Near-Data Processing

陈辉

Contents

Part I
文献

1 *Evolution-Strategies-as-a-Scalable-Alternative-to-Reinforcement-Learning*

(P1)

2 *NEAR-DATA-PROCESSING-FOR-MACHINE-LEARNING*

(P1)

3 *Practical-Near-Data-Processing-for-In-memory-Analytics-Frameworks*

See paper(P1) See presentation(P1)

Evolution Strategies as a Scalable Alternative to Reinforcement Learning

Tim Salimans **Jonathan Ho** **Xi Chen** **Szymon Sidor** **Ilya Sutskever**
OpenAI

Abstract

We explore the use of Evolution Strategies (ES), a class of black box optimization algorithms, as an alternative to popular MDP-based RL techniques such as Q-learning and Policy Gradients. Experiments on MuJoCo and Atari show that ES is a viable solution strategy that scales extremely well with the number of CPUs available: By using a novel communication strategy based on common random numbers, our ES implementation only needs to communicate scalars, making it possible to scale to over a thousand parallel workers. This allows us to solve 3D humanoid walking in 10 minutes and obtain competitive results on most Atari games after one hour of training. In addition, we highlight several advantages of ES as a black box optimization technique: it is invariant to action frequency and delayed rewards, tolerant of extremely long horizons, and does not need temporal discounting or value function approximation.

1 Introduction

Developing agents that can accomplish challenging tasks in complex, uncertain environments is a key goal of artificial intelligence. Recently, the most popular paradigm for analyzing such problems has been using a class of reinforcement learning (RL) algorithms based on the Markov Decision Process (MDP) formalism and the concept of value functions. Successes of this approach include systems that learn to play Atari from pixels [Mnih et al., 2015], perform helicopter aerobatics Ng et al. [2006], or play expert-level Go [Silver et al., 2016].

An alternative approach to solving RL problems is using black-box optimization. This approach is known as direct policy search [Schmidhuber and Zhao, 1998], or neuro-evolution [Risi and Togelius, 2015], when applied to neural networks. In this paper, we study Evolution Strategies (ES) [Rechenberg and Eigen, 1973], a particular set of optimization algorithms in this class. We show that ES can reliably train neural network policies, in a fashion well suited to be scaled up to modern distributed computer systems, for controlling robots in the MuJoCo physics simulator [Todorov et al., 2012] and playing Atari games with pixel inputs [Mnih et al., 2015]. Our key findings are as follows:

1. We found that the use of virtual batch normalization [Salimans et al., 2016] and other reparameterizations of the neural network policy (section 2.2) greatly improve the reliability of evolution strategies. Without these methods ES proved brittle in our experiments, but with these reparameterizations we achieved strong results over a wide variety of environments.
2. We found the evolution strategies method to be highly parallelizable: by introducing a novel communication strategy based on common random numbers, we are able to achieve linear speedups in run time even when using over a thousand workers. In particular, using 1,440 workers, we have been able to solve the MuJoCo 3D humanoid task in under 10 minutes.
3. The data efficiency of evolution strategies was surprisingly good: we were able to match the final performance of A3C [Mnih et al., 2016] on most Atari environments while using between 3x and 10x as much data. The slight decrease in data efficiency is partly offset by a

reduction in required computation of roughly 3x due to not performing backpropagation and not having a value function. Our 1-hour ES results require about the same amount of computation as the published 1-day results for A3C, while performing better on 23 games tested, and worse on 28. On MuJoCo tasks, we were able to match the learned policy performance of Trust Region Policy Optimization [TRPO; Schulman et al., 2015], using no more than 10x as much data.

4. We found that ES exhibited better exploration behaviour than policy gradient methods like TRPO: on the MuJoCo humanoid task, ES has been able to learn a very wide variety of gaits (such as walking sideways or walking backwards). These unusual gaits are never observed with TRPO, which suggests a qualitatively different exploration behavior.
5. We found the evolution strategies method to be robust: we achieved the aforementioned results using fixed hyperparameters for all the Atari environments, and a different set of fixed hyperparameters for all MuJoCo environments (with the exception of one binary hyperparameter, which has not been held constant between the different MuJoCo environments).

Black-box optimization methods have several highly attractive properties: indifference to the distribution of rewards (sparse or dense), no need for backpropagating gradients, and tolerance of potentially arbitrarily long time horizons. However, they are perceived as less effective at solving hard RL problems compared to techniques like Q-learning and policy gradients. The contribution of this work, which we hope will renew interest in this class of methods and lead to new useful applications, is a demonstration that evolution strategies can be competitive with competing RL algorithms on the hardest environments studied by the deep RL community today, and that this approach can scale to many more parallel workers.

2 Evolution Strategies

Evolution Strategies (ES) is a class of black box optimization algorithms [Rechenberg and Eigen, 1973, Schwefel, 1977] that are heuristic search procedures inspired by natural evolution: At every iteration ("generation"), a population of parameter vectors ("genotypes") is perturbed ("mutated") and their objective function value ("fitness") is evaluated. The highest scoring parameter vectors are then recombined to form the population for the next generation, and this procedure is iterated until the objective is fully optimized. Algorithms in this class differ in how they represent the population and how they perform mutation and recombination. The most widely known member of the ES class is the covariance matrix adaptation evolution strategy [CMA-ES; Hansen and Ostermeier, 2001], which represents the population by a full-covariance multivariate Gaussian. CMA-ES has been extremely successful in solving optimization problems in low to medium dimension.

The version of ES we use in this work belongs to the class of natural evolution strategies (NES) [Wierstra et al., 2008, 2014, Yi et al., 2009, Sun et al., 2009, Glasmachers et al., 2010a,b, Schaul et al., 2011] and is closely related to the work of Sehnke et al. [2010]. Let F denote the objective function acting on parameters θ. NES algorithms represent the population with a distribution over parameters $p_\psi(\theta)$—itself parameterized by ψ—and proceed to maximize the average objective value $\mathbb{E}_{\theta\sim p_\psi} F(\theta)$ over the population by searching for ψ with stochastic gradient ascent. Specifically, using the score function estimator for $\nabla_\psi \mathbb{E}_{\theta\sim p_\psi} F(\theta)$ in a fashion similar to REINFORCE [Williams, 1992], NES algorithms take gradient steps on ψ with the following estimator:

$$\nabla_\psi \mathbb{E}_{\theta\sim p_\psi} F(\theta) = \mathbb{E}_{\theta\sim p_\psi} \left\{F(\theta) \nabla_\psi \log p_\psi(\theta)\right\}$$

For the special case where p_ψ is factored Gaussian (as in this work), the resulting gradient estimator is also known as *simultaneous perturbation stochastic approximation* [Spall, 1992], *parameter-exploring policy gradients* [Sehnke et al., 2010], or *zero-order gradient estimation* [Nesterov and Spokoiny, 2011].

In this work, we focus on RL problems, so $F(\cdot)$ will be the stochastic return provided by an environment, and θ will be the parameters of a deterministic or stochastic policy π_θ describing an agent acting in that environment, controlled by either discrete or continuous actions. Much of the innovation in RL algorithms is focused on coping with the lack of access to or existence of derivatives of the environment or policy. Such non-smoothness can be addressed with ES as follows. We instantiate the population distribution p_ψ as an isotropic multivariate Gaussian with mean ψ and fixed covariance $\sigma^2 I$, allowing us to write $\mathbb{E}_{\theta\sim p_\psi} F(\theta)$ in terms of a mean parameter vector θ directly: we

set $\mathbb{E}_{\theta \sim p_\psi} F(\theta) = \mathbb{E}_{\epsilon \sim N(0,I)} F(\theta + \sigma\epsilon)$. With this setup, our stochastic objective can be viewed as a Gaussian-blurred version of the original objective F, free of non-smoothness introduced by the environment or potentially discrete actions taken by the policy. Further discussion on how ES and policy gradient methods cope with non-smoothness can be found in section 3.

With our objective defined in terms of θ, we optimize over θ directly using stochastic gradient ascent with the score function estimator:

$$\nabla_\theta \, \mathbb{E}_{\epsilon \sim N(0,I)} \, F(\theta + \sigma\epsilon) = \frac{1}{\sigma} \, \mathbb{E}_{\epsilon \sim N(0,I)} \left\{ F(\theta + \sigma\epsilon) \, \epsilon \right\}$$

which can be approximated with samples. The resulting algorithm (1) repeatedly executes two phases: 1) Stochastically perturbing the parameters of the policy and evaluating the resulting parameters by running an episode in the environment, and 2) Combining the results of these episodes, calculating a stochastic gradient estimate, and updating the parameters.

Algorithm 1 Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  Sample $\epsilon_1, \ldots \epsilon_n \sim \mathcal{N}(0, I)$
  Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
  Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
end for

2.1 Scaling and parallelizing ES

ES is well suited to be scaled up to many parallel workers: 1) It operates on complete episodes, thereby requiring only infrequent communication between workers. 2) The only information obtained by each worker is the scalar return of an episode: if we synchronize random seeds between workers before optimization, each worker knows what perturbations the other workers used, so each worker only needs to communicate a single scalar to and from each other worker to agree on a parameter update. ES thus requires extremely low bandwidth, in sharp contrast to policy gradient methods, which require workers to communicate entire gradients. 3) It does not require value function approximations. RL with value function estimation is inherently sequential: To improve upon a given policy, multiple updates to the value function are typically needed to get enough signal. Each time the policy is significantly changed, multiple iterations are necessary for the value function estimate to catch up.

A simple parallel version of ES is given in Algorithm 2. The main novelty here is that the algorithm makes use of shared random seeds, which drastically reduces the bandwidth required for communication between the workers.

Algorithm 2 Parallelized Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
Initialize: n workers with known random seeds, and initial parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  for each worker $i = 1, \ldots, n$ **do**
    Sample $\epsilon_i \sim \mathcal{N}(0, I)$
    Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
  end for
  Send all scalar returns F_i from each worker to every other worker
  for each worker $i = 1, \ldots, n$ **do**
    Reconstruct all perturbations ϵ_j for $j = 1, \ldots, n$ using known random seeds
    Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
  end for
end for

In practice, we implement sampling by having each worker instantiate a large block of Gaussian noise at the start of training, and then perturbing its parameters by adding a randomly indexed subset of these noise variables at each iteration. Although this means that the perturbations are not strictly

independent across iterations, we did not find this to be a problem in practice. Using this strategy, we find that the second part of Algorithm 2 (lines 9-12) only takes up a small fraction of total time spend for all our experiments, even when using up to 1,440 parallel workers. When using many more workers still, or when using very large neural networks, we can reduce the computation required for this part of the algorithm by having workers only perturb a subset of the parameters θ rather than all of them: In this case the perturbation distribution p_ψ corresponds to a mixture of Gaussians, for which the update equations remain unchanged. At the very extreme, every worker would perturb only a single coordinate of the parameter vector, which means that we would be using pure finite differences.

