configuration where the optimal problem size (e.g., number of particles) is used for each architecture. For instance, the full-machine WSE-2

performance results given in Table 4 correspond to a total problem size of about 22.3 million particles. The A100 GPU utilized a total problem size of 100 million particles. This is notable because the Cerebras is about 50x larger than the GPU, yet was running with high efficiency on a problem size 4.5x smaller than what was needed to saturate a single A100 GPU. This is an important point of divergence between the two architectures and an area where the WSE-2 tends to stand out. While GPUs require a massive amount of parallelism to be expressed in order to allow for full masking of latency to main memory, the WSE-2 architecture can achieve reasonable efficiency with even only a single starting particle per PE.

To more fully understand the relative differences in small problem size performance, we ask a simple question: For a problem size of 744,000 particles running on the WSE-2, how many GPUs would be needed to strong scale to in order to be able to match the WSE-2's wall time? In this case, not even an infinite number of GPUs would be able to match the WSE-2's performance when considering small problems like this. This is shown clearly by the asymptotic strong scaling limit of GPU performance, where we consider the wall time it takes for the GPU to process just a single particle in isolation. The kernel time on an A100 is about 368 μs to process the single particle (with sorting overhead excluded, since it would not be needed if just a single particle were used). Comparatively, the wall time it takes for a WSE-2 to process 744,000 particles (i.e., one starting particle per PE) is only 280 μs (237,914 cycles), including all communication costs. Thus, we highlight that while the WSE-2 is shown to be 130x faster than a single A100, it may often take far more than 130 A100 GPUs to match the performance of a single WSE-2 on all but the largest problem sizes, given the A100's loss of efficiency when strong scaling.

7. Future Work

7.1. Prospects for a Full Monte Carlo Application on the WSE-2

Our analysis has covered only a single kernel from the Monte Carlo particle transport algorithm. While it is typically the most expensive kernel in the MC algorithm (at least in the context of nuclear reactor simulation problems), significant additional research would be required in order to implement a fully featured Monte Carlo application on the WSE-2. For a full-physics MC simulation to be performed on a WSE-2, several additional kernels would need to be implemented to complete the particle transport loop, and several additions would need to be made to the cross section lookup kernel as well due to simplifications that were made in this analysis.

The major simplifications made in this analysis for the MC cross section lookup kernel were as follows:

- Use of only single-temperature cross section data. A realistic simulation will likely need to handle multiple XS datasets, each at a different temperature, with stochastic interpolation performed between the levels.
- Lack of $S(\alpha, \beta)$ thermal scattering data. A realistic simulation will need to store a small amount of additional data for certain nuclides in this low-energy range.
- Lack of probability tables in the unresolved resonance range. A realistic simulation will need to store a small amount of additional data in this high-energy range.
- Representation of only a single material. A realistic simulation will need to handle multiple material types, each with their own isotopic compositions.

We believe that these missing capabilities should be feasible to add to the kernel without significantly changing the fundamental decomposition scheme or particle exchange routines. For instance, use of multiple temperature levels may require only that slightly larger tiles be used in order to reduce the per-PE memory overhead of storing nuclide XS datasets for multiple temperature levels.

Implementation of the remaining kernels that form the basic Monte Carlo particle transport loop would undoubtedly require additional research and the development of new algorithms. However, the two other fundamental kernels that are required (ray tracing and collision physics) are not likely to require as much data as is required for cross section storage, such that data replication may be feasible. For instance, full-core nuclear reactors often feature lattice-based geometries, where only a handful of basic pin cell universes are defined, a few fuel assembly types are defined as lattices of these basic pin cells, and the reactor as a whole is defined as a lattice of fuel assemblies. In these instances, few surfaces and constructive solid geometry cells actually need to be stored in memory, such that it is feasible to define the full-reactor geometry within only a few kilobytes of data. In cases where the full geometry cannot be replicated across all PEs, it may still be feasible to domain decompose only across tiles, rather than across all PEs. In this paper's CSL implementation, we replicated everything between tiles, but in theory it would be trivial to domain decompose particles across tiles and to add a reshuffling stage to allow particles to be exchanged between tiles when necessary.

7.2. Alternative Algorithms

In addition to the algorithms described so far in this paper for decomposing cross section data across the WSE-2 grid and for moving particles through the network

for processing, we also considered several other algorithms.

We first consider an alternative to the method for performing the row exchanges for accumulating nuclide information defined in subsection 4.2. In the alternative "row-reduction" method, nuclides are decomposed across the PEs in a row (just as in our original "round-robin" method), but particles begin by being copied to all PEs. Each PE then processes all the particles at once, and then a row reduction operation is performed. The immediate advantage to this method is that it leverages the CSL row reduction abstraction, meaning that the communication pattern between PEs within a row does not need to be manually programmed or coordinated. As shown in Figure 10, however, this method resulted in an increase of 2.7–4.5x in overall runtime costs as compared with the round-robin approach, such that the added complexity of the round-robin did appear to be unavoidable. The significant difference in cost was likely due to the row-reduction method not being able to mask communication costs with useful work, since all work had to be completed up-front before communication could begin. Another downside to the row-reduction method is that particles must be copied to all PEs in the row before the algorithm can begin; and ultimately particles end up reduced on only a single PE of the row, which does not map naturally to the event-based algorithm that this kernel would likely be used with if extended to a full-physics implementation. Conversely, the round-robin approach does not require that particles be copied, and at its conclusion particles are still evenly distributed throughout the row, which allows for other kernels (e.g., ray tracing, collision physics, tallying) to be called in an event-based algorithm without having to necessarily reorganize the particles again.

Another key idea we developed for further cutting communication costs when many particles are used (or when particle objects are very large) is to flow the cross section data through the round-robin row exchange instead of the particle data. This would seem to be advantageous when the total nuclide cross section data per PE is less than the total particle buffer size, as might be the case when simulating many particles at once. However, it is left for future work to experiment with this algorithm.

Additionally, it may be advantageous to decompose only in energy space and to simultaneously decompose energy across the global 2D grid. That is, instead of decomposing by energy in one dimension and by nuclide in another dimension, each PE would hold data for all nuclides but would hold only a very narrow slice of energy (potentially, just three or four energy points per nuclide). Particles would then be tasked with sorting themselves into the correct band in two dimensions before computation begins. This algorithm is also left for future work for testing.

It also may be possible to load balance across rows dynamically in response to

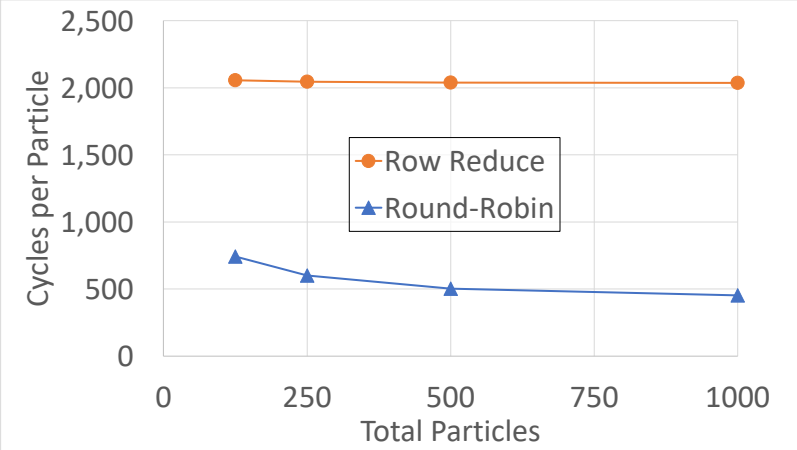


Figure 10: Comparison of the row-reduce algorithm with the round-robin algorithm. This study uses a single row of 125 PEs, with 125 nuclides, 10 gridpoints per nuclide, and 5 reaction channels in total, with the number of total particles varied. Cycle counts per particle per nuclide are reported, with both methods using traditional linear interpolation and without use of vector intrinsics.

particle energy distributions. For instance, a row having twice as many particles as its neighbor might shift a portion of its cross section data to its neighbor along with particles within that energy space in order to even the load.

8. Conclusions

In this study we ported a simplified version of the Monte Carlo cross section lookup kernel using the Cerebras SDK and the Cerebras CSL programming model and evaluated the performance of the kernel on the Cerebras WSE-2 wafer-scale machine. Beyond the challenge of porting the kernel into the low-level CSL programming model, a number of new algorithms were proposed and tested to handle the decomposition of cross section data into the small 48 kB local memory domains that each of the WSE-2's approximately 750,000 PEs contains. We also had to develop several algorithms to sort particles in energy space, load balance them, and then flow them through portions of the WSE-2 so as to accumulate all required cross section data.

Our decomposition and communication scheme involved three stages: (1) the sorting of particles in row-wise energy bands within each column of PEs, (2) an iterative diffusion-based load balancing stage for balancing starting particle loads within each row, and (3) a row-wise round-robin exchange of particles to allow par-

ticles to accumulate nuclide information from each column in the row. Importantly, all of these communication patterns had to be developed to avoid any concept of global synchronization or point-to-point message passing, given the limitations of the WSE-2 hardware. Each of the communication patterns is limited to only neighbor-to-neighbor exchanges within the 2D grid of PEs that composes the WSE-2.

In addition to these communication patterns, we developed an architecture-specific optimization that leverages the unique hardware capabilities of the WSE-2. In particular, each PE of the WSE-2 has specialized silicon dedicated to the generation of random numbers (in support of stochastic gradient descent and other common stochastic machine learning algorithms). We found we were able to leverage this to replace an expensive linear interpolation operation (which involves a very expensive floating-point division operation) with a stochastic interpolation scheme that improved overall lookup kernel performance by over 65%.

When our algorithm was run at scale on a full Cerebras WSE-2 chip, we found that our dynamic load balancing scheme was able to successfully flatten peak per-PE load factors from 1.8x down to 1.1–1.2x, resulting in a similar factor of speedup. We also found that the total communication cost of our decomposition scheme was only about 21%, which is surprisingly low given the fine-grained nature of our data decomposition across the nearly 750,000 PEs of the WSE-2.

To provide a baseline to contextualize the performance of the WSE-2, we also developed a highly optimized CUDA kernel for testing on an A100 GPU. We implemented stochastic interpolation optimizations on the GPU; but because of the A100's lack of dedicated random number generation hardware, these strategies were not found to be helpful there. However, other GPU-specific optimizations (like particle sorting via CUDA thrust, implementing a memory-expensive unionized energy grid, and investigating the use of shared memory) were implemented to ensure that the GPU kernel was maximally optimized given the specific capabilities of the NVIDIA architecture.

Overall, we found that a single Cerebras WSE-2 wafer-scale chip was about 130 times faster than a single A100 GPU, when both systems were run using system-optimal problem sizes (particle counts). This result is significantly more than expected given the 50–65x relative difference of silicon provisions and power metrics of the systems.