To reduce variance, we use antithetic sampling Geweke [1988], also known as *mirrored sampling* Brockhoff et al. [2010] in the ES literature: that is, we always evaluate pairs of perturbations $\epsilon, -\epsilon$, for Gaussian noise vector ϵ. We also find it useful to perform fitness shaping Wierstra et al. [2014] by applying a rank transformation to the returns before computing each parameter update. Doing so removes the influence of outlier individuals in each population and decreases the tendency for ES to fall into local optima early in training. In addition, we apply weight decay to the parameters of our policy network: this prevents the parameters from growing very large compared to the perturbations.

Unlike Wierstra et al. [2014] we did not see benefit from adapting σ during training, and we therefore treat it as a fixed hyperparameter instead. We perform the optimization directly in parameter space; exploring indirect encodings Stanley et al. [2009], van Steenkiste et al. [2016] is left for future work.

Evolution Strategies, as presented above, works with full-length episodes. In some rare cases this can lead to low CPU utilization, as some episodes run for many more steps than others. For this reason, we cap episode length at a constant m steps for all workers, which we dynamically adjust as training progresses. For example, by setting m to be equal to twice the mean number of steps taken per episode, we can guarantee that CPU utilization stays above 50% in the worst case.

2.2 The impact of network parameterization

Whereas RL algorithms like Q-learning and policy gradients explore by sampling actions from a stochastic policy, Evolution Strategies derives learning signal from sampling instantiations of policy parameters. Exploration in ES is thus driven by parameter perturbation. For ES to improve upon parameters θ, some members of the population must achieve better return than others: i.e. it is crucial that Gaussian perturbation vectors ϵ occasionally lead to new individuals $\theta + \sigma\epsilon$ with better return.

For the Atari environments, we found that Gaussian parameter perturbations on DeepMind's convolutional architectures [Mnih et al., 2015] did not always lead to adequate exploration: For some environments, randomly perturbed parameters tended to encode policies that always took one specific action regardless of the state that was given as input. However, we discovered thatwe could match the performance of policy gradient methods for most games by using virtual batch normalization [Salimans et al., 2016] in the policy specification. Virtual batch normalization is precisely equivalent to batch normalization [Ioffe and Szegedy, 2015] where the minibatch used for calculating normalizing statistics is chosen at the start of training and is fixed. This change in parameterization makes the policy more sensitive to very small changes in the input image at the early stages of training when the weights of the policy are random, ensuring that the policy takes a wide-enough variety of actions to gather occasional rewards. For most applications, a downside of virtual batch normalization is that it makes training more expensive. For our application, however, the minibatch used to calculate the normalizing statistics is much smaller than the number of steps taken during a typical episode, meaning that the overhead is negligible.

For the MuJoCo tasks, we achieved good performance on nearly all the environments with the standard multilayer perceptrons mapping to continuous actions. However, we observed that for some environments, we could encourage more exploration by discretizing the actions. This forced the actions to be non-smooth with respect to input observations and parameter perturbations, and thereby encouraged a wide variety of behaviors to be played out over the course of rollouts.

3 Smoothing in parameter space versus smoothing in action space

As mentioned in section 2, a large source of difficulty in RL stems from the lack of informative gradients of policy performance: such gradients may not exist due to non-smoothness of the environ-

ment or policy, or may only be available as high-variance estimates because the environment usually can only be accessed via sampling. Explicitly, suppose we wish to solve general decision problems that give a return $R(\mathbf{a})$ after we take a sequence of actions $\mathbf{a} = \{a_1, \ldots, a_T\}$, where the actions are determined by a either a deterministic or a stochastic policy function $a_t = \pi(s; \theta)$. The objective we would like to optimize is thus

$$F(\theta) = R(\mathbf{a}(\theta)).$$

Since the actions are allowed to be discrete and the policy is allowed to be deterministic, $F(\theta)$ can be non-smooth in θ. More importantly, because we do not have explicit access to the underlying state transition function of our decision problems, the gradients cannot be computed with a backpropagation-like algorithm. This means we cannot directly use standard gradient-based optimization methods to find a good solution for θ.

In order to both make the problem smooth and to have a means of to estimate its gradients, we need to add noise. Policy gradient methods add the noise in action space, which is done by sampling the actions from an appropriate distribution. For example, if the actions are discrete and $\pi(s; \theta)$ calculates a score for each action before selecting the best one, then we would sample an action $\mathbf{a}(\epsilon, \theta)$ (here ϵ is the noise source) from a categorical distribution over actions at each time period, applying a softmax to the scores of each action. Doing so yields the objective $F_{PG}(\theta) = \mathbb{E}_\epsilon R(\mathbf{a}(\epsilon, \theta))$, with gradients

$$\nabla_\theta F_{PG}(\theta) = \mathbb{E}_\epsilon \left\{ R(\mathbf{a}(\epsilon, \theta)) \nabla_\theta \log p(\mathbf{a}(\epsilon, \theta); \theta) \right\}.$$

Evolution strategies, on the other hand, add the noise in parameter space. That is, they perturb the parameters as $\tilde{\theta} = \theta + \xi$, with ξ from a multivariate Gaussian distribution, and then pick actions as $a_t = \mathbf{a}(\xi, \theta) = \pi(s; \tilde{\theta})$. It can be interpreted as adding a Gaussian blur to the original objective, which results in a smooth, differentiable cost $F_{ES}(\theta) = \mathbb{E}_\xi R(\mathbf{a}(\xi, \theta))$, this time with gradients

$$\nabla_\theta F_{ES}(\theta) = \mathbb{E}_\xi \left\{ R(\mathbf{a}(\xi, \theta)) \nabla_\theta \log p(\tilde{\theta}(\xi, \theta); \theta) \right\}.$$

The two methods for smoothing the decision problem are thus quite similar, and can be made even more so by adding noise to both the parameters and the actions.

3.1 When is ES better than policy gradients?

Given these two methods of smoothing the decision problem, which should we use? The answer depends strongly on the structure of the decision problem and on which type of Monte Carlo estimator is used to estimate the gradients $\nabla_\theta F_{PG}(\theta)$ and $\nabla_\theta F_{ES}(\theta)$. Suppose the correlation between the return and the individual actions is low (as is true for any hard RL problem). Assuming we approximate these gradients using simple Monte Carlo (REINFORCE) with a good baseline on the return, we have

$$\begin{aligned}
\mathrm{Var}[\nabla_\theta F_{PG}(\theta)] &\approx \mathrm{Var}[R(\mathbf{a})] \, \mathrm{Var}[\nabla_\theta \log p(\mathbf{a}; \theta)], \\
\mathrm{Var}[\nabla_\theta F_{ES}(\theta)] &\approx \mathrm{Var}[R(\mathbf{a})] \, \mathrm{Var}[\nabla_\theta \log p(\tilde{\theta}; \theta)].
\end{aligned}$$

If both methods perform a similar amount of exploration, $\mathrm{Var}[R(\mathbf{a})]$ will be similar for both expressions. The difference will thus be in the second term. Here we have that $\nabla_\theta \log p(\mathbf{a}; \theta) = \sum_{t=1}^{T} \nabla_\theta \log p(a_t; \theta)$ is a sum of T uncorrelated terms, so that the variance of the policy gradient estimator will grow nearly linearly with T. The corresponding term for evolution strategies, $\nabla_\theta \log p(\tilde{\theta}; \theta)$, is independent of T. Evolution strategies will thus have an advantage compared to policy gradients for long episodes with very many time steps. In practice, the effective number of steps T is often reduced in policy gradient methods by discounting rewards. If the effects of actions are short-lasting, this allows us to dramatically reduce the variance in our gradient estimate, and this has been critical to the success of applications such as Atari games. However, this discounting will bias our gradient estimate if actions have long lasting effects. Another strategy for reducing the effective value of T is to use value function approximation. This has also been effective, but once again runs the risk of biasing our gradient estimates. Evolution strategies is thus an attractive choice if the effective number of time steps T is long, actions have long-lasting effects, and if no good value function estimates are available.

3.2 Problem dimensionality

The gradient estimate of ES can be interpreted as a method for randomized finite differences in high-dimensional space. Indeed, using the fact that $\mathbb{E}_{\epsilon\sim N(0,I)}\left\{F(\theta)\,\epsilon/\sigma\right\} = 0$, we get

$$\nabla_\theta \eta(\theta) = \mathbb{E}_{\epsilon\sim N(0,I)}\left\{F(\theta+\sigma\epsilon)\,\epsilon/\sigma\right\} = \mathbb{E}_{\epsilon\sim N(0,I)}\left\{(F(\theta+\sigma\epsilon) - F(\theta))\,\epsilon/\sigma\right\}$$

It is now apparent that ES can be seen as computing a finite difference derivative estimate in a randomly chosen direction, especially as σ becomes small. The resemblance of ES to finite differences suggests the method will scale poorly with the dimension of the parameters θ. Theoretical analysis indeed shows that for general non-smooth optimization problems, the required number of optimization steps scales linearly with the dimension [Nesterov and Spokoiny, 2011]. However, it is important to note that this does not mean that larger neural networks will perform worse than smaller networks when optimized using ES: what matters is the difficulty, or intrinsic dimension, of the optimization problem. To see that the dimensionality of our model can be completely separate from the effective dimension of the optimization problem, consider a regression problem where we approximate a univariate variable y with a linear model $\hat{y} = \mathbf{x}\cdot\mathbf{w}$: if we double the number of features and parameters in this model by concatenating $\mathbf{x}$ with itself (i.e. using features $\mathbf{x}' = (\mathbf{x}, \mathbf{x})$), the problem does not become more difficult. The ES algorithm will do exactly the same thing when applied to this higher dimensional problem, as long as we divide the standard deviation of the noise by two, as well as the learning rate.

In practice, we observe slightly better results when using larger networks with ES. For example, we tried both the larger network and smaller network used in A3C [Mnih et al., 2016] for learning Atari 2600 games, and on average obtained better results using the larger network. We hypothesize that this is due to the same effect that makes standard gradient-based optimization of large neural networks easier than for small ones: large networks have fewer local minima [Kawaguchi, 2016].

3.3 Advantages of not calculating gradients

In addition to being easy to parallelize, and to having an advantage in cases with long action sequences and delayed rewards, black box optimization algorithms like ES have other advantages over RL techniques that calculate gradients. The communication overhead of implementing ES in a distributed setting is lower than for competing RL methods such as policy gradients and Q-learning, as the only information that needs to be communicated across processes are the scalar return and the random seed that was used to generate the perturbations ϵ, rather than a full gradient. Also, ES can deal with maximally sparse and delayed rewards; only the total return of an episode is used, whereas other methods use individual rewards and their exact timing.

By not requiring backpropagation, black box optimizers reduce the amount of computation per episode by about two thirds, and memory by potentially much more. In addition, not explicitly calculating an analytical gradient protects against problems with exploding gradients that are common when working with recurrent neural networks. By smoothing the cost function in parameter space, we reduce the pathological curvature that causes these problems: bounded cost functions that are smooth enough can't have exploding gradients. At the extreme, ES allows us to incorporate non-differentiable elements into our architecture, such as modules that use *hard attention* [Xu et al., 2015].

Black box optimization methods are uniquely suited to low precision hardware for deep learning. Low precision arithmetic, such as in binary neural networks, can be performed much cheaper than at high precision. When optimizing such low precision architectures, biased low precision gradient estimates can be a problem when using gradient-based methods. Similarly, specialized hardware for neural network inference, such as TPUs [Jouppi et al., 2017], can be used directly when performing optimization using ES, while their limited memory usually makes backpropagation impossible.

By perturbing in parameter space instead of action space, black box optimizers are naturally invariant to the frequency at which our agent acts in the environment. For MDP-based reinforcement learning algorithms, on the other hand, it is well known that *frameskip* is a crucial parameter to get right for the optimization to succeed [Braylan et al., 2005]. While this is usually a solvable problem for games that only require short-term planning and action, it is a problem for learning longer term strategic behavior. For these problems, RL needs hierarchy to succeed [Parr and Russell, 1998], which is not as necessary when using black box optimization.

4 Experiments

4.1 MuJoCo

We evaluated ES on a benchmark of continuous robotic control problems in the OpenAI Gym [Brockman et al., 2016] against a highly tuned implementation of Trust Region Policy Optimization [Schulman et al., 2015], a policy gradient algorithm designed to efficiently optimize neural network policies. We tested on both classic problems, like balancing an inverted pendulum, and more difficult ones found in recent literature, like learning 2D hopping and walking gaits. The environments were simulated by MuJoCo [Todorov et al., 2012].

We used both ES and TRPO to train policies with identical architectures: multilayer perceptrons with two 64-unit hidden layers separated by tanh nonlinearities. We found that ES occasionally benefited from discrete actions, since continuous actions could be too smooth with respect to parameter perturbation and could hamper exploration (see section 2.2). For the hopping and swimming tasks, we discretized the actions for ES into 10 bins for each action component.

We found that ES was able to solve these tasks up to TRPO's final performance after 5 million timesteps of environment interaction. To obtain this result, we ran ES over 6 random seeds and compared the mean learning curves to similarly computed curves for TRPO. The exact sample complexity tradeoffs over the course of learning are listed in Table 1, and detailed results are listed in Table 3 of the supplement. Generally, we were able to solve the environments in less than 10x penalty in sample complexity on the hard environments (Hopper and Walker2d) compared to TRPO. On simple environments, we achieved up to 3x better sample complexity than TRPO.

Table 1: MuJoCo tasks: Ratio of ES timesteps to TRPO timesteps needed to reach various percentages of TRPO's learning progress at 5 million timesteps.

Environment	25%	50%	75%	100%
HalfCheetah	0.15	0.49	0.42	0.58
Hopper	0.53	3.64	6.05	6.94
InvertedDoublePendulum	0.46	0.48	0.49	1.23
InvertedPendulum	0.28	0.52	0.78	0.88
Swimmer	0.56	0.47	0.53	0.30
Walker2d	0.41	5.69	8.02	7.88

4.2 Atari

We ran our parallel implementation of Evolution Strategies, described in Algorithm 2, on 51 Atari 2600 games available in OpenAI Gym [Brockman et al., 2016]. We used the same preprocessing and feedforward CNN architecture used by Mnih et al. [2016]. All games were trained for 1 billion frames, which requires about the same amount of neural network computation as the published 1-day results for A3C [Mnih et al., 2016] which uses 320 million frames. The difference is due to the fact that ES does not perform backpropagation and does not use a value function. By parallelizing the evaluation of perturbed parameters across 720 CPUs on Amazon EC2, we can bring down the time required for the training process to about one hour per game. After training, we compared final performance against the published A3C results and found that ES performed better in 23 games tested, while it performed worse in 28. The full results are in Table 2 in the supplementary material.

4.3 Parallelization

ES is particularly amenable to parallelization because of its low communication bandwidth requirement (Section 2.1). We implemented a distributed version of Algorithm 2 to investigate how ES scales with the number of workers. Our distributed implementation did not rely on special networking setup and was tested on public cloud computing service Amazon EC2.

We picked the 3D Humanoid walking task from OpenAI Gym [Brockman et al., 2016] as the test problem for our scaling experiment, because it is one of the most challenging continuous control problems solvable by state-of-the-art RL techniques, which require about a day to learn on modern hardware [Schulman et al., 2015, Duan et al., 2016a]. Solving 3D Humanoid with ES on one 18-core machine takes about 11 hours, which is on par with RL. However, when distributed across 80

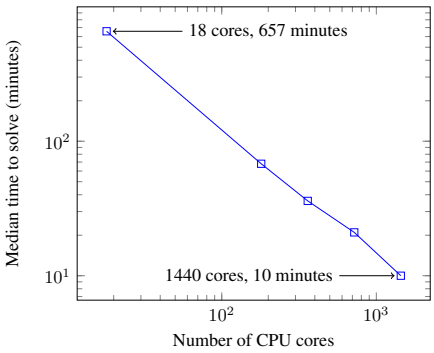


Figure 1: Time to reach a score of 6000 on 3D Humanoid with different number of CPU cores. Experiments are repeated 7 times and median time is reported.

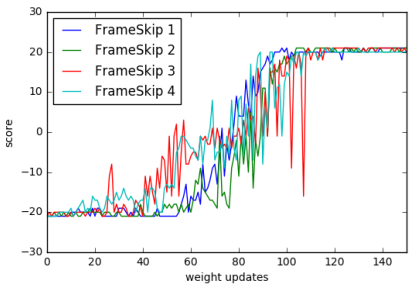


Figure 2: Learning curves for Pong using varying frame-skip parameters. Although performance is stochastic, each setting leads to about equally fast learning, with each run converging in around 100 weight updates.

machines and $1,440$ CPU cores, ES can solve 3D Humanoid in just 10 minutes, reducing experiment turnaround time by two orders of magnitude. Figure 1 shows that, for this task, ES is able to achieve linear speedup in the number of CPU cores.

4.4 Invariance to temporal resolution

It is common practice in RL to have the agent decide on its actions in a lower frequency than is used in the simulator that runs the environment. This action frequency, or *frame-skip*, is a crucial parameter in many RL algorithms [Braylan et al., 2005]. If the frame-skip is set too high, the agent cannot make its decisions at a fine enough timeframe to perform well in the environment. If, on the other hand, the frameskip is set too low, the effective time length of the episode increases too much, which deteriorates optimization performance as analyzed in section 3.1. An advantage of ES is that its gradient estimate is invariant to the length of the episode, which makes it much more robust to the action frequency. We demonstrate this by running the Atari game Pong using a frame skip parameter in $\{1, 2, 3, 4\}$. As can be seen in Figure 2, the learning curves for each setting indeed look very similar.

5 Related work

There have been many attempts at applying methods related to ES to train neural networks Risi and Togelius [2015]. For Atari, Hausknecht et al. [2014] obtain impressive results. Sehnke et al. [2010] proposed a method closely related the one investigated in our work. Koutník et al. [2013, 2010] and Srivastava et al. [2012] have similarly applied an an ES method to RL problems with visual inputs, but where the policy was compressed in a number of different ways. Natural evolution strategies has been successfully applied to black box optimization Wierstra et al. [2008, 2014], as well as for the training of the recurrent weights in recurrent neural networks Schmidhuber et al. [2007]. Stulp and Sigaud [2012] explored similar approaches to black box optimization. An interesting hybrid of black-box optimization and policy gradient methods was recently explored by Usunier et al. [2016]. Hyper-Neat Stanley et al. [2009] is an alternative approach to evolving both the weights of the neural networks and their parameters. Derivative free optimization methods have also been analyzed in the convex setting Duchi et al. [2015], Nesterov [2012].

The main contribution in our work is in showing that this class of algorithms is extremely scalable and efficient to use on distributed hardware. We have shown that ES, when carefully implemented, is competitive with competing RL algorithms in terms of performance on the hardest problems solvable today, and is surprisingly close in terms of data efficiency, while taking less wallclock time to train.

6 Conclusion

We have explored Evolution Strategies, a class of black-box optimization algorithms, as an alternative to popular MDP-based RL techniques such as Q-learning and policy gradients. Experiments on Atari and MuJoCo show that it is a viable option with some attractive features: it is invariant to action frequency and delayed rewards, and it does not need temporal discounting or value function approximation. Most importantly, ES is highly parallelizable, which allows us to make up for a decreased data efficiency by scaling to more parallel workers.

In future work, we plan to apply evolution strategies to those problems for which MDP-based reinforcement learning is less well-suited: problems with long time horizons and complicated reward structure. We are particularly interested in meta-learning, or learning-to-learn. A proof of concept for meta-learning in an RL setting was given by Duan et al. [2016b]: Using black-box optimization we hope to be able to extend these results. We also plan to examine combining ES with fast low precision neural network implementations to fully make use of the gradient-free nature of ES.

References

Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. *Space*, 1600:1800, 2005.

Dimo Brockhoff, Anne Auger, Nikolaus Hansen, Dirk V Arnold, and Tim Hohm. Mirrored sampling and sequential selection for evolution strategies. In *International Conference on Parallel Problem Solving from Nature*, pages 11–21. Springer, 2010.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016a.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016b.

John C Duchi, Michael I Jordan, Martin J Wainwright, and Andre Wibisono. Optimal rates for zero-order convex optimization: The power of two function evaluations. *IEEE Transactions on Information Theory*, 61(5):2788–2806, 2015.

John Geweke. Antithetic acceleration of monte carlo integration in bayesian inference. *Journal of Econometrics*, 38(1-2):73–89, 1988.

Tobias Glasmachers, Tom Schaul, and Jürgen Schmidhuber. A natural evolution strategy for multi-objective optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 627–636. Springer, 2010a.

Tobias Glasmachers, Tom Schaul, Sun Yi, Daan Wierstra, and Jürgen Schmidhuber. Exponential natural evolution strategies. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 393–400. ACM, 2010b.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

Kenji Kawaguchi. Deep learning without poor local minima. In *Advances In Neural Information Processing Systems*, pages 586–594, 2016.

Jan Koutník, Faustino Gomez, and Jürgen Schmidhuber. Evolving neural networks in compressed weight space. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 619–626. ACM, 2010.

Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

Yurii Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, pages 1–40, 2011.

Andrew Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX*, pages 363–372, 2006.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.

I. Rechenberg and M. Eigen. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog Stuttgart, 1973.

Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 2015.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2226–2234, 2016.

Tom Schaul, Tobias Glasmachers, and Jürgen Schmidhuber. High dimensions and heavy tails for natural evolution strategies. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 845–852. ACM, 2011.

Juergen Schmidhuber and Jieyu Zhao. Direct policy search and uncertain policy evaluation. In *Aaai spring symposium on search under uncertain and incomplete information, stanford univ*, pages 119–124, 1998.