Was this exercise worthwhile then? Clearly, the WSE-2 provides a significant speedup over the A100, even exceeding expectations given the provisions of the two systems. Arguably, the increase in performance did come at a cost—namely, vast increases in both software programming and algorithmic complexity. However, we also note that similar statements could be said about GPU general-purpose programming

when it was in its infancy. In light of how AI accelerators such as the WSE-2 were designed almost exclusively around deep learning AI tasks, it is noteworthy that the WSE-2 is already able to exceed performance expectations relative to GPUs—an architecture that has had two decades to mature and that is now quite friendly to HPC simulation applications. One can imagine that relatively small hardware design changes might be made to future Cerebras architectures (and other AI-centric accelerators) that may further improve the performance of simulation applications on them and that new programming models might be developed to target these architectures in a higher-level and more portable manner. In this light, we believe that algorithmic design and optimization for these architectures will be an important topic in the field of HPC simulation going forward.

References

[1] J. R. Tramm, A. R. Siegel, T. Islam, M. Schulz, XSBench – the development and verification of a performance abstraction for Monte Carlo reactor analysis, in: PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future, Kyoto, 2014.
URL https://www.mcs.anl.gov/papers/P5064-0114.pdf

[2] M. Jacquelin, M. Araya-Polo, J. Meng, Massively scalable stencil algorithm, https://doi.org/10.48550/arXiv.2204.03775 (2022). arXiv:2204.03775.

[3] M. Woo, T. Jordan, R. Schreiber, I. Sharapov, S. Muhammad, A. Koneru, M. James, D. V. Essendelft, Disruptive changes in field equation modeling: A simple interface for wafer scale engines, https://doi.org/10.48550/arXiv.2209.13768 (2022). arXiv:2209.13768.

[4] R. Sai, M. Jacquelin, F. P. Hamon, M. Araya-Polo, R. R. Settgast, Massively distributed finite-volume flux computation, https://doi.org/10.48550/arXiv.2304.11274 (2023). arXiv:2304.11274.

[5] H. Ltaief, Y. Hong, L. Wilson, M. Jacquelin, M. Ravasi, D. E. Keyes, Scaling the "memory wall" for multi-dimensional seismic processing with algebraic compression on Cerebras CS-2 systems, in: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'23), 2023, http://hdl.handle.net/10754/694388.

[6] N. Choi, K. M. Kim, H. G. Joo, Optimization of neutron tracking algorithms for GPU-based continuous energy Monte Carlo calculation, Annals of Nuclear

Energy 162 (2021). doi:https://doi.org/10.1016/j.anucene.2021.108508.
URL https://www.sciencedirect.com/science/article/pii/S0306454921003844

[7] F. B. Brown, W. R. Martin, Monte Carlo methods for radiation transport analysis on vector computers, Prog. Nucl. Energy 14 (3) (1984) 269–299. doi:10.1016/0149-1970(84)90024-6.

[8] J. R. Tramm, P. K. Romano, J. Doerfert, A. L. Lund, P. C. Shriwise, A. R. Siegel, G. Ridley, A. Pastrello, Toward portable GPU acceleration of the OpenMC Monte Carlo particle transport code, in: PHYSOR 2022 - International Conference on Physics of Reactors, 2022.
URL https://www.researchgate.net/publication/360792320_Toward_Portable_GPU_Acceleration_of_the_OpenMC_Monte_Carlo_Particle_Transport_Code

[9] S. P. Hamilton, T. M. Evans, Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code, Ann. Nucl. Energy 128 (2019) 236–247. doi:10.1016/j.anucene.2019.01.012.

[10] J. Tramm, K. Yoshii, P. Romano, Power at your fingertips: Assessing the performance of a Monte Carlo neutron transport mini-app on consumer laptop GPUs, in: International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering, Niagara Falls, Ontario, Canada, 2023.

[11] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, K. Smith, OpenMC: A state-of-the-art Monte Carlo code for research and development, in: Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo, Paris, France, 2013.

[12] A. Siegel, K. Smith, K. Felker, P. Romano, B. Forget, P. Beckman, Improved cache performance in Monte Carlo transport calculations using energy banding, Computer Physics Communications 185 (4) (2014) 1195–1199. doi:https://doi.org/10.1016/j.cpc.2013.10.008.
URL https://www.sciencedirect.com/science/article/pii/S0010465513003366

[13] J. Leppänen, Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation, Annals of Nuclear Energy 36 (2009) 878–885. doi:10.1016/j.anucene.2009.03.019.

Exploring long context transformer models for Genomics

Azton Wells, Kyle Hippe, Arvind Ramanathan

October 2023

1 Introduction to the Science problem

Transformer architectures [1] have become prolific in natural language processing (NLP) and have considerable ability in many domains of sequential data. Despite this natural fit to sequential, language-like data, transformers have found limited use in realms such as genomics. By construction, transformers have an input window, or context, across which they have"perfect" memory–meaning the weights of the transformer model can consider information at any point in this context window. Outside this window, however, the transformer has no memory, although some works [4, 9] attempt to address this in various ways. In the NLP domain, a context may cover sentences, paragraphs, short stories, or even books. Even in the case of books, the model only has to consider $\sim 10^5$ words. In contrast, even small DNA sequences have $> 10^6$ bases. Current state-of-the-art models have contexts of $< 32,000$ tokens, rendering them unable to even consider 1% of a typical eukaryotic genome in a single context window.

Genomic models, if they are to account for known interactions in genomes, must be able to have specific coding and non-coding regions accessible in the same context. For example, in eukaryotic genomes, genes may be spread along a string of intron and exon segments spanning $> 10^4$ bases, where exons are removed from the string before translation. In addition, regulation of the expression of that gene may be influenced by markers $> 10^5$ bases away. Although logically necessary for genomics, such large contexts are not useful in many other domains, and so remain a largely unstudied facet of transformer architecture. Prior efforts to use transformer architectures in genomics have had very short contexts < 1024 bases, or have used approximate attention on toy datasets [8].

We have developed GenomeLM as a platform to study this problem on other architectures. It includes decoder and encoder models with ALiBi positional embeddings and memory-efficient flash attention in native Pytorch, avoiding unnecessary external dependencies where possible. Using GenomeLM along with Pytorch fully sharded data parallel, we have scaled training models across all 2K GPUs on the Polaris supercomputer. However, as model size increases, the context window that fits in GPU memory gets progressively smaller. A small 30M parameter model can successfully train with up to $\sim$ 20K input tokens, whereas a 3.3B parameter model can only achieve $\sim$ 8K token inputs, and a 33B parameter model has poor throughput even at 4K input tokens. Since there is a wealth of genomic data available, we presume that large models will be beneficial, similar to the scaling relationships that exist for language data in transformer models. If so, the field will require a method to have large models ($\sim$ 30B parameters) with very large contexts ($>$ 100K).

In this work, we attempt to expand the context window and model size of dense-attention transformer models trained on genomic data in order to study the effects of such architectural changes.

2 Description of the AI model and implementation

Prior works [2, 5, 7] have found that encoder-only style models are effective at producing embeddings for DNA models, while others [6] have used decoder-only models to great effect. As an initial study, we decide to focus on encoder-only models, with the option of expanding the study to decoder models in the future. Since many more prior works have used encoder-only models, training similar architecture will enable a more direct comparison where possible. We use an encoder-only architecture with the masking and tokenization scheme of DNABert [2] with ALiBi positional embeddings [3] to enable effective extrapolation beyond the training context window. In the event that pretraining a large context window is unfeasible, extrapolation

Parameters	n_l	n_h	n_{hidden}	n_{ff}
87M	12	12	768	3072
3.3B	40	32	2912	8400
33B	72	64	7424	16000

Table 1: Model architecture definitions used in this work. The 87M parameter model is the default size used throughout this work, unless otherwise stated.

beyond the trained window could become important for downstream tasks or for expanding the context via fine-tuning.

For consistent testing, we have adopted fixed architecture dimensions for this project, shown in Table 1. The reference architecture has 87M parameters, and is the default test case throughout this work. As is usual in language modeling, we use the cross-entropy loss as our objective. We have three datasets: a human reference genome (HG-38), a multispecies database that samples 30000 species (NCBI-MS), and a bacterial genome dataset (BRC). The datasets have 3B (HG-38), 200B (NCBI-MS), and 67B (BRC) nucleotides. Each dataset is tokenized with a sliding window of 3 nucleotides: the sequence "actgg" is tokenized as the fragments "act ctg tgg", with a total vocabulary of 71 (64 3-mer blocks with 7 special tokens). Initial training presented here uses the HG-38 dataset, although changing datasets should have no impact on training throughput.

3 What was needed to get the model running on the AI Accelerator

Initially, we were hopeful that we could compile and use our own encoder implementation from GenomeLM in order to make direct comparisons to runs performed on Polaris and other platforms. However, after significant effort to get the GenomeLM implementation to compile, we were advised by the Cerebras team that it would be much more expedient to use their ModelZoo implementation of Bert. In order to get the ModelZoo to function with our data, we implemented a new dataset designed for genomic data. This is similar to the GenomeLM dataset, with slight modification to have outputs expected by the Cerebras stack. While ALiBi positional embeddings exist in the ModelZoo, they were not functionally incorporated into ModelZoo Bert models. Enabling ALiBi required numerous modifications to the ModelZoo embedding and model implementations, and several iterations of debugging. The debugging cycle could be drastically improved with better compilation error messages to better guide the user.

4 Performance Evaluation

After adapting the ModelZoo to our application, we parameterized performance throughput in tokens per second, as shown in Table 2. A single CS-2 offers decent baseline performance on par with 41.4 A100 GPUs on Polaris. As with GPU accelerators, there is significant degradation as the context window of the model is increased. Increasing the context from 4096 to 8192 tokens decreases throughput by $5\times$ and offers performance equivalent to only 3 A100 GPUs. As well, with regards to the reference model, the Cerebras stack is also unable to scale to larger contexts than the GPU implementation with pytorch flash attention. The weight-streaming method employed by Cerebras does enable larger contexts for larger models: the final tests in Table 2 use a significantly larger encoder model, instead of the reference 87M parameters. These runs show that increasing model size has a decreased effect on throughput compared to increasing context length. As model size is increased from 3.3B to 33B parameters, the throughput drops by $< 10\times$, whereas increasing context from 4096 to 16384 decreases throughput by $73\times$. It is worth mentioning that scaling model size on the CS-2 system is trivial in comparison to using GPU systems–the memory size of the CS-2 and efficient weight-streaming algorithms mean that we did not have to work around memory limitations for model loading or checkpointing.