Jürgen Schmidhuber, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. Training recurrent networks by evolino. *Neural computation*, 19(3):757–779, 2007.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

H.-P. Schwefel. *Numerische optimierung von computer-modellen mittels der evolutionsstrategie*. 1977.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

James C Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control*, 37(3):332–341, 1992.

Rupesh Kumar Srivastava, Jürgen Schmidhuber, and Faustino Gomez. Generalized compressed network search. In *International Conference on Parallel Problem Solving from Nature*, pages 337–346. Springer, 2012.

Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

Freek Stulp and Olivier Sigaud. Policy improvement methods: Between black-box optimization and episodic reinforcement learning. 2012.

Yi Sun, Daan Wierstra, Tom Schaul, and Juergen Schmidhuber. Efficient natural evolution strategies. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 539–546. ACM, 2009.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

Nicolas Usunier, Gabriel Synnaeve, Zeming Lin, and Soumith Chintala. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *arXiv preprint arXiv:1609.02993*, 2016.

Sjoerd van Steenkiste, Jan Koutník, Kurt Driessens, and Jürgen Schmidhuber. A wavelet-based encoding for neuroevolution. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 517–524. ACM, 2016.

Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15(1):949–980, 2014.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C Courville, Ruslan Salakhutdinov, Richard S Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *ICML*, volume 14, pages 77–81, 2015.

Sun Yi, Daan Wierstra, Tom Schaul, and Jürgen Schmidhuber. Stochastic search using the natural gradient. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1161–1168. ACM, 2009.

Game	DQN	A3C FF, 1 day	HyperNEAT	ES FF, 1 hour	A2C FF
Amidar	133.4	283.9	184.4	112.0	**548.2**
Assault	3332.3	**3746.1**	912.6	1673.9	2026.6
Asterix	124.5	**6723.0**	2340.0	1440.0	3779.7
Asteroids	697.1	**3009.4**	1694.0	1562.0	1733.4
Atlantis	76108.0	772392.0	61260.0	1267410.0	**2872644.8**
Bank Heist	176.3	**946.0**	214.0	225.0	724.1
Battle Zone	17560.0	11340.0	**36200.0**	16600.0	8406.2
Beam Rider	8672.4	**13235.9**	1412.8	744.0	4438.9
Berzerk		**1433.4**	1394.0	686.0	720.6
Bowling	41.2	36.2	**135.8**	30.0	28.9
Boxing	25.8	33.7	16.4	49.8	**95.8**
Breakout	303.9	**551.6**	2.8	9.5	368.5
Centipede	3773.1	3306.5	**25275.2**	7783.9	2773.3
Chopper Command	3046.0	**4669.0**	3960.0	3710.0	1700.0
Crazy Climber	50992.0	**101624.0**	0.0	26430.0	100034.4
Demon Attack	12835.2	**84997.5**	14620.0	1166.5	23657.7
Double Dunk	**21.6**	0.1	2.0	0.2	3.2
Enduro	**475.6**	82.2	93.6	95.0	0.0
Fishing Derby	2.3	13.6	**49.8**	49.0	33.9
Freeway	25.8	0.1	29.0	**31.0**	0.0
Frostbite	157.4	180.1	**2260.0**	370.0	266.6
Gopher	2731.8	**8442.8**	364.0	582.0	6266.2
Gravitar	216.5	269.5	370.0	**805.0**	256.2
Ice Hockey	3.8	4.7	**10.6**	4.1	4.9
Kangaroo	2696.0	106.0	800.0	**11200.0**	1357.6
Krull	3864.0	8066.6	**12601.4**	8647.2	6411.5
Montezuma's Revenge	50.0	**53.0**	0.0	0.0	0.0
Name This Game	5439.9	5614.0	**6742.0**	4503.0	5532.8
Phoenix		**28181.8**	1762.0	4041.0	14104.7
Pit Fall		**123.0**	0.0	0.0	8.2
Pong	16.2	11.4	17.4	**21.0**	20.8
Private Eye	298.2	194.4	**10747.4**	100.0	100.0
Q*Bert	4589.8	13752.3	695.0	147.5	**15758.6**
River Raid	4065.3	**10001.2**	2616.0	5009.0	9856.9
Road Runner	9264.0	31769.0	3220.0	16590.0	**33846.9**
Robotank	**58.5**	2.3	43.8	11.9	2.2
Seaquest	**2793.9**	2300.2	716.0	1390.0	1763.7
Skiing		13700.0	7983.6	**15442.5**	15245.8
Solaris		1884.8	160.0	2090.0	**2265.0**
Space Invaders	1449.7	**2214.7**	1251.0	678.5	951.9
Star Gunner	34081.0	**64393.0**	2720.0	1470.0	40065.6
Tennis	2.3	10.2	0.0	4.5	**11.2**
Time Pilot	5640.0	5825.0	**7340.0**	4970.0	4637.5
Tutankham	32.4	26.1	23.6	130.3	**194.3**
Up and Down	3311.3	54525.4	43734.0	67974.0	**75785.9**
Venture	54.0	19.0	0.0	**760.0**	0.0
Video Pinball	20228.1	**185852.6**	0.0	22834.8	46470.1
Wizard of Wor	246.0	**5278.0**	3360.0	3480.0	1587.5
Yars Revenge		7270.8	**24096.4**	16401.7	8963.5
Zaxxon	831.0	2659.0	3000.0	**6380.0**	5.6

Table 2: Final results obtained using Evolution Strategies on Atari 2600 games (feedforward CNN policy, deterministic policy evaluation, averaged over 10 re-runs with up to 30 random initial no-ops), and compared to results for DQN and A3C from Mnih et al. [2016] and HyperNEAT from Hausknecht et al. [2014]. A2C is our synchronous variant of A3C, and its reported scores are obtained with 320M training frames with the same evaluation setup as for the ES results. All methods were trained on raw pixel input.

Table 3: MuJoCo tasks: Ratio of ES timesteps to TRPO timesteps needed to reach various percentages of TRPO's learning progress at 5 million timesteps. These results were computed from ES learning curves averaged over 6 reruns.

Environment	% TRPO final score	TRPO score	TRPO timesteps	ES timesteps	ES timesteps / TRPO timesteps
HalfCheetah	25%	-1.35	9.05e+05	1.36e+05	0.15
	50%	793.55	1.70e+06	8.28e+05	0.49
	75%	1589.83	3.34e+06	1.42e+06	0.42
	100%	2385.79	5.00e+06	2.88e+06	0.58
Hopper	25%	877.45	7.29e+05	3.83e+05	0.53
	50%	1718.16	1.03e+06	3.73e+06	3.64
	75%	2561.11	1.59e+06	9.63e+06	6.05
	100%	3403.46	4.56e+06	3.16e+07	6.94
InvertedDoublePendulum	25%	2358.98	8.73e+05	3.98e+05	0.46
	50%	4609.68	9.65e+05	4.66e+05	0.48
	75%	6874.03	1.07e+06	5.30e+05	0.49
	100%	9104.07	4.39e+06	5.39e+06	1.23
InvertedPendulum	25%	276.59	2.21e+05	6.25e+04	0.28
	50%	519.15	2.73e+05	1.43e+05	0.52
	75%	753.17	3.25e+05	2.55e+05	0.78
	100%	1000.00	5.17e+05	4.55e+05	0.88
Swimmer	25%	41.97	1.04e+06	5.88e+05	0.56
	50%	70.73	1.82e+06	8.52e+05	0.47
	75%	99.68	2.33e+06	1.23e+06	0.53
	100%	128.25	4.59e+06	1.39e+06	0.30
Walker2d	25%	957.68	1.55e+06	6.43e+05	0.41
	50%	1916.48	2.27e+06	1.29e+07	5.69
	75%	2872.81	2.89e+06	2.31e+07	8.02
	100%	3830.03	4.81e+06	3.79e+07	7.88

NEAR-DATA PROCESSING FOR MACHINE LEARNING

Hyeokjun Choe, Seil Lee, Hyunha Nam, Seongsik Park, Seijoon Kim
Electrical and Computer Engineering
Seoul National University
Seoul, 08826, Republic of Korea
{genesis1104,lees231,godqhr825,pss015,hokiespa}@snu.ac.kr

Eui-Young Chung
Electrical and Electronic Engineering
Yonsei University
Seoul 03722, Republic of Korea
eychung@yonsei.ac.kr

Sungroh Yoon*
Electrical and Computer Engineering
Seoul National University
Seoul, 08826, Republic of Korea
sryoon@snu.ac.kr

ABSTRACT

In computer architecture, near-data processing (NDP) refers to augmenting the memory or the storage with processing power so that it can process the data stored therein. By offloading the computational burden of CPU and saving the need for transferring raw data in its entirety, NDP exhibits a great potential for acceleration and power reduction. Despite this potential, specific research activities on NDP have witnessed only limited success until recently, often owing to performance mismatches between logic and memory process technologies that put a limit on the processing capability of memory. Recently, there have been two major changes in the game, igniting the resurgence of NDP with renewed interest. The first is the success of machine learning (ML), which often demands a great deal of computation for training, requiring frequent transfers of big data. The second is the advent of NAND flash-based solid-state drives (SSDs) containing multicore processors that can accommodate extra computation for data processing. Sparked by these application needs and technological support, we evaluate the potential of NDP for ML using a new SSD platform that allows us to simulate in-storage processing (ISP) of ML workloads. Our platform (named ISP-ML) is a full-fledged simulator of a realistic multi-channel SSD that can execute various ML algorithms using the data stored in the SSD. For thorough performance analysis and in-depth comparison with alternatives, we focus on a specific algorithm: stochastic gradient decent (SGD), which is the de facto standard for training differentiable learning machines including deep neural networks. We implement and compare three variants of SGD (synchronous, Downpour, and elastic averaging) using ISP-ML, exploiting the multiple NAND channels for parallelizing SGD. In addition, we compare the performance of ISP and that of conventional in-host processing, revealing the advantages of ISP. Based on the advantages and limitations identified through our experiments, we further discuss directions for future research on ISP for accelerating ML.

1 INTRODUCTION

Recent successes in deep learning can be accredited to the availability of big data that has made the training of large deep neural networks possible. In the conventional memory hierarchy, the training data stored at the low level (e.g., hard disks) need to be moved upward all the way to the CPU registers. As larger and larger data are being used for training large-scale models such as deep networks (LeCun et al., 2015), the overhead incurred by the data movement in the hierarchy becomes more salient, critically affecting the overall computational efficiency and power consumption.

*To whom correspondence should be addressed.

The idea of near-data processing (NDP) (Balasubramonian et al., 2014) is to equip the memory or storage with intelligence (i.e., processors) and let it process the data stored therein firsthand. A successful NDP implementation would reduce the data transfers and power consumption, not to mention offloading the computational burden of CPUs. The types of NDP realizations include processing in memory (PIM) (Gokhale et al., 1995) and in-storage processing (ISP) (Acharya et al., 1998; Kim et al., 2016c; Lee et al., 2016; Choi & Kee, 2015). Despite the potential of NDP, it has not been considered significantly for commercial systems. For PIM, there has been a wide performance gap between the separate processes to manufacture logic and memory chips. For ISP, commercial hard disk drives (HDDs), the mainstream storage devices for a long time, normally have limited processing capabilities due to tight selling prices.