While we were able to ascertain performance throughput metrics, we were unable to assess the time to solution or evaluate models trained on the Cerebras system. Figure 1 shows loss evolution during training

Platform	Context	Token/s	N_{acc}	Token/GPU/s	T_{est}
Polaris	4096	524000	128	4093.75	13.56 K GPU-hr
Polaris	10256	35235	80	440.43	126.2 K GPU-hr
ThetaGPU	8192	5418	8	677.25	81.94 K GPU-hr
Cerebras	4096	169820	1	1326	325 hr
Cerebras	8192	32931	1	257.27	1687 hr
Cerebras	16384	2358	1	18.42	23.56 K hr
3.3B–Cerebras	10256	25024	1	195.51	2220.1 hr
33B–Cerebras	10256	3179	1	24.84	17.48K hr

Table 2: Comparing throughput of different platforms. We use a Bert-like encoder architecture with ALiBi positional embeddings for all tests. We also note the number of accelerators used (N_{acc}) and show normalized throughput per accelerator. For reference, we assume that one CS-2 wafer is equivalent to 128 A100 GPUs in the Token/GPU/s metric. T_{est} denotes the time required to iterate our 200B token dataset with the given throughput of a single accelerator (one A100 for Polaris/ThetaGPU, one CS-2 wafer for Cerebras). Examples that use a non-standard model size (denoted by parameter count under platform) explore how model scale affects throughput on CS-2 systems.

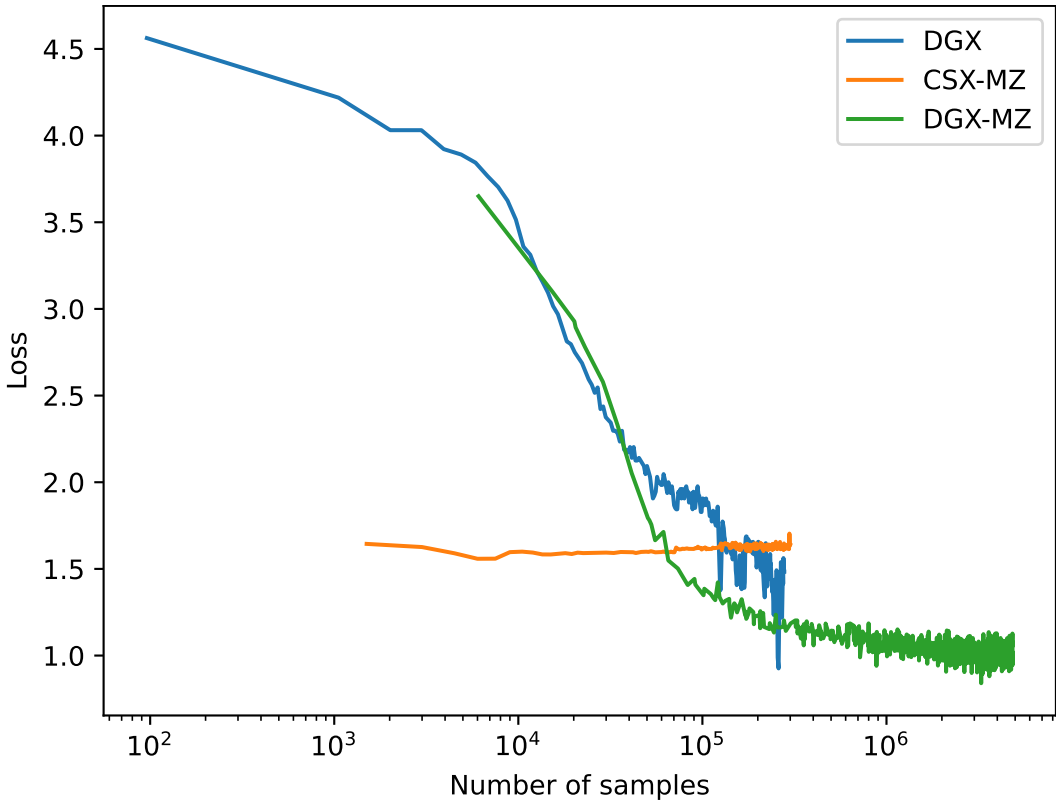


Figure 1: Training loss as function of tokens iterated. DGX denotes an 8-GPU run on a full V100 DGX node using the GenomeLM application, CSX-MZ shows training using ModelZoo on a single CS-2 wafer, and DGX-MZ shows results using ModelZoo on a single V100. Notably, the CSX-MZ and DGX-MZ implementations are *not* converged, and CSX-MZ shows evidence of approaching a solution at all.

for three examples: GenomeLM trained on a Nvidia DGX node (8 V100 GPUs) (DGX), ModelZoo encoder model trained on 1 CS-2 wafer (CSX-MZ), and the ModelZoo encoder model trained on one V100 in a DGX node. The implementations of ModelZoo are identical; only the command to run the job changes between platforms. Training fails to converge (or even make progress) on the CS-2 system despite having iterated 300K samples. The cause of this behavior is currently unknown, but is being investigated by the Cerebras team.

5 Conclusion and next steps

Although the CS-2 system offers a larger context for transformer architectures than generic GPU applications, the technology is immature from a user perspective, as it was difficult to incorporate our own workflows and to debug issues that arise during compilation. From a throughput perspective, the CS-2 system shines with larger models using $\sim$ 10K token contexts. For genomics, where we require significantly larger contexts, the system would require further development to handle $>$ 100K tokens per sample. While we plan to continue working with the Cerebras team to improve our models and their systems, our current primary focus will be to explore more exotic parallelism schemes that operate on current GPU architecture. Notably, Megatron with Deepspeed[1] has shown the ability to process $\sim$ 500K token contexts in decoder-only models, but requires further testing to evaluate training convergence and whether such a context is possible in encoder-only models.

6 Acknowledgements

We thank the Cerebras team for their quick involvement and advice during all stages of this project. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory, and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

References

1. Vaswani, A. *et al. Attention is All you Need* in *Advances in Neural Information Processing Systems* (eds Guyon, I. *et al.*) **30** (Curran Associates, Inc., 2017). https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

2. Ji, Y., Zhou, Z., Liu, H. & Davuluri, R. V. DNABERT: pre-trained Bidirectional Encoder Representations from Transformers model for DNA-language in genome. *Bioinformatics* **37,** 2112–2120. ISSN: 1367-4803. eprint: https://academic.oup.com/bioinformatics/article-pdf/37/15/2112/50927437/btab083.pdf. https://doi.org/10.1093/bioinformatics/btab083 (Feb. 2021).

3. Press, O., Smith, N. A. & Lewis, M. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. *arXiv e-prints,* arXiv:2108.12409. arXiv: 2108.12409 [cs.CL] (Aug. 2021).

4. Bulatov, A., Kuratov, Y. & Burtsev, M. *Recurrent Memory Transformer* in *Advances in Neural Information Processing Systems* (eds Oh, A. H., Agarwal, A., Belgrave, D. & Cho, K.) (2022). https://openreview.net/forum?id=Uynr3iPhksa.

5. Lin, Z. *et al.* Language models of protein sequences at the scale of evolution enable accurate structure prediction. *bioRxiv* (2022).

6. Zvyagin, M. T. *et al.* GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics. *bioRxiv* (2022).

7. Dalla-Torre, H. *et al.* The Nucleotide Transformer: Building and Evaluating Robust Foundation Models for Human Genomics. *bioRxiv,* 2023–01 (2023).

[1] https://github.com/microsoft/Megatron-DeepSpeed

8. Nguyen, E. *et al.* HyenaDNA: Long-Range Genomic Sequence Modeling at Single Nucleotide Resolution. arXiv: 2306.15794 [cs.LG] (2023).

9. Yu, L. *et al.* MEGABYTE: Predicting Million-byte Sequences with Multiscale Transformers. arXiv: 2305.07185 [cs.LG] (2023).

LLVM's Frontend and Runtime modifications to support OpenMP in the GraphCore architecture

Jose M Monsalve Diaz, Rodrigo Ceccato de Freitas, Esteban Rangel, Siddhisanket Raskar
Argonne National Laboratory, Lemont IL

October 17, 2023

Abstract

As new architectures and accelerators are introduced, the number of software libraries, tools, and systems that users must learn to use increases exponentially. System designers provide support for commonly used AI/ML frameworks (e.g. Tensorflow and Pytorch) that force applications to rely on Python and limit the use cases of these systems to applications that can be written in these frameworks. In the meantime, low-level APIs and languages are also available in these new systems, but they have a steep learning curve that makes it difficult for applications to use these systems. Furthermore, the tools and compilers used are often proprietary, limiting the ability to extend and improve these. On the other hand, scientific computation is often found in traditional programming languages like C/C++ and Fortran, and parallelization of scientific programs is achieved via OpenMP. While it is unrealistic to imagine full support for OpenMP in AI/ML accelerators, it is possible to use the offloading abstraction, work-sharing, and data management abstractions in OpenMP to interact with low-level interfaces of these systems. This work extends from a previous endeavor that designs a possible mapping between the OpenMP syntax and GraphCore's IPU operational semantics. This work describes the necessary changes to the LLVM's front end and the OpenMP runtime system to support mapping the previous design interface into the GraphCore architecture.

1 Introduction

This document summarizes the modifications to LLVM's C/C++ front-end, clang, in order to lower OpenMP syntax into GraphCore's IPU architecture [2]. This work extends from the work performed during 2022's LDRD Expedition that mapped OpenMP syntax and the IPU's execution model. We provide additional details on the implementation and lay the ground for a complete end-to-end solution. We also describe some challenges faced in the process that would require additional time and resources to complete. This document is divided into four parts. First, the introduction will provide a summary of GraphCore's IPU architecture and its software infrastructure. Second, the mapping between OpenMP and the execution model of the IPU architecture is summarized from [2]. Next, we present the implementation changes. Finally, we provide conclusions and future work.

1.1 Graphcore IPU

The Graphcore IPU-M2000 system [9] is designed to accelerate and support scale-up and scale-out machine learning applications. The IPU-M2000 system is powered by four GC200 IPU (Intelligence Processing Unit) processors and delivers 1 PFLOPS of FP16 performance, with 3.6 GB on-chip memory and up to 256 GB of Streaming Memory. Each GC200 IPU has 1472 processor cores, running 8832 independent parallel program threads with 250 TFLOPS of FP16 performance. Each GC200 IPU holds 918 MB on-chip memory with a bandwidth of 47.5 TB/s. A great description of the hardware and its capabilities can be found in [7].

This work uses the IPU-M2000 system but applies to the previous generation Colossus IPU and the latest BOW IPU [3].