Recently, we have seen a resurrection of NDP with renewed interest, which has been triggered by two major factors, one in the application side and the other in the technology side: First, computing- and data-intensive deep learning is rapidly becoming the method of choice for various machine learning tasks. To train deep neural networks, a large volume of data is typically needed to ensure performance. Although GPUs and multicore CPUs often provide an effective means for massive computation required by deep learning, it remains inevitable to store big training data in the storage and then transfer them to the CPU/GPU level for computation. Second, NAND flash-based solid-state drives (SSDs) are becoming popular, gradually replacing HDDs in various computing sectors. To interface SSDs with the host seamlessly replacing HDDs, SSDs require various software running inside, e.g., for address translation and garbage collection (Kim et al., 2002; Gupta et al., 2009). To suit such needs, SSDs are often equipped with multicore processors, which provide far more processing capabilities than those in HDDs. Usually, there exists a plenty of idle time in the processors in SSDs that can be exploited for other purposes than SSD housekeeping (Kim et al., 2010; 2016b).

Motivated by these changes and opportunities, we propose a new SSD platform that allows us to simulate in-storage processing (ISP) of machine learning workloads and evaluate the potential of NDP for machine learning in ISP. Our platform named ISP-ML is a full-fledged system-level simulator of a realistic multi-channel SSD that can execute various machine learning algorithms using the data stored in the SSD. For thorough performance analysis and in-depth comparison with alternatives, we focus on describing our implementation of a specific algorithm in this paper: the stochastic gradient decent (SGD) algorithm, which is the *de facto* standard for training differentiable learning machines including deep neural networks. Specifically, we implement three types of parallel SGD: synchronous SGD (Zinkevich et al., 2010), Downpour SGD (Dean et al., 2012), and elastic averaging SGD (EASGD) (Zhang et al., 2015). We compare the performance of these implementations of parallel SGD using a 10 times amplified version of MNIST (LeCun et al., 1998). Furthermore, to evaluate the effectiveness of ISP-based optimization by SGD, we compare the performance of ISP-based and the conventional in-host processing (IHP)-based optimization.

To the best of the authors' knowledge, this work is one of the first attempts to apply NDP to a multi-channel SSD for accelerating SGD-based optimization for training differentiable learning machines. Our specific contributions can be stated as follows:

- We created a full-fledged ISP-supporting SSD platform called ISP-ML, which required multi-year team efforts. ISP-ML is versatile and can simulate not only storage-related functionalities of a multi-channel SSD but also NDP-related functionalities in realistic manner. ISP-ML can execute various machine learning algorithms using the data stored in the SSD while supporting the simulation of multi-channel NAND flash SSDs to exploit data-level parallelism.

- We thoroughly tested the effectiveness of our platform by implementing and comparing multiple versions of parallel SGD, which is widely used for training various machine learning algorithms including deep learning. We also devised a methodology that can carefully and fairly compare the performance of IHP-based and ISP-based optimization.

- We identified intriguing future research opportunities in terms of exploiting the parallelism provided by the multiple NAND channels inside SSDs. As in high-performance computing, there exist multiple "nodes" (i.e., NAND channel controllers) for sharing workloads, but the communication cost is negligible (due to negligible-latency on-chip communication) unlike the conventional parallel computing. Using our platform, we envision new designs of parallel optimization and training algorithms that can exploit this characteristic, producing enhanced results.

2 BACKGROUND AND RELATED WORK

2.1 MACHINE LEARNING AS AN OPTIMIZATION PROBLEM

Various types of machine learning algorithms exist (Murphy, 2012; Goodfellow et al., 2016), and their core concept can often be explained using the following equations:

$$F(D, \theta) = L(D, \theta) + r(\theta) \tag{1}$$

$$\theta_{t+1} = \theta_t + \Delta\theta(D) \tag{2}$$

$$\Delta\theta(D) = -\eta \nabla F(D, \theta) \tag{3}$$

where D and θ denote the input data and model parameters, respectively, and a loss function $L(D, \theta)$ reflects the difference between the optimal and current hypotheses. A regularizer to handle overfitting is denoted by $r(\theta)$, and the objective function $F(D, \theta)$ is the sum of the loss and regularizer terms. The main purpose of supervised machine learning can then be formulated as finding optimal θ that minimizes $F(D, \theta)$. Gradient descent is a first-order iterative optimization algorithm to find the minimum value of $F(D, \theta)$ by updating θ on every iteration t to the direction of negative gradient of $F(D, \theta)$, where η is the learning rate. SGD computes the gradient of the parameters and updates them using a single training sample per iteration. Minibatch (stochastic) gradient decent uses multiple (but far less than the whole) samples per iteration. As will be explained shortly, we employ minibatch SGD in our framework, setting the size of a minibatch to the number of training samples in a NAND flash page, which is named 'page-minibatch' (see Figure 2).

2.2 PARALLEL AND DISTRIBUTED SGD

Zinkevich et al. (2010) proposed an algorithm that implements parallel SGD in a distributed computing setup. This algorithm often suffers from excessive latency caused by the need for synchronization of all slave nodes. To overcome this weakness, Recht et al. (2011) proposed the lock-free Hogwild! algorithm that can update parameters asynchronously. Hogwild! is normally implemented in a single machine with a multicore processor. Dean et al. (2012) proposed the Downpour SGD for a distributed computing systems by extending the Hodwild! algorithm. While they successfully implemented asynchronous SGD in a distributed computing system, it often fails to overcome communication bottlenecks and shows inefficient bandwidth usage, caused by substantial data movements between computing nodes. Recently proposed EASGD (Zhang et al., 2015) attempted to minimize communication overhead by reducing the frequency of parameter updates. Many EASGD-based approaches reported its effectiveness in distributed environments.

2.3 FUNDAMENTALS OF SOLID-STATE DRIVES (SSDS)

SSDs have emerged as a type of next-generation storage device using NAND flash memory (Kim et al., 2010). As shown in the right image in Figure 1(a), a typical SSD consists of an SSD controller, a DRAM buffer, and a NAND flash array. The SSD controller is typically composed of an embedded processor, a cache controller, and channel controllers. The DRAM component, controlled by the cache controller, plays the role of a cache buffer when the NAND flash array is read or written. The NAND flash array contains multiple NAND chips that can be accessed simultaneously thanks to multi-channel configurations and per-channel controllers. Every channel controller is managed by the software called flash translation layer (FTL), which executes wear-leveling and garbage collection to improve the performance and durability of the NAND flash array.

2.4 PREVIOUS WORK ON NEAR-DATA PROCESSING

Most of the previous work on ISP focused on popular but inherently simple algorithms, such as scan, join, and query operations (Kim et al., 2016c). Lee et al. (2016) proposed to run the merge operation (frequently used by external sort operation in Hadoop) inside an SSD to reduce IO transfers and read/write operations, also extending the lifetime of the NAND flash inside the SSD. Choi & Kee (2015) implemented algorithms for linear regression, k-means, and string match in the flash memory controller (FMC) via reconfigurable stream processors. In addition, they implemented a MapReduce application inside the embedded processor and FMC of the SSD by using partitioning and pipelining methods that could improve performance and reduce power consumption. BlueDBM (Jun

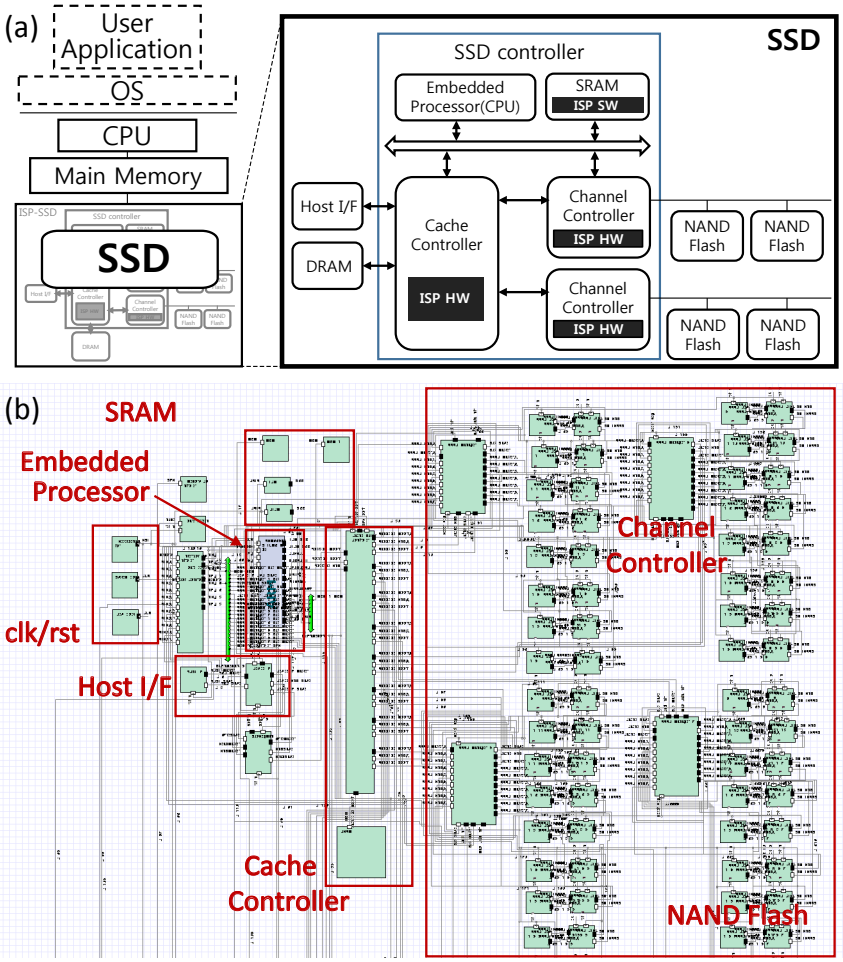


Figure 1: (a) Block diagram of a typical computing system equipped with an SSD and a magnified view of a usual SSD depicting its internal components and their connections. (b) Schematic of the proposed ISP-ML framework, which is implemented in SystemC using Synopsys Platform Architect (http://www.synopsys.com).

et al., 2015) is an ISP system architecture for distributed computing systems with a flash memory-based embedded field programmable gate array (FPGA). The authors implemented nearest-neighbor search, graph traversal, and string search algorithms. No prior work ever implemented and evaluated SSD-based optimization of machine learning algorithms using SGD.

3 Proposed Methodology

Figure 1(a) shows the block diagram of a typical computing system, which is assumed to have an SSD as its storage device. Also shown in the figure is a magnified view of the SSD block diagram that shows the major components of an SSD and their interconnections. Starting from the baseline SSD depicted above, we can implement ISP functionalities by modifying the components marked with black boxes (i.e., ISP HW and ISP SW in the figure). Figure 1(b) shows the detailed schematic of our proposed ISP-ML platform that corresponds to the SSD block (with ISP components) shown in Figure 1(a).