1.2 Graphcore Poplar SDK

Poplar [6] is a low-level graph-programming framework for the Graphcore Intelligence Processing Unit (IPU). Poplar is the equivalent of Cuda for Nvidia GPUs. Figure 1 shows a diagram of the different components that are part of a program written in the Poplar SDK. The user must write at least two files. First, Codelets (i.e., GraphCore's names for tasks that run in the IPU tiles) are defined in a C++ interface that extends from the `poplar::Vertex` class. Each codelet defines inputs and outputs and a compute function that describes the vertex's behavior. The *popc* compiler is designed to produce a binary file (i.e., `.gp` file) that is a bundle that contains implementation for codelets in different IPU architectures and a CPU implementation for running in the IPU simulator.

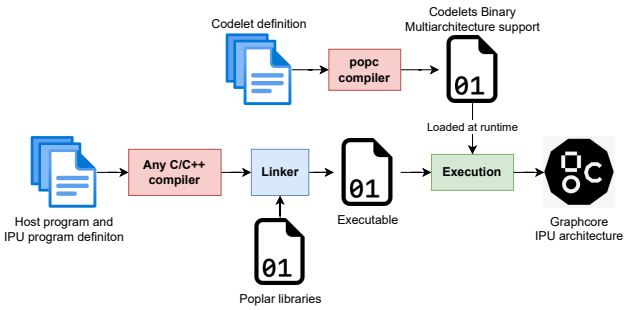


Figure 1: Compilation process of a Graphcore program written in the low-level poplar SDK. Two source codes are needed. The Codelet program describes Vertex behavior, and the host program that defines the complete IPU program that uses the codelets.

In addition to the Codelets, the user must write the host program, a conventional C++ program that can be built with any compiler. This program manages all the necessary I/O but also more importantly, it describes the IPU program. An IPU program is constructed through a series of calls to the Poplar SDK API, particularly to the `poplar::Program` class and its derivates. A complete description of the poplar API can be found in [5]. The IPU program defines the application's control flow. For example, `poplar::Sequence` is a sequence of instructions, `poplar::If` allows to create an if statement that evaluates a condition over a `poplar::Tensor` that is scalar, and `poplar::Repeat` that creates a loop for a certain number of iterations. Program subclasses also include a collection of useful procedures for debugging (e.g., `poplar::PrintTensor` that dumps the content of a tensor to the command line), moving data (e.g., `poplar::copy` that copies a tensor, or part of it, to another tensor), or other operations. The `poplar::Execute` class is also an important program component. This class receives a `poplar::ComputeSet`, a collection of codelets (vertex) executed independently. Codelets are created as vertex assigning `poplar::Tensor` to the inputs and outputs, then mapped into an IPU tile and added to a Compute Set. The Compute Set is then used as an argument of the `poplar::Execute` program stage that is then added to the rest of the program.

1.3 Graph Description in OpenMP Tasks

One of the major differences between the OpenMP tasking programming model and that of GraphCore is the need for a static program/graph description. OpenMP tasks declare dependencies through the `depend` clause. However, OpenMP relies on dynamic dependencies that are discovered based on the order of execution during runtime. Furthermore, compiler analysis is not always possible because the result of a previous task or other runtime values may determine dependencies. Some prior work has used delayed scheduling of OpenMP tasks to bridge the gap between these two models [10]. This work uses a different approach. We will describe this process in the following section.

2 Mapping OpenMP to GraphCore IPUs

Our approach takes advantage of already existing functionality in OpenMP device offloading. A `target` region in OpenMP describes a piece of the program that will be mapped into another architecture in the form of an accelerator. We use target regions to delimit code that is to be mapped into the GraphCore architecture. We describe the OpenMP syntax and the corresponding equivalent in the GraphCore IPU execution model. We also mention necessary extensions to the OpenMP syntax to exploit the architecture's capability fully.

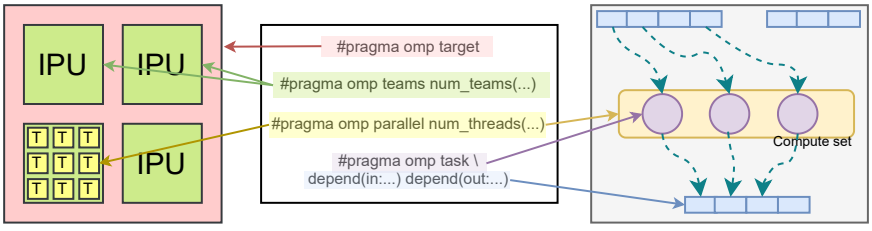


Figure 2: Syntax mapping between Graphcore architecture and OpenMP

2.1 IPU Offloading

The OpenMP `target` construct indicates the GraphCore program. All code inside the target region is effectively translated to a Poplar `program` as described before. Inside the Poplar architecture, variables must be expressed as tensors. The data clauses are used with primitive types but are then translated into corresponding tensors. However, to support this, all array sections must be discoverable at compile

time. Listing 1 illustrates the OpenMP semantics to describe a Poplar program. In the example, we use the OpenMP `map` clause with the `alloc` map-type to create Poplar tensors for the mapped regions. These tensors are the inputs and outputs for the codelets described in subsection 2.2. The OpenMP `teams` construct creates a league of threads, with each thread starting its own team of threads, i.e., contention group. We see this naturally mapping to the graph replication functionality of the Poplar SDK, where the same program is running in a data-parallel fashion on (potentially) several sets of IPUs. Lastly, Poplar intrinsic functions, e.g., `copy`, can be inferred by the dependencies between parallel regions. Intrinsic functions can also be provided to support those functionalities that are not obvious or provide further control (e.g., copy and print tensor). Finally, the Control flow of the target region must be transformed into the control flow described in the Poplar program.

Listing 1: OpenMP description of Poplar program

```
int v1[4]; // Static array sizes or mappings
float v2[4]; // Static array sizes or mappings
// Create tensors for mapped regions
#pragma omp target map(alloc: v1[:], v2[:])
  // Translated to graph replication
  #pragma omp teams num_teams(4)
  {
    const float c1[4] = {1.0,2.0,3.0,4.0};
    // This is an intrinsic funciton
    // that is part of the program.
    // Translated to prog.add(Copy(c1, v1));
    copy(c1,v1);
    // This is translated to an intrinsic
    // function that is part of the program.
    if (v1[0] > 3) { ...}
  }
```

Listing 2: OpenMP description of Poplar compute set and codelets

```
// Create compute set of 6 codelets
#pragma omp parallel num_threads(4) nowait
{ #pragma omp task depend(in:\
    v1[omp_get_thread_num():4-omp_get_thread_num()]) \
    depend(out: v2[omp_get_thread_num()])
  {
    // Body translated to codelet compute function.
    // Depend "in" and "out" do tensor mapping to the
    // inputs and outputs of the Codelets.
    *v2 = 0;
    for (const auto &v : v1)
      *v2 *= v;
  }
}
#pragma omp parallel num_threads(2)
  #pragma omp task depend(in:\
    v1[omp_get_thread_num():4-omp_get_thread_num()]) \
    depend(out: v2[omp_get_thread_num()])
  {
    *v2 = 0;
    for (const auto &v : v1)
      *v2 += v;
  }
}
```

2.2 Expressing Parallelism

Listing 2 illustrates using OpenMP semantics to create Poplar's Compute Set. We then use the OpenMP `parallel` construct and use the `num_threads` clause to determine the number of vertex to create in the compute set. Different parallel regions can be merged through the `nowait` clause, an extension to the original syntax. When used, the compiler will maintain the compute set across parallel regions, allowing the expression of different codelet types. The body of the OpenMP `task` region defines the codelet.

2.3 Computational Graph and Data

The variables defined inside the OpenMP `depend` clause correspond to inputs and outputs [`in, out, inout`] of the task region. The input/output dependencies among vertices across compute sets define the computational graph, expressing the relationship between compute and data. We use the `omp_get_thread_num()` API call to determine tensor sections.

3 Implementation

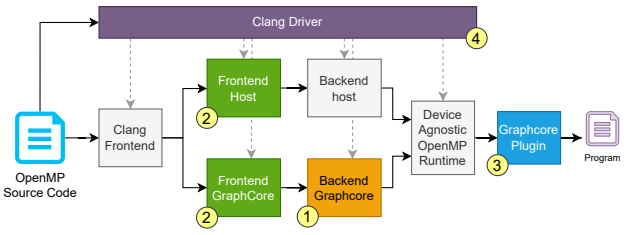


Figure 3: Implementation of OpenMP Support in LLVM. Modifications needed as they relate to the following subsections

The overall compiler infrastructure was previously described in [2], as a result of last year's LDRD expedition. Figure 3 shows the compilation pipeline used by our integration. This takes advantage of already existing modifications of the clang driver that allow to call different compilers multiple times to differentiate code generation across different target regions. Previously, the `popc` was a closed-source compiler. During the past year, GraphCore has opened the source code for its compiler [4]. As part of this year's effort, we have integrated the backend into our LLVM repository [8], reducing errors during code generation. The rest of this document focuses on the Front end modifications and the runtime system.

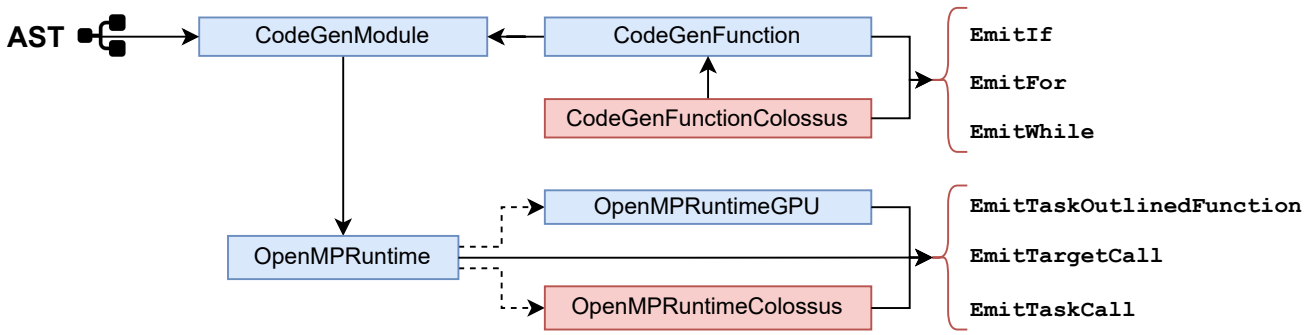


Figure 4: Part of codeGen infrastructure in LLVM. In red, are additions created to emit LLVM-IR compatible with the new Runtime Plugin for the GraphCore architecture. LLVM-IR is generated based on the architecture.

3.1 Frontend

By taking advantage of already existing offloading support [1], we can create the necessary infrastructure to emit LLVM-IR compatible with the GraphCore IPU architecture. In LLVM, the front end is called multiple times for the different targets when building OpenMP offloading code. It is possible to change the code generation based on the target architecture. This allows us to change the code generation behavior to accommodate for the generation of the two programs shown in Figure 1.

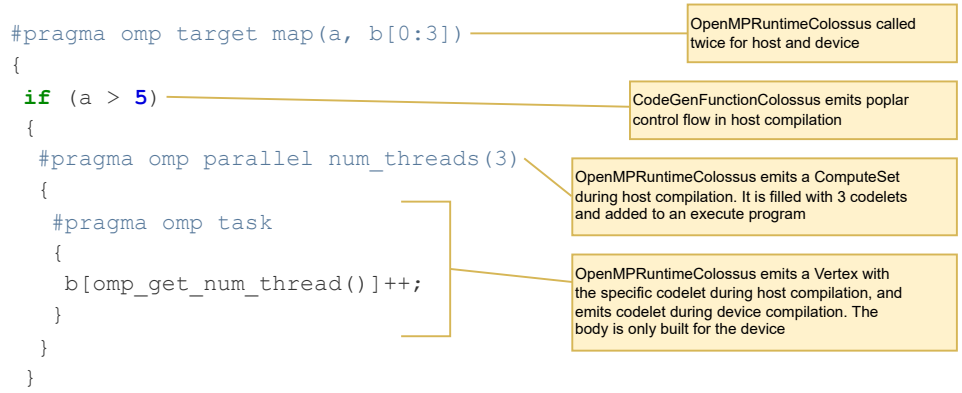


Figure 5: Code Generation for OpenMP code with the Colossus front-end infrastructure. This highlights the role of the different modifications in Figure 4

Figure 4 shows three components of the LLVM Code Generation part of Clang. After the AST is generated, the `CodeGen` pass generates LLVM-IR. The *CodeGenModule* class contains the context for the different functions of the current LLVM module being generated. Based on the target architecture, this class instantiates a version of the *OpenMPRuntime* class. We create the *OpenMPRuntimeColossus* class that inherits from *OpenMPRuntime* and overloads some of the methods to generate OpenMP Code. We show some examples of methods on the right side of the figure. This class is used both for host and device compilation. Finally, the *CodeGenFunction* usually emits code for simple control flow operations. However, due to the nature of the `poplar::Program` (Described in the introduction of this document), we must emit function calls instead of simple branches. Therefore, we create a second *CodeGenFunctionColossus* that changes the behavior of these methods while bypassing unknown methods to the original *CodeGenFunction*. Figure 5 shows an example program annotated. The annotations relate to the role that each new component in red in Figure 4 has.

3.2 Plugin

Adding a new OpenMP plug-in for a device is relatively simple. Libomptarget has been engineered to have a well-defined interface a device needs to support. The most important functions currently include: `register_lib`, `unregister_lib`, `is_valid_binary`, `init_device`, `load_binary`, `data_{alloc, submit,`

`retrieve, delete, exchange`}, and `run_{target_region,target_teams_region}` The difficulties, however, are that the Poplar programming model fundamentally differs from these interfaces inherently inspired by GPUs.

We have incorporated a proof-of-concept plugin into *libomptarget* that effectively generates and executes a Poplar program by offering a minimal API for the front end to create calls. This capability allows the plugin to instantiate and manage the Graphcore programs representing each OpenMP target region. The plugin creates the necessary boilerplate to run the Graphcore program. In addition to the agnostic interface functions, this initial implementation also includes routines for creating a new program, a new compute set, adding vertices, and executing the program.

This setup allows the plugin to construct the Graphcore programs by making calls to the Poplar API, essentially adding vertices to compute sets and subsequently triggering their execution. These functions depend on a codelet definition file generated by the front end in advance.

When the execution reaches a target region, the OpenMP runtime will invoke our plugin. Subsequently, the plugin will create the Graphcore program and add each task as a vertex to it. Before execution, the plugin will evenly distribute each vertex to an IPU tile and, in the end, initiate its execution.

4 Conclusion and next steps

Although OpenMP is designed to support a wide range of devices, the current specification has been heavily influenced by GPGPU-like systems. As a result, most of the target regions are intended to support Single Program Multiple Data (SPMD) execution. Compiler implementations have also exposed functions directly to the runtime API that follows this model. However, there is little support for devices that require an explicit description of graphs. Although GPU architectures could also benefit from this description (e.g., CUDA graphs), AI accelerators often take advantage of these program descriptions. While the OpenMP specification does not restrict the use of `task` within `target`, the description of static task creation requires further exploration and clarification. Additionally, inter-device data distribution, or mapping clauses that do not map the entire array but rather perform some form of data distribution, is an area for improvement if OpenMP intends to support these programming models. Finally, task placement (affinity) is not a straightforward process.

4.1 Limitations of this work

The current implementation has been trumped by limitations in the Poplar SDK. Although, we have been successful in generating Codelets using `popc`, the runtime system throws an error for the lack of CPU code. We are currently exploring if this limitation can be overcome, or if the driver would require additional tunning for generating a CPU version as well. The work performed in the front end is preliminary and incomplete. The infrastructure to develop the features is present, but we must implement more support for the architecture.

Just like any other device currently supported by OpenMP offloading, there are certain features that this work does not yet support. For instance, virtual graphs, mapping of tensors and vertexes to specific tiles, optimizations, or task-level inter-IPU programming are currently not supported. Additionally, we have limited the definition of tasks to represent compute sets, which means that cross-dependencies between compute sets may not necessarily respect OpenMP semantics. Furthermore, control flow within parallel regions has been restricted to represent the full parallel nature of compute sets. Lastly, there are several Poplar SDK functions that have not been explored yet, such as vector transpose.

Some other limitations are hardware constraints. First, given the small IPU memory, if streaming from the CPU host is not enabled during data mapping, it is not possible to fit really large applications. Another limitation is that the IPU architecture does not support double precision floating point units.

4.2 Future work

It is necessary to find applications that properly map to this new programming model. We believe that applications such as signal processing, and filtering, or those with long pipelines that could exploit streaming are good candidates. There are still open questions w.r.t. how to map programs to this architecture. How can we describe the characteristics of the architecture in a way that can engage teams?

5 Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

References

[1] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading support for openmp in clang and llvm," in *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2016, pp. 1–11.

[2] J. M. Diaz, E. Rangel, S. Raskar, and J. Doerfert, "A pathway to openmp in the graphcore architecture," 2022.

[3] Graphcore, "BOW IPU Processor," https://www.graphcore.ai/bow-processors, online; accessed 14 Oct 2022.

[4] GraphCore, "LLVM fork repository for popc compiler," https://github.com/graphcore/llvm-project-fork.

[5] ——, "POPLAR AND POPLIBS API REFERENCE," https://docs.graphcore.ai/projects/poplar-api/en/latest/index.html, online; accessed 16 Oct 2023.

[6] Graphcore, "Poplar Tutorial 3: Writting vertex code," https://github.com/graphcore/tutorials/tree/sdk-release-3.0/tutorials/poplar/tut3_vertices, online; accessed 14 Oct 2022.

[7] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore IPU architecture via microbenchmarking," *CoRR*, vol. abs/1912.03413, 2019. [Online]. Available: http://arxiv.org/abs/1912.03413

[8] JM. Monsalve Diaz and R. Ceccato de Freitas and E. Rangel and S. Raskar, "LLVM repository containing limited support for GraphCore," https://github.com/josemonsalve2/llvm-project/tree/graphcore.

[9] Karl Freund and Patrick Moorhead, "THE GRAPHCORE SECOND GENERATION IPU," https://www.graphcore.ai/mk2-ipu-m2000-white-paper, online; accessed 14 Oct 2022.

[10] H. Yviquel, L. Cruz, and G. Araujo, "Cluster programming using the openmp accelerator model," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, aug 2018. [Online]. Available: https://doi.org/10.1145/3226112

Towards rapid 3D X-ray Imaging of Nanocrystals at APS-U resolutions enabled by Physics-Informed AI Models on SambaNova

Henry Chan, Assistant Scientist (NST)
Mathew Cherukara, Computational Scientist, Group Leader (XSD)
Ross Harder, Physicist (XSD)

October 2023

1 Introduction to the Science problem

Nanocrystals posses unique size-dependent properties, making them crucial in various applications including catalysis, energy devices, and photonics. Despite their nanoscale dimensions, a novel 3D X-ray imaging technique called Bragg Coherent Diffraction Imaging (BCDI) enables the simultaneous characterization of their three-dimensional local structures and strains. This technique holds great potential for studying dynamic phenomena nanocrystals such as their growth and dissolution. With the forthcoming Advanced Photon Source Upgrade (APS-U), BCDI is expected to gain unprecedented imaging resolutions that will provide important insights into nanoscale phenomena like interface/surface reconstruction, dislocations, defects, and solid-solid phase transitions.

The retrieval of nanocrystal information in real-space from the measured diffraction images is challenging, partly due to a partial loss of phase information during the measurement step. Traditionally, the real-space information is retrieved using slow iterative algorithms with no guaranteed convergence. Our AI system leverage a scalable physics-aware neural network, AutoPhaseNN, to enhance the performance and applicability of traditional phase-retrieval algorithms in BCDI. The resolution of input images supported by AutoPhaseNN is mostly dictated by hardware memory available to support the training process. With the use of SambaNova, we aim to overcome memory limitations of GPU-based systems and enable AutoPhaseNN to process input diffraction images of nanocrystals closer to the upcoming imaging resolutions brought forth by the APS-U.

2 Description of the AI model and implementation

Our AI model, AutoPhaseNN, addresses the phase retrieval problem in BCDI by predicting the shape and local strain field of nanocrystals from their diffraction images. AutoPhaseNN is an autoencoder-based convolutional neural network (CNN) with a physics-informed feedback loop. Both encoder and decoder blocks consist of Convolutional layers, Leaky ReLu activation functions, and BatchNorm. The physics-informed feedback loop incorporates a forward model that performs Fast Fourier Transform (FFT). Prior to this expedition, we trained the model on 8 Nvidia A100 GPUs using 64^3 3D images, which were upsampled from 32^3 images generated through atomistic simulations and unlabeled experimental images. The code is available in both TensorFlow and PyTorch implementations at `https://github.com/mcherukara/AutoPhaseNN`. The model has been successfully deployed on LCRC Swing, ALCF Theta GPUs, and at the sector 34-ID beamline for inference.