In this section, we provide more details of our ISP-ML framework. In addition, we propose a performance comparison methodology that can compare the performance of ISP and the conventional IHP in a fair manner. As a specific example of the ML algorithms that can be implemented in ISP-ML, we utilize parallel SGD.

3.1 ISP-ML: ISP PLATFORM FOR MACHINE LEARNING ON SSDS

Our ISP-ML is a system-level simulator implemented in SystemC on the Synopsys Platform Architect environment (http://www.synopsys.com). ISP-ML can simulate hardware and software ISP components marked in Figure 1(b) simultaneously. This integrative functionality is crucial for design space exploration in SSD developments. Moreover, ISP-ML allows us to execute various machine learning algorithms described in high-level languages (C or C++) directly on ISP-ML only with minor modifications.

At the conception of this research, we could not find any publicly available SSD simulator that could be modified for implementing ISP functionalities. This motivated us to implement a new simulator. There exist multiple ways of realizing the idea of ISP in an SSD. The first option would be to use the embedded core inside the SSD controller (Figure 1(a)). This option does not require designing a new hardware logic and is also flexible, since the ISP capability is implemented by software. However, this option is not ideal for exploiting hardware acceleration and parallelization. The second option would be to design dedicated hardware logics (such as those boxes with black marks in Figure 1(a) and the entire Figure 1(b)) and integrate them into the SSD controller. Although significantly more efforts are needed for this option compared the first, we chose this second option due to its long-term advantages provided by hardware acceleration and power reduction.

Specifically, we implemented two types of ISP hardware components, in addition to the software components. First, we let each channel controller not only manage read/write operations to/from its NAND flash channel (as in the usual SSDs) but also perform primitive operations on the data stored in its NAND channel. The type of primitive operation performed depends on the machine learning algorithm used (the next subsection explains more details of such operations for SGD). Additionally, each channel controller in ISP-ML (slave) communicates with the cache controller (master) in a master-slave architecture. Second, we designed the cache controller so that it can collect the outcomes from each of the channel controllers, in addition to its inherent functionality as a cache (DRAM) manager inside the SSD controller. This master-slave architecture can be interpreted as a tiny-scale version of the master-slave architecture commonly used in distributed systems. Just as the channel controllers, the exact functionality of the cache controller can be optimized depending on the specific algorithm used. Both the channel controllers and the cache controller have internal memory, but the memory size in the latter is far greater than that in the former.

Specific parameters and considerations used in our implementation can be found in Section 4.1. There are a few points worth mentioning. Unlike existing conventional SSD simulators, the baseline SSD implemented in ISP-ML can store data in the NAND flash memory inside. In order to support reasonable simulation speed, we modeled ISP-ML at cycle-accurate transaction level while minimizing negative impact on accuracy. We omit to describe other minor details of hardware logic implementations, as are beyond the scope of the conference.

3.2 PARALLEL SGD IMPLEMENTATION ON ISP-ML

Using our ISP-ML platform, we implemented the three types of parallel SGD algorithms outlined in Figure 2: synchronous SGD (Zinkevich et al., 2010), Downpour SGD (Dean et al., 2012), and EASGD (Zhang et al., 2015). For brevity, we focus on describing the implementation details of these algorithms in ISP-ML and omit the purely algorithmic details of each algorithm; we refer the interested to the corresponding references. Note that the size of a minibatch for the minibatch SGD in our framework is set to the number of training samples in a NAND flash page (referred to as 'page-minibatch' in Figure 2).

For implementing synchronous SGD, we let each of the n channel controllers synchronously compute the gradient. Firstly, each channel controller reads page-sized data from the NAND flash memory and then stores the data in the channel controller's buffer. Secondly, the channel controller pulls the cache controller's parameters (θ_{cache}) and stores them in the buffer. Using the data and parameters stored in the buffer, each channel controller calculates the gradient in parallel. After transferring the gradient to the cache controller, the channel controllers wait for a signal from the cache controller. The cache controller aggregates and updates the parameters and then sends the channel controller signals to pull and replicate the parameters.

We implemented Downpour SGD in a similar way to implementing synchronous SGD; the major difference is that each channel controller immediately begins the next iteration after transferring the

Synchronous SGD Processing by *i*-th channel controller and cache controller	**Downpour SGD** Processing by *i*-th channel controller and cache controller	**EASGD** Processing by *i*-th channel controller and cache controller

Repeat
Read a page from NAND
pull θ_{cache}
$\theta^i = \theta_{cache}$
$\Delta\theta^i = 0$
Repeat for page-minibatch
$\theta^i = \theta^i - \eta \nabla_t^i(\theta)$
$\Delta\theta^i = \Delta\theta^i + \eta \nabla_t^i(\theta)$
t++
end
push $\Delta\theta^i$ and wait
sync.
$\theta_{cache} = \theta_{cache} - 1/n \cdot \sum \Delta\theta^i$
end
	Repeat
	Read a page from NAND
	pull θ_{cache}
	$\theta^i = \theta_{cache}$
	$\Delta\theta^i = 0$
	Repeat for page-minibatch
	$\theta^i = \theta^i - \eta \nabla_t^i(\theta)$
	$\Delta\theta^i = \Delta\theta^i + \eta \nabla_t^i(\theta)$
	t++
	end
	if(τ devides t) **then**
	push $\Delta\theta^i$
	$\theta_{cache} = \theta_{cache} - \Delta\theta^i$
	end
	end
		Repeat
		Read a page from NAND
		Repeat for page-minibatch
		$\theta^i = \theta^i - \eta \nabla_t^i(\theta)$
		t++
		end
		if(τ devides t) **then**
		pull θ_{cache}
		$\theta^i = \theta - \alpha(\theta^i - \theta_{cache})$
		push $(\theta^i - \theta_{cache})$
		$\theta_{cache} = \theta_{cache} + \alpha(\theta^i - \theta_{cache})$
		end
		end

Figure 2: Pseudo-code of the three SGD algorithms implemented in ISP-ML: synchronous SGD (Zinkevich et al., 2010), Downpour SGD (Dean et al., 2012), and EASGD (Zhang et al., 2015). The shaded line indicates the computation occurring in the cache controller (master); the other lines are executed in the channel controllers (slaves). Note that the term 'page-minibatch' refers to the minibatch SGD used in our framework, where the size of a minibatch is set to the number of training samples in a NAND flash page.

gradient to cache controller. The cache controller updates the parameters with the gradient from the channel controllers sequentially.

For EASGD, we let each of the channel controllers have its own SGD parameters unlike synchronous SGD and Downpour SGD. Each channel controller pulls the parameters from the cache controller after computing the gradient and updating its own parameters. Each channel controller calculates the differences between its own parameters and the cache controller's parameters and then pushes the differences to the cache controller.

Of note is that, besides its widespread use, SGD has some appealing characteristics that facilitate hardware implementations. We can implement parallel SGD on top of the master-slave architecture realized by the cache controller and the channel controllers. We can also take advantage of effective techniques developed in the distributed and parallel computation domain. Importantly, each SGD iteration is so simple that it can be implemented without incurring excessive hardware overhead.

3.3 Methodology for IHP-ISP Performance Comparison

To evaluate the effectiveness of ISP, it is crucial to accurately and fairly compare the performances of ISP and the conventional IHP. However, performing this type of comparison is not trivial (see Section 4.3 for additional discussion). Furthermore, the accurate modeling of commercial SSDs equipped with ISP-ML is impossible due to lack of information about commercial SSDs (e.g., there is no public information on the FTL and internal architectures of any commercial SSD). Therefore, we propose a practical methodology for accurate comparison of IHP and ISP performances, as depicted in Figure 3. Note that this comparison methodology is applicable not only to the parallel SGD implementations explained above but also to other ML algorithms that can be executed in ISP-ML.

In the proposed comparison methodology, we focus on the data IO latency time of the storage (denoted as T_{IO}), since it is the most critical factor among those that affect the execution time of IHP. The total processing time of IHP ($\mathrm{IHP}_{\mathrm{time}}$ or T_{total}) can then be divided into the data IO time and the non-data IO time (T_{nonIO}) as follows:

$$\mathrm{IHP}_{\mathrm{time}} = T_{\mathrm{total}} = T_{\mathrm{nonIO}} + T_{\mathrm{IO}}. \tag{4}$$

To calculate the expected IHP simulation time adjusted to ISP-ML, the data IO time of IHP is replaced by the data IO time of the baseline SSD in ISP-ML (T_{IOsim}). By using Eq. (4), the expected IHP simulation time can then be represented by

$$\text{Expected IHP simulation time} = T_{\mathrm{nonIO}} + T_{\mathrm{IOsim}} = T_{\mathrm{total}} - T_{\mathrm{IO}} + T_{\mathrm{IOsim}}. \tag{5}$$

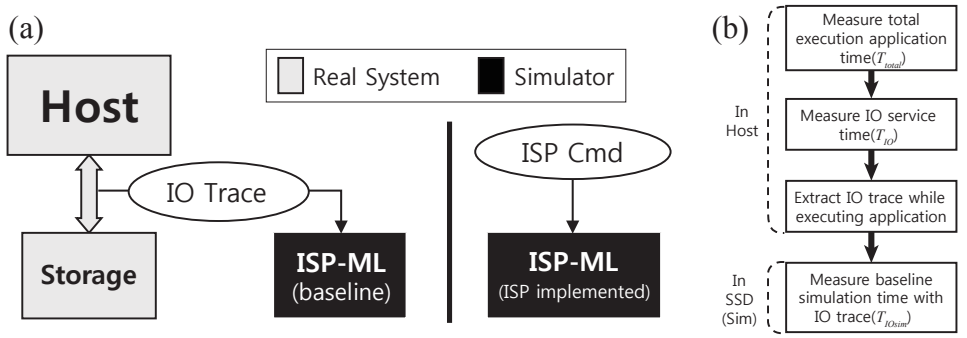


Figure 3: (a) Overview of our methdology to compare the performance of in-host processing (IHP) and in-storage processing (ISP). (b) Details of our IHP-ISP comparison flow.

The overall flow of the proposed comparison methodology is depicted in Figure 3(b). First, the total processing time (T_{total}) and the data IO time of storage (T_{IO}) are measured in IHP, extracting the IO trace of storage during an application execution. The simulation IO time (T_{IOsim}) is then measured using the IO trace (extracted from IHP) on the baseline SSD of ISP-ML. Finally, the expected IHP simulation time is calculated by plugging the total processing time (T_{total}), the data IO time of storage (T_{IO}) and the simulation IO time (T_{IOsim}) into Eq. (5). With the proposed method and ISP-ML, which is applicable to a variety of IHP environments regardless of the type of storage used, it is possible to quickly and easily compare performances of various ISP implementations and IHP in a simulation environment.