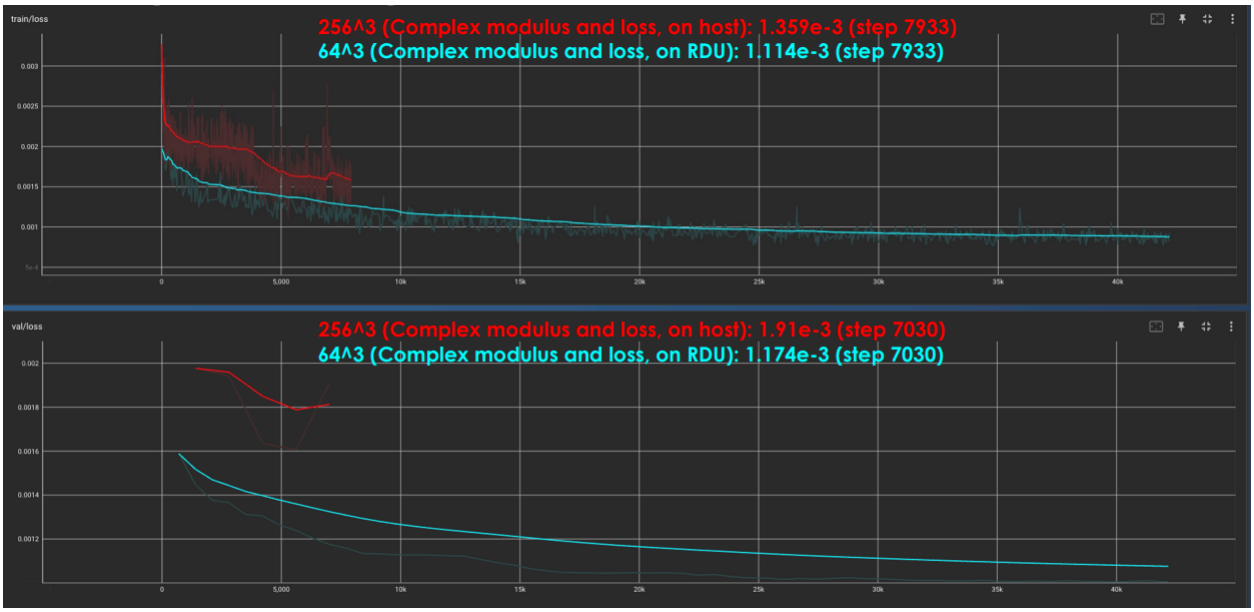


Figure 1: Comparison between training and validation loss of 64-sized and 256-size models.

3 What was needed to get the model running on the AI Accelerator

To deploy AutoPhaseNN on SambaNova, we began with the PyTorch version of our GPU implementation. The only substantial change required was the conversion from PyTorch tensors to SambaTensors for the RDUs. To maintain consistency in training accuracies across developers and runs, we stabilized our Python/Conda environment with PyTorch version 1.12. We experimented with batch sizes of 2, 32, and 2048 to observe their effects.

4 Performance Evaluation

Our evaluation of the AutoPhaseNN model on SambaNova focused on two model sizes: 64^3 (the largest achieved on GPUs) and 256^3 (the resolution required for APS-U). We set two main goals: 1) achieving training accuracy parity between 64-sized models on GPUs and RDUs, and 2) enhancing training efficiency so that 256-sized models can be trained on RDUs in a reasonable time frame. For Goal 1, collaboration with the SambaNova team was pivotal in addressing various issues, identified by varying floating point precision and selective execution of operations on host (CPU) and RDU. Once these issues were resolved by the SambaNova compiler team, we restored the large batch size and moved the optimizer back to RDU. For Goal 2, the SambaNova team implemented DataLoader and graph improvements to slightly reduce epoch-to-epoch latency and increase throughput. At batch size 32, the latest latency and throughput for 64-sized models on GPU vs. RDU are 3.3 vs. 8.65 mins/epoch and 229 vs. 87 samples/sec, respectively.

While the performance of 64-sized models on RDUs lags behind GPUs, we anticipate that this gap can be narrowed with further benchmarking. To strike a balance between the two goals, we shifted our focus towards making 256-sized models functional on RDUs. In the last month, we identified new compiler issues after moving complex number and loss function operations to CPUs. Our latest interaction with the SambaNova team revealed that one source of the issue is related to an edge case when Conv-tiling is enabled. After this has been addressed, we started training the 256-sized model on RDU using upsampled datasets and have begun to compare the results with 64-sized models (see Figure 1).

5 Conclusion and next steps

In conclusion, our research has demonstrated the possibility to deploy AutoPhaseNN on SambaNova RDUs. In particular, the 64-sized models provide valuable comparison between the performance of GPUs and RDUs whereas the 256-sized models highlight the unique capability of SambaNova AI accelerator, enabling future handling of high resolution 3D images obtainable via APS-U that may be challenging for GPU-based systems.

Overall, the successful deployment of these models on SambaNova accelerators represents a significant milestone although the training efficiency on these accelerator systems need to be signigicantly improved in order for the training to converge. Moving forward, we outline several key steps:

1. Regenerating training datasets at native model input resolutions to evaluate the impact of different model sizes without upsampling procedures.

2. Supplementing experimentally obtained samples to assess the effect of downsampling and compare our AI approach with traditional iterative methods.

3. Further exploring the capabilities of 256-sized models on RDUs and narrowing the 64-sized model performance gap with GPUs through continued benchmarking.

6 Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory, and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

Work performed at the Center for Nanoscale Materials and Advanced Photon Source, both U.S. Department of Energy Office of Science User Facilities, was supported by the U.S. DOE, Office of Basic Energy Sciences, under Contract No. DE-AC02-06CH11357.

Foundation Vision Models for Robotic Surgery

Neil Getty, Assistant Computational Scientist Ruoxi Zhao, Research Aide
Fangfang Xia, Computational Scientist

October 2023

1 Introduction to the Science problem

AI has the potential to improve training, augment performance, understand outcomes, or even automate surgery. Automatically detecting and localizing the different surgical instruments in endoscopic videos is a vital step towards context awareness in robotic surgery.

Recent advancements in foundational language and vision models have achieved state-of-the-art accuracy. For example, Meta AI's Segment Anything Model (SAM) setting benchmarks in image segmentation after training on over 100 million images and one billion masks. Yet, there are still gaps in these model's ability to understand and respond to tasks in specific areas, particularly scientific and biomedical domains.

As part of this study, we investigated the performance of several foundational vision and vision language models applied to minimally invasive robotic surgery. We were particularly interested in vision language models which present an opportunity to prompt these models in a natural, intuitive way even for those without a computing or data science background. For instance, one can segment all surgical tools within a robotic surgery scene with simple text prompts like "metal" or "tool". We experimented with AI-generated descriptions from CLIP, and using those as text prompts for SAM, and identified several challenges due to the both models limited surgical domain expertise. We test a similar process using DINOv2, Grounded DINO and SAM.

Our goal is to bridge the gaps between emerging AI capabilities and the medical domain, and ultimately assist surgeons in training, review, and real-time augmentation of surgical scenes with rich context awareness. At the same time, we recognize this scientific problem is uniquely challenging in key areas potentially addressed by non-traditional AI hardware, (1) overhead and latency in real-time application can be extremely important, (2) resolution, frame rate, length, and dimensionality of the data could easily exceed memory limitations on traditional hardware, (3) recent foundation vision models are growing in size and complexity and training these models becoming more expensive/time-consuming. We were thrilled to help explore the AI testbed for this problem and hope the testbed continues to grow and address these challenges in this and challenge-related domains.

2 Description of the AI model and implementation

CLIP is a vision language model trained on pairs of text and images to learn a joint representation across images and text. We experimented with using the CLIP as first a text decoder to generate a description for each surgical instrument, then we used these descriptions to compare with segmented regions from SAM to segment out the corresponding tools. Our findings indicate that the words "metal", "handle", and "knife" usually generate a mask that covers all the existing tools in the image, but not specific ones.

Segment anything model (SAM) developed by Meta, which is a foundation model in image segmentation that was trained on one billion masks and 11 million images, to segment the desired surgical instrument based on the bounding box prompts. Since SAM was not specifically trained on surgery videos, it lacks the domain knowledge in identifying and segmenting the entire surgical tools, so we endeavoured to fine-tune it on domain data. We experimented with fine-tuning SAM 2 on our customized dataset, the MICCAI challenges in 2017 and 2018 on segmenting surgical tools. SAM takes bounding boxes, points, or text as

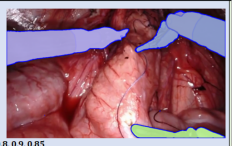
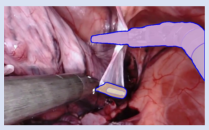
Surgical Tool Name	Segmentation using SAM	Text Decoder from CLIP (Classic Mode, CLIP model: ViT-L-14/openai, Blip-large)	Feeding Text Decoder to SAM with CLIP (With context prompt only)	Testing on surgery frames with context prompt only
Needle Driver		there is a close up of a metal object with a black background, a digital rendering inspired by Katsukawa Shunshō, polycount, cobra, holding dagger, closeup of fist, pen		 0.8, 0.9, 0.85
Cadiere Forceps		there is a metal object with a metal handle on it, a digital rendering inspired by Jozef Israëls, cg society, figuration libre, mechanical paw, neck shackle, large chain		 0.9, 0.8, 0.85

Figure 1: Two examples of utilizing CLIP-interrogator to get the text decoder, using the keywords as text prompt to generate mask from SAM, and comparing results.

prompts, in this case, we are using the bounding boxes, where they correspond to different parts of the surgical instruments.

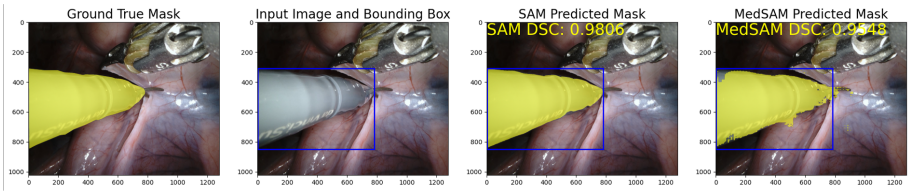


Figure 2: An example of the ground true mask, the input image and bounding box, and predicted mask from SAM with a mean dice similarity coefficient(DSC) score and comparing with the predicted mask from the fine-tuned model MedSAM with a DSC score. A higher DSC score means that the predicted mask has more pixel-wise agreement to the ground truth mask.

DINOv2 is a fully self-supervised vision transformer model with diverse downstream capabilities including segmentation, classification, depth estimation, and image retrieval. A diverse dataset of 142 million images was used to train the model. We experimented with DINOv2 for classification, depth estimation, and unsupervised clustering of the latent representations. Further, we experimented with Grounded DINO, an extension that allows natural language prompting to produce bounding boxes of objects in images. Combined with SAM we can go straight from language to segmented regions of an image as seen in 3.

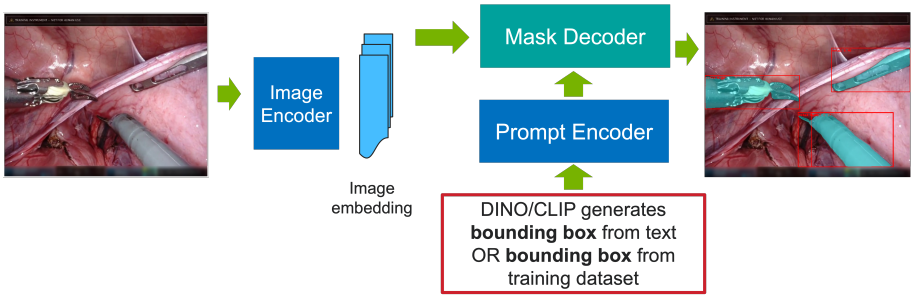


Figure 3: Grounded Dino generates bounding boxes according to the text prompt “metal”, SAM then generates mask over the tool areas inside the bounding boxes.

3 What was needed to get the model running on the AI Accelerator

Two models were configured to run on the SambaNova Datascale SN30 system, the Segment Anything Model (SAM) and DINOv2. SAM was not able to compile properly, and was abandoned. The error was unclear, compilation resulted an infinite loop of the form: Found existing tensor name

sammodel__vision_encoder__layers__0__attn__reshape_1__outputs__0_slice_0, tensor renamed to

sammodel__vision_encoder__layers__0__attn__reshape_1__outputs__0_slice_0_X. This message continued to populate with no end. Compiling the DINO model resulted in the following error: AssertionError: Only support modes nearest, bilinear, trilinear at the Samba level, provided mode bicubic. This was fairly easily fixed by changing the interpolation method the model uses to trilinear, it is assumed the effect to the output will be minimal but not nonexistent. Following the tutorials to run inference with pytorch dataloaders converted with the SambaLoader() method resulted in Tile Fault: Fatal tile fault error. Instead, the samba.from_torch_tensor() method was used. According to the documentation, this may result in decreased performance. One minor inconvenience was the need to manually set all script arguments explicitly. Without setting all arguments, compilation would error out with: AttributeError: 'Namespace' object has no attribute.

Overall the experience of converting a Huggingface model and custom pytorch dataloaders was challenging but manageable. Without direct communication with support, there are few ways to get insights into solutions, as unlike with popular hardware/packages there is little or no existing community support.

4 Performance Evaluation

Inference of DINOv2 was benchmarked on both GPU and RDU on a subset of 1000 surgical images with 512x640 resolution. One GPU was utilized for these tests, and 7 RDU tiles. In 4 we can see that increasing the batch size on GPU decreased speed, while on RDU speed was increased to a point. The IO overhead may be clearly observed in 5, where the batch size is very small. The idle times are when data is loaded on chip, in 6 we see this occurs only a single time.

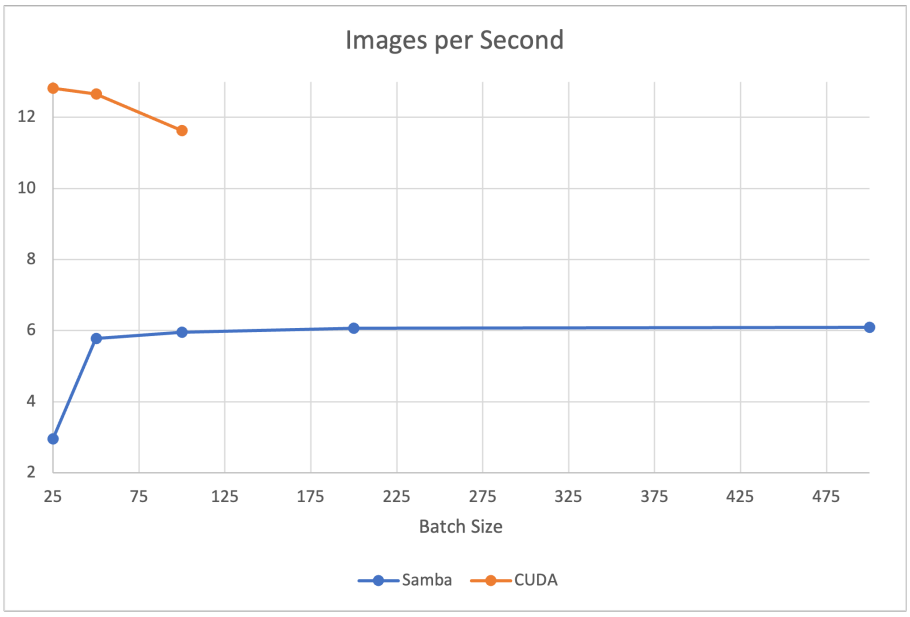


Figure 4: Comparison of GPU and Datascale iterations per second with varying batch sizes.

SAM was trained on a dataset with 6270 training images with 21128 masks, and 93,728,252 trainable parameters in the image encoder and mask decoder. The images original sizes were 1024*1280 and 1080*1920

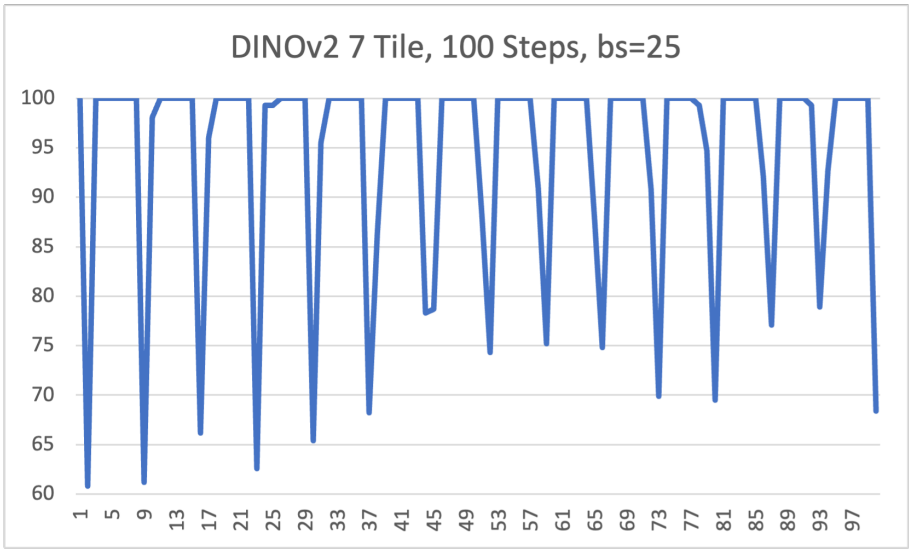


Figure 5: Samba Tile execution over 100 steps.

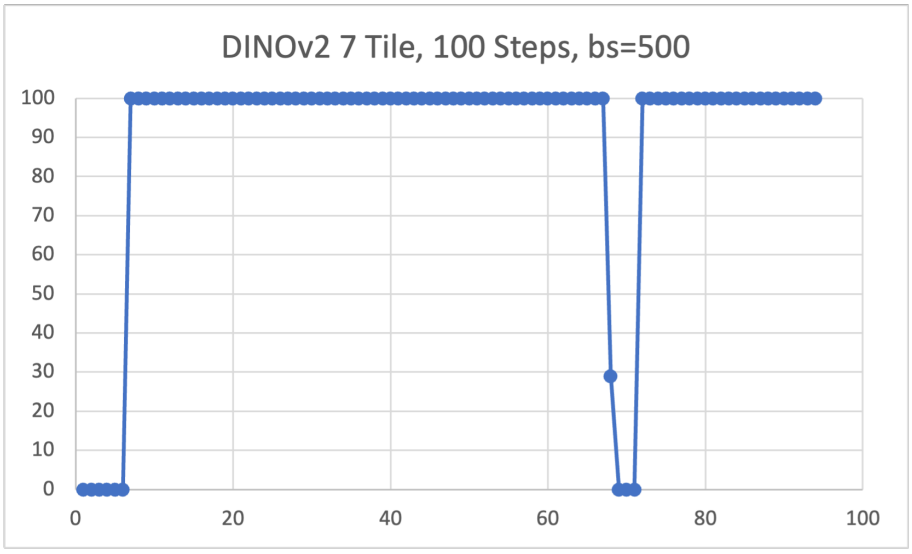


Figure 6: Samba Tile execution over 100 steps with higher batch size.

pixels, which we resized and padded to 1024*1024 during the preprocessing. We trained the model on four Tesla V100-SXM2-32GB GPUs with a batch size of 3, learning rate of 0.0008. Each GPU was allocated with 29 GBs CUDA space after loading the entire dataset, and each epoch took around 11 minutes. 67 epochs was sufficient to minimize with a loss of around 0.15. We are still interested in comparing this training benchmark with the Samba system, but were unable to compile SAM.

5 Conclusion and next steps

From our initial exploration here, we see a gap in inference speed between GPU and RDU 4, however we may need to ensure the effect of being unable to use the SambaLoader is not the significant bottleneck. The V100 GPU nodes were able to handle significantly fewer images overall, and it is still possible that when exploring vision language models with larger resolutions or video datasets that unique challenges will arise. While the models we experimented with are small by no means, it may be more appropriate to look at much larger vision models. The recent RT-2 by Deepmind is a vision language action model that makes use of a ViT backend with 4B paramters. Vision transformer models up to 22B parameters have been developed, with available datasets that warrant the increased parameter space. While this effort has proven it non-trivial to convert and deploy a model on RDU, I still see the potential for this and other domains. The overhead to learning a new framework/hardware may be fairly high, but is clearly necessary to make sure we are not limited by our own comfort in existing technologies.

6 Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory, and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

Pushing the mapping limits of the cosmological evolution

Nesar Ramachandra, Assistant Computational Scientist, CPS
Azton Wells, Postdoctoral candidate, CPS

October 2023

1 Introduction to the Science problem

Understanding the evolution of structures in the Universe is a complex problem that requires sophisticated numerical simulations to model the dynamics of matter and gravity on large scales. These traditional simulation methods trace the motion of the matter with cosmic time. This process is computationally expensive, which limits our ability to explore the full range of cosmological scenarios and to test the accuracy of theoretical models against observational data. Besides, for problems involving covariance studies or Bayesian inference, mock Universes have to be simulated thousands of times.

This motivates the exploration of alternative approaches that can generate inexpensive cosmological simulations. One such approach is machine learning-based image-to-image translations and generative models, which involve training a neural network to learn the mapping between initial and final dark matter particle positions. This approach has the advantage of generating simulations much faster than traditional methods, while also allowing for greater flexibility and control over the simulation parameters. However, there are many technical and scientific challenges associated with this approach, including the need for large and diverse training datasets, the choice of appropriate loss functions and validation metrics, and the need to ensure that the simulations are physically meaningful and consistent with observational data from telescopes.