4 Experimental Results

4.1 Setup

All the experiments presented in this section were run on a machine equipped with an 8-core Intel(R) Core i7-3770K CPU (3.50GHz) with DDR3 32GB RAM, Samsung SSD 840 Pro, and Ubuntu 14.04 LTS (kernel version: 3.19.0-26-generic). We used ARM 926EJ-S (400MHz) as the embedded processor inside ISP-ML and DFTL (Gupta et al., 2009) as the FTL of ISP-ML. The simulation model we used was derived from a commercial product (Micron NAND MT29F8G08ABACA) and had the following specifications: page size = 8KB, $t_{\text{prog}} = 300\mu$s, $t_{\text{read}} = 75\mu$s, and $t_{\text{block erase}} = 5ms$.[1] Each channel controller had 24KB of memory [8KB (page size) for data and 16KB for ISP] and a floating point unit (FPU) having 0.5 instruction/cycle performance (with pipelining). The cache controller had memory of $(n+1)\times$ 8KB (page size), where n is the number of channels ($n = 4, 8, 16$). Depending on the algorithm running in ISP-ML, we can adjust these parameters.

Note that the main purpose of our experiments in this paper was to verify the functionality of our ISP-ML framework and to evaluate the effectiveness of ISP over the conventional IHP using SGD, even though our framework is certainly not limited only to SGD. To this end, we selected logistic regression, a fundamental ML algorithm that can directly show the advantage of ISP-based optimizations over IHP-based optimizations without unnecessary complications. We thus implemented the logistic regression algorithm as a single-layer perceptron (with cross entropy loss) in SystemC and uploaded it to ISP-ML. As stated in Section 5.3, our future work includes the implementation and testing of more complicated models (such as deep neural networks) by reflecting the improvement opportunities revealed from the experiments presented in this paper.

As test data, we utilized the samples from the MNIST database (LeCun et al., 1998). To amplify the number of training samples (for showing the scalability of our approach), we used elastic distortion (Simard et al., 2003), producing 10 times more data than the original MNIST (approximately 600,000 training and 10,000 test samples were used in total). To focus on the performance evaluation of running ISP operations, we preloaded our NAND flash simulation model with the simulation

[1]These are conservative settings, compared with those of the original commercial product; using the specifications of a commercial product will thus improve the performance of ISP-ML.

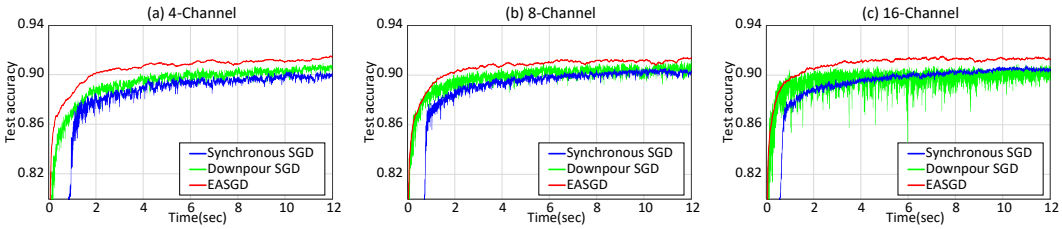


Figure 4: Test accuracy of three ISP-based SGD algorithms versus wall-clock time with a varying number of NAND flash channels: (a) 4 channels, (b) 8 channels, and (c) 16 channels.

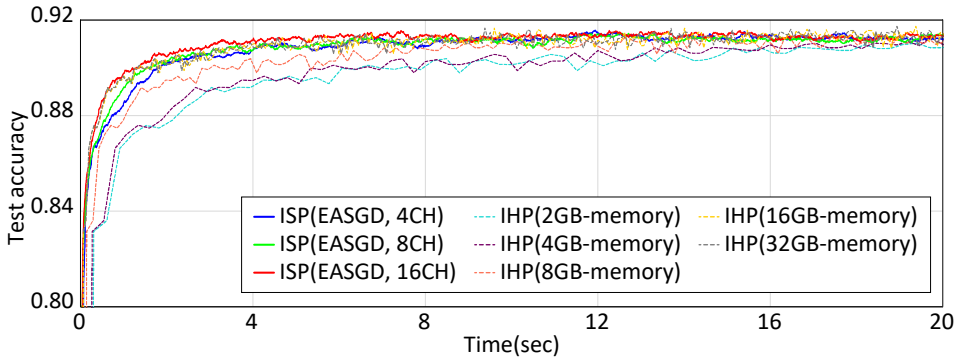


Figure 5: Test accuracy of ISP-based EASGD in the 4, 8, and 16 channel configurations and IHP-based minibatch SGD using diverse memory sizes.

data (the same condition was used for the alternatives for fairness). Based on the size of a training sample this dataset and the size of a NAND page (8KB), we set the size of each minibatch to 10.

4.2 Performance Comparison: ISP-Based Optimization

As previously explained, to identify which SGD algorithm would be best suited for use in ISP, we implemented and analyzed three types of SGD algorithms: synchronous SGD, Downpour SGD, and EASGD. For EASGD, we set the moving rate (α) and the communication period (τ) to 0.001 and 1, respectively. For a fair comparison, we chose different learning rates for different algorithms that gave the best performance for each algorithm. Figure 4 shows the test accuracy of three algorithms with varying numbers of channels (4, 8, and 16) with respect to wall-clock time.

As shown in Figure 4, using EASGD gave the best convergence speed in all of the cases tested. EASGD outperformed synchronous and Downpour SGD by factors of 5.24 and 1.96 on average, respectively. Synchronous SGD showed a slower convergence speed when compared to Downpour SGD because it could not start learning on the next set of minibatch until the results of all the channel controllers reported to the cache controller. Moreover, one delayed worker could halt the entire process. This result suggests that EASGD is adequate for all the channel configurations tested in that ISP can benefit from ultra-fast on-chip level communication and employ application-specific hardware that can eliminate any interruptions from other processors.

4.3 Performance comparison: IHP versus ISP

In large-scale machine learning, the computing systems used may suffer from memory shortage, which incurs significant data swapping overhead. In this regard, ISP can provide an effective solution that can potentially reduce data transfer penalty by processing core operations at the storage level.

In this context, we carried out additional experiments to compare the performance of IHP-based and ISP-based EASGD. We tested the effectiveness of ISP in a memory shortage situation with 5 different configurations of IHP memory: 2GB, 4GB, 8GB, 16GB, and 32GB. We assumed that the host already loaded all of the data to the main memory for IHP. This assumption is realistic because

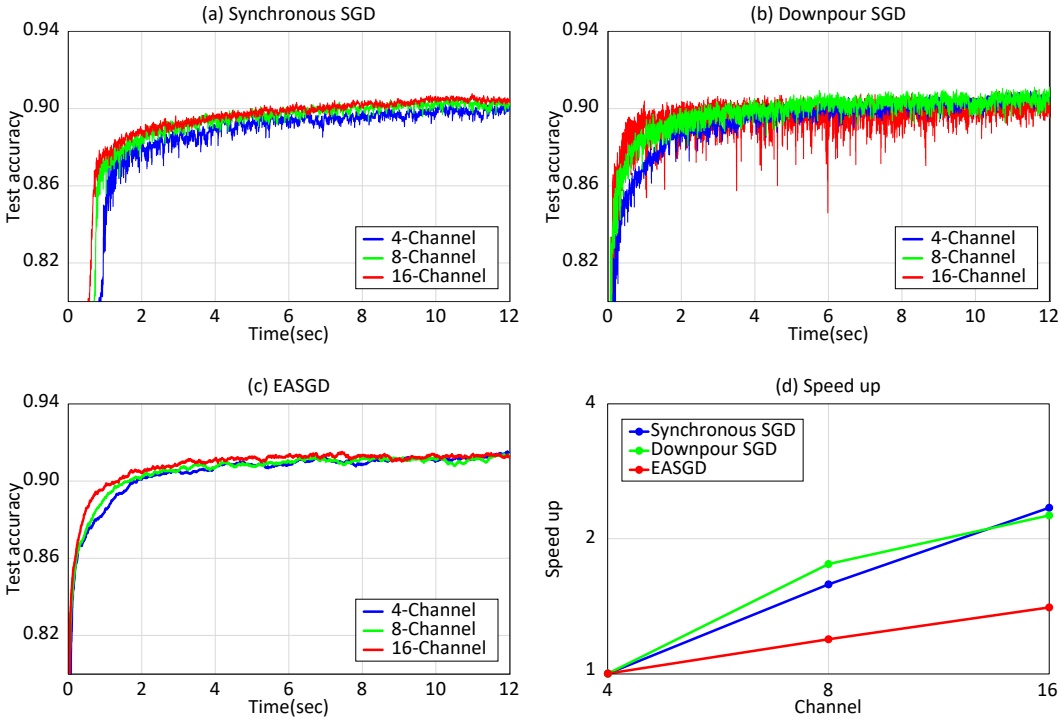


Figure 6: Test accuracy of different ISP-based SGD algorithms for a varied number of channels: (a) synchronous SGD, (b) Downpour SGD, and (c) EASGD. (d) Training speed-up for the three SGD algorithms for a various number of channels.

state-of-the-art machine learning techniques often employ a prefetch strategy to hide the initial data transfer latency.

As depicted in Figure 5, ISP-based EASGD with 16 channels gave the best performance in our experiments. The convergence speed of the IHP-based optimization slowed down, in accordance with the reduced memory size. The results with 16GB and 32GB of memory gave similar results because using 16GB of memory was enough to load and allocate most of the resource required by the process. As a result, ISP was more efficient when memory was insufficient, as would be often the case with large-scale datasets in practice.

4.4 Channel Parallelism

To closely examine the effect of exploiting data-level parallelism on performance, we compared the accuracy of the three SGD algorithms, varying the number of channels (4, 8, and 16), as shown in Figure 6. All the three algorithms resulted in convergence speed-up by using more channels; synchronous SGD achieved $1.48\times$ speed-up when the number of channels increased from 8 to 16. From Figure 6(d), we can also note that the convergence speed-up tends to be proportional to number of channels. These results suggest that the communication overhead in ISP is negligible, and that ISP does not suffer from the communication bottleneck that commonly occurs in distributed computing systems.

4.5 Effects of Communication Period in Asynchronous SGD

Finally, we investigated how changes in the communication period (i.e., how often data exchange occurs during distributed optimization) affect SGD performance in the ISP environment. Figure 7 shows the test accuracy of the Downpour SGD and EASGD algorithms versus wall-clock time when we varied their communication periods. As described in Zhang et al. (2015), Downpour SGD normally achieved a high performance for a low communication period [$\tau = 1, 4$] and became unstable for a high communication period [$\tau = 16, 64$] in ISP. Interestingly, in contrast to the conventional

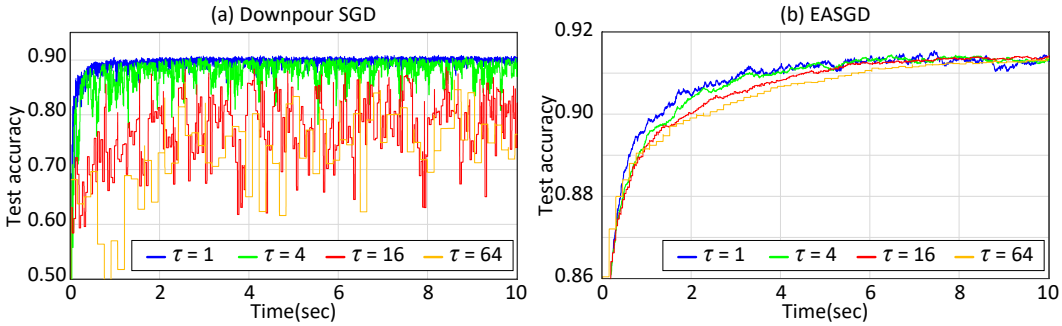


Figure 7: Test accuracy of ISP-based Downpour SGD and EASGD algorithms versus wall-clock time for different communication periods.

distributed computing system setting, the performance of EASGD decreased as the communication period increased in the ISP setting. This is because the on-chip communication overhead in ISP is significantly lower than that in the distributed computing system. As a result, there would be no need for extending the communication period to reduce communication overhead in the ISP environment.

5 Discussion

5.1 Parallelism in ISP

Given the advances in underlying hardware and semiconductor technology, ISP can provide various advantages for data processing involved in machine learning. For example, our ISP-ML could minimize (practically eliminate) the communication overheads between parallel nodes leveraged by ultra-fast on-chip communication inside an SSD. Minimizing communication overheads can improve various key aspects of data-processing systems, such as energy efficiency, data management, security, and reliability. By exploiting this advantage of fast on-chip communications in ISP, we envision that we will be able to devise a new kind of parallel algorithms for optimization and machine learning running on ISP-based SSDs.

Our experiment results also revealed that a high degree of parallelism could be achieved by increasing the number of channels inside an SSD. Some of the currently available commercial SSDs have as many as 16 channels. Given that the commercial ISP-supporting SSDs would (at least initially) be targeted at high-end SSD markets with many NAND flash channels, our approach is expected to add a valuable functionality to such SSDs. Unless carefully optimized, a conventional distributed system will see diminishing returns as the number of nodes increases, due to the increased communication overhead and other factors. Exploiting a hierarchy of parallelism (i.e., parallel computing nodes, each of which has ISP-based SSDs with parallelism inside) may provide an effective acceleration scheme, although a fair amount of additional research is needed before we can realize this idea.

5.2 ISP-IHP Comparison Methodology

To fairly compare the performances of ISP and IHP, it would be ideal to implement ISP-ML in a real semiconductor chip, or to simulate IHP in the ISP-ML framework. Selecting either option, however, is possible but not plausible (at least in academia), because of high cost of manufacturing a chip, and the prohibitively high simulation time for simulating IHP in the Synopsys Platform Architect environment (we would have to implement many components of a modern computer system in order to simulate IHP). Another option would be to implement both ISP and IHP using FPGAs, but it will take another round of significant efforts for developments.

To overcome these challenges (still assuring a fair comparison between ISP and IHP), we have proposed the comparison methodology described in Section 3.3. In terms of measuring the absolute running time, our methodology may not be ideal. However, in terms of highlighting relative performance between alternatives, our method should provide a satisfactory solution.

Our comparison methodology extracts IO trace from the storage while executing an application in the host, which is used for measuring simulation IO time in the baseline SSD in ISP-ML. In this procedure, we assume that the non-IO time of IHP is consistent regardless of the kind of storage the host has. The validity of this assumption is warranted by the fact that the amount of non-IO time changed by the storage is usually negligible compared with the total execution time or IO time.

5.3 Opportunities for Future Research

In this paper we focused on the implementation and testing of ISP-based SGD as a proof of concept. The simplicity and popularity of (parallel) SGD underlie our choice. By design, it is possible to run other algorithms in our ISP-ML framework immediately; recall that our framework includes a general-purpose ARM processor that can run executables compiled from C/C++ code. However, it would be meaningless just to have an ISP-based implementation, if its performance is unsatisfactory. To unleash the full power of ISP, we need additional ISP-specific optimization efforts, as is typically the case with hardware design.

With this in mind, we have started implementing deep neural networks (with realistic numbers of layers and hyperparameters) using our ISP-ML framework. Especially, we are carefully devising a way of balancing the memory usage in the DRAM buffer, the cache controller, and the channel controllers inside ISP-ML. It would be reasonable to see an SSD with a DRAM cache with a few gigabytes of memory, whereas it is unrealistic to design a channel controller with that much memory. Given that a large amount of memory is needed only to store the parameters of such deep models, and that IHP and ISP have different advantage and disadvantages, it would be intriguing to investigate how to make IHP and ISP can cooperate to enhance the overall performance. For instance, we can let ISP-based SSDs perform low-level data-dependent tasks while assigning high-level tasks to the host, expanding the current roles of the cache controller and the channel controllers inside ISP-ML to the whole system level.

Our future work also includes the following: First, we will be able to implement adaptive optimization algorithms such as Adagrad (Duchi et al., 2011) and Adadelta (Zeiler, 2012). Second, precomputing meta-data during data writes (instead of data reads) could provide another direction of research that can bring even more speedup. Third, we will be able to implement data shuffle functionality in order to maximize the effect of data-level parallelism. Currently, ISP-ML arbitrarily splits the input data into its multi-channel NAND flash array. Fourth, we may investigate the effect of NAND flash design on performance, such as the NAND flash page size. Typically, the size of a NAND flash page significantly affects the performance of SSDs, given that the page size (e.g., 8KB) is the basic unit of NAND operation (read and write). In case where the size of a single example often exceeds the page size, frequent data fragmentation is inevitable, eventually affecting the overall performance. The effectiveness of using multiple page sizes was already reported for conventional SSDs (Kim et al., 2016a), and we may borrow this idea to further optimize ISP-ML.

Acknowledgments

The authors would like to thank Byunghan Lee at Data Science Laboratory, Seoul National University for proofreading the manuscript. This work was supported in part by BK21 Plus (Electrical and Computer Engineering, Seoul National University) in 2016, in part by a grant from SK Hynix, and in part by a grant from Samsung Electronics.

References

Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *ACM SIGOPS Operating Systems Review*, volume 32, pp. 81–91. ACM, 1998.

Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. *Micro, IEEE*, 34(4):36–42, 2014.

I. Stephen Choi and Yang-Suk Kee. Energy efficient scale-in clusters with in-storage processing for big-data analytics. In *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 265–273. ACM, 2015.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL `http://www.deeplearningbook.org`.

Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: an appliance for big data analytics. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 1–13. IEEE, 2015.

Dong Kim, Kwanhu Bang, Seung-Hwan Ha, Sungroh Yoon, and Eui-Young Chung. Architecture exploration of high-performance pcs with a solid-state disk. *IEEE Transactions on Computers*, 59(7):878–890, 2010.

Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2): 366–375, 2002.

Jin-Young Kim, Sang-Hoon Park, Hyeokjun Seo, Ki-Whan Song, Sungroh Yoon, and Eui-Young Chung. Nand flash memory with multiple page sizes for high-performance storage devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):764–768, 2016a.

Jin-Young Kim, Tae-Hee You, Sang-Hoon Park, Hyeokjun Seo, Sungroh Yoon, and Eui-Young Chung. An effective pre-store/pre-load method exploiting intra-request idle time of nand flash-based storage devices. *under review*, 2016b.

Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Information Sciences*, 327:183–200, 2016c.

Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

Young-Sik Lee, Luis Cavazos Quero, Sang-Hoon Kim, Jin-Soo Kim, and Seungryoul Maeng. Activesort: Efficient external sorting using active ssds in the mapreduce framework. *Future Generation Computer Systems*, 2016.

Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

Patrice Y Simard, David Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pp. 958–962, 2003.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pp. 685–693, 2015.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pp. 2595–2603, 2010.

2015 International Conference on Parallel Architecture and Compilation

Practical Near-Data Processing for In-memory Analytics Frameworks

Mingyu Gao, Grant Ayers, Christos Kozyrakis
Stanford University
{*mgao12,geayers,kozyraki*}*@stanford.edu*

***Abstract*—The end of Dennard scaling has made all systems energy-constrained. For data-intensive applications with limited temporal locality, the major energy bottleneck is data movement between processor chips and main memory modules. For such workloads, the best way to optimize energy is to place processing near the data in main memory. Advances in 3D integration provide an opportunity to implement near-data processing (NDP) without the technology problems that similar efforts had in the past.**

This paper develops the hardware and software of an NDP architecture for in-memory analytics frameworks, including MapReduce, graph processing, and deep neural networks. We develop simple but scalable hardware support for coherence, communication, and synchronization, and a runtime system that is sufficient to support analytics frameworks with complex data patterns while hiding all the details of the NDP hardware. Our NDP architecture provides up to 16x performance and energy advantage over conventional approaches, and 2.5x over recently-proposed NDP systems. We also investigate the balance between processing and memory throughput, as well as the scalability and physical and logical organization of the memory system. Finally, we show that it is critical to optimize software frameworks for spatial locality as it leads to 2.9x efficiency improvements for NDP.

***Keywords*-Near-data processing; Processing in memory; Energy efficiency; In-memory analytics;**

I. Introduction

The end of Dennard scaling has made all systems energy-limited [1], [2]. To continue scaling performance at exponential rates, we must minimize energy overhead for every operation [3]. The era of "big data" is introducing new workloads which operate on massive datasets with limited temporal locality [4]. For such workloads, cache hierarchies do not work well and most accesses are served by main memory. Thus, it is particularly important to improve the memory system since the energy overhead of moving data across board-level and chip-level interconnects dwarfs the cost of instruction processing [2].

The best way to reduce the energy overhead of data movement is to avoid it altogether. There have been several efforts to integrate processing with main memory [5]–[10]. A major reason for their limited success has been the cost and performance overheads of integrating processing and DRAM on the same chip. However, advances in 3D integration technology allow us to place computation near memory through TSV-based stacking of logic and memory chips [11], [12]. As a result, there is again significant interest in integrating processing and memory [13].

With a practical implementation technology available, we now need to address the system challenges of near-data processing (NDP). The biggest issues are in the hardware/software interface. NDP architectures are by nature highly-distributed systems that deviate from the cache-coherent, shared-memory models of conventional systems. Without careful co-design of hardware and software runtime features, it can be difficult to efficiently execute analytics applications with non-trivial communication and synchronization patterns. We must also ensure that NDP systems are energy-optimized, balanced in terms of processing and memory capabilities, and able to scale with technology.

The goal of this paper is twofold. First, we want to design an efficient and practical-to-use NDP architecture for popular analytics frameworks including MapReduce, graph processing, and deep neural networks. In addition to using simple cores in the logic layers of 3D memory stacks in a multi-channel memory system, we add a few simple but key hardware features to support coherence, communication, and synchronization between thousands of NDP threads. On top of these features, we develop an NDP runtime that provides services such as task launch, thread