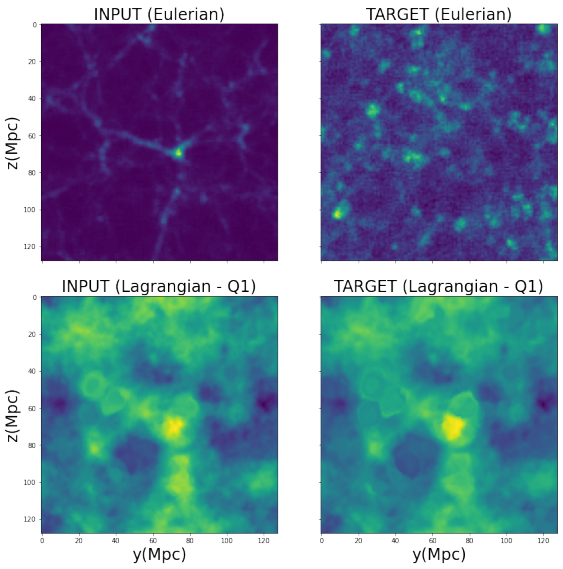


Figure 1: Input and Target cosmic fields. 2D projections of the 3D fields are shown.

In this project, we aim to increase the limits of our 3D voxel translation network. While the numerical simulations of cosmological evolution (such as Argonne's Hardware/Hybrid Accelerated Cosmology Code) simulate particles close to 2 trillion, the current state-of-the-art AI-based cosmology reconstructions have barely achieved scaling to 100 million particles. This is primarily due to the limitations in the GPU memory and the complexities involved in parallelizing training schemes, such as periodicity and long-range gravitational effects. We use a particle-mesh numerical method for cosmic evolution simulations to generate the training datasets for the neural network. These methods provide a way to model the dynamics of matter and gravity in the Universe, and to generate realistic initial and final fields that can be used as input and target data for the neural network. We convert the particle positions to their Lagrangian co-ordinates (related to initial grid, bottom panels of Figure 1) so that the same grid point represents the co-ordinate value of the same particle at all times. Corresponding matter density fields are shown in the top panels of Figure 1). The input fields are chosen at time-steps of redshift $z_0 = 0.4$ (cosmic age of roughly 9.8 billion years) and the target field is at $z_1 = 0.1$ (cosmic age of roughly 12.5 billion years).

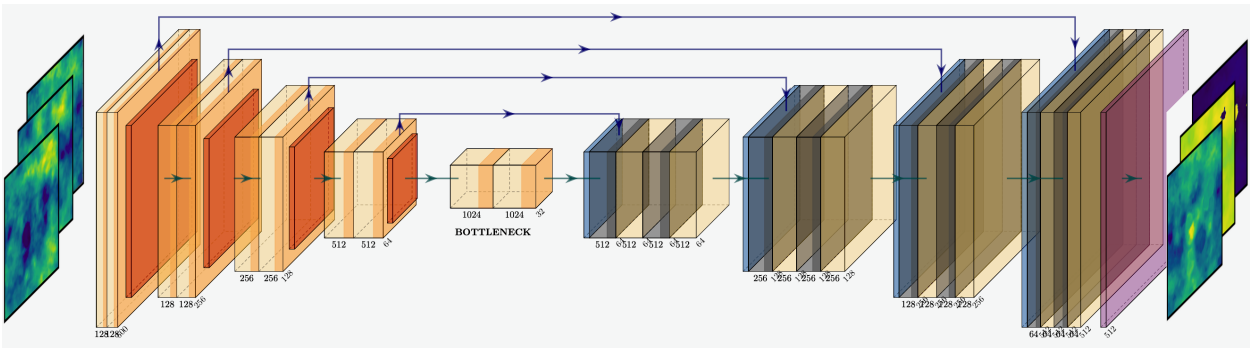


Figure 2: The 3D U-Net architecture for cosmic evolution mapping. The three channel inputs and targets correspond to Lagrangian displacement fields in three dimensions.

2 Description of the AI model and implementation

A U-Net architecture is used to model the evolution of cosmological structures. It is composed of a contracting path and an expansive path, both consisting of convolutional layers. The contracting path acts as a feature encoder and is made up of a sequence of 3D unpadded convolutional layers. Each of these layers is followed by an activation layer (Rectified Linear Unit, or 'ReLU') and 3D max pooling layers. With each downsampling step, the number of feature maps doubles. This allows the network to capture increasingly complex features and patterns in the input data.

The expansive path serves as the decoder and is designed to mirror the contracting path. It incorporates upsampling layers to enlarge the spatial dimensions. This path is structured with upsampling layers (or ConvTranspose) followed by 3D convolution layers. Each convolution reduces the number of feature maps by half and is concatenated with the corresponding feature maps from the contracting path. Subsequently, additional convolutional layers with activation functions are applied. In the upsampling segments, the substantial number of feature channels ensures the flow of context information to higher-resolution layers. Figure 2 shows the architecture with the encoder, decoder and the connections.

The basic composing unit for U-Net is transposed convolutional layers followed by ReLU activation and batch normalization layers. Starting from the input feature map, it goes through a couple of 3x3 convolution layers with strides 1 and 2. The first few layers have an output number of channels from 64, 128 to 256, forming an expansion path. The set of 5 layers is each followed by a batch normalization layer and a ReLU layer. They are then connected to a periodic padding layer, and the resulting middle layer gets chopped and concatenated with previous layers, then goes through a series of convolution layers with shrinking feature maps, leading to the final output result. During the training of the 3D U-Net, a pair of initial-final displacement fields of the dark matter particles are selected randomly from the training set as the input, and the neural network is trained to learn the mapping between the z_0 and z_1 displacement fields.

3 What was needed to get the model running on the AI Accelerator

Initially, we began by adapting our existing PyTorch implementation to SambaFlow, guided by the tutorials from ALCF workshops on AI-testbeds. However, the intricacies of our original code, which involved complex Fourier transforms, made the conversion challenging. We then explored modifying existing codes for 3D segmentation to suit our needs. This approach also presented difficulties due to the input-output format of multiple channels in our problem. In the end, a hybrid method proved successful: we developed a custom code, incorporating significant portions from existing segmentation implementations. This was successfully compiled and executed on SambaNova RDUs.

Our initial goal was to adjust the shape of the input/outputs to 3(channel)$\times$(128$\times$128$\times$128). However, this has not been achieved yet, as compiling our 3D U-Net code on a single RDU demands more resources than the chip can provide. As of now, we've successfully compiled and trained smaller models with input/output

Train-test split	Input-output size	Architecture	Time/epoch/N_{train}
400/40	(3x48x48x48)	SN	1.2 seconds
400/40	(3x32x32x32)	SN	0.6 seconds
20/40	(3x48x48x48)	V100	3.6 seconds
20/40	(3x32x32x32)	V100	2.1 seconds

Table 1: Training time comparison between SambaNova and NVIDIA-V100 GPU node. The training on SambaNova is done for all the available training points ($N_{train} = 400$. Whereas the NVIDIA tests are only done with 20 training points.

shapes of 3(channel)×(32×32×32) and 3(channel)×(48×48×48). Moreover, the batch sizes in the training has been limited to 2. With a compiled model, we have tested both single-RDU and multiple-RDU training.

4 Performance Evaluation

We evaluate the performance of 3D U-Net model on SambaNova machine. The training data consists of $N_{train} = 400$ pairs of input-output simulations, where each data-point is a 3(channel)×(32×32×32) or 3(channel)×(48×48×48). We also have test and validation datasets of size 200 each. Within the U-Net, mean square error (MSE) between the target and predicted Lagrangian co-ordinates is used as the loss function. We also use AdamW optimization method for optimizing the weights.

The timing tests have revealed SambaNova to be faster in training time over traditional architectures. For (3x48x48x48) box, training on RDU took roughly 1.2 seconds-per-epoch-per-N_{train}. This number was halved when the box size was reduced to (3x32x32x32). The corresponding numbers on GPU node were 3.6 seconds and 2.1 seconds respectively. However, there are considerably differences in the scripts and I/O between the SambaFlow and native PyTorch implementation, and further timing studies have to be conducted.

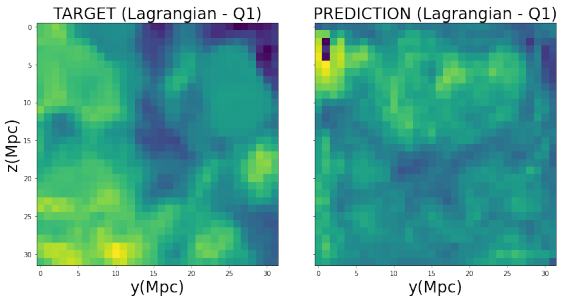


Figure 3: Target fields and U-Net predicted fields.

We also note that our losses in the SambaFlow model have not converged so far. From our experience with GPU code, we desire deeper U-Net models with extensive hyper-parameter tuning. This two aspects have not been incorporated with our SambaNova implementation as of now. The resulting U-Net predictions for an arbitrary test data is seen in the right panel of Figure 3, where there are signatures of structures in Lagrangian space despite the model being sub-optimally trained. In addition, we also note that the simulation datapoints were cropped in order to fit to (32x32x32) or (48x48x48) models, resulting in residual edge effects in the U-Net predictions.

5 Conclusion and next steps

In this project, we have tested the AI-surrogate models for numerical simulation of cosmological evolution. Having successfully implemented our current datasets and a 3D U-Net model, the next step is hyper-parameter tuning to arrive at optimal models. Given the limited amount of simulation datasets (roughly 400 datapoints for training), we have found that model optimization is crucial in 3D voxel mapping. We are also eager to explore the following advancements in the near future:

- Scaling up the model from (48×48×48) to accommodate larger data sizes, and simultaneously increasing the batch size beyond 2. In addition, we would also like to try deeper U-Net models (upto 5 convolutional blocks in encoder and decoder each). This is a major hurdle that will require expertise from the developers at SambaNova, in terms of memory consumption and optimization of the code compilation.

- Integrating physics-informed loss functions, such as power spectra and bispectra, into the training of the U-Net models.
- Incorporating benchmarks like percolation statistics and mass functions to evaluate the models' performance.

Should these implementations prove the feasibility of capturing dynamics of the cosmic evolution, they could lead a transformative direction in cosmological data analyses. Such rapid emulation techniques would be invaluable for covariance studies and inferring cosmological parameters from upcoming telescope survey programs.

6 Acknowledgements

We thank Varuni Sastry (ALCF) for her invaluable support throughout the project, encompassing everything from tutorials to model implementation. We also thank Venkatram Vishwanath (ALCF) and Rick Weisner (SambaNova Systems) for their timely and insightful discussions.

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory, and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

Computing, Environment and Life Sciences
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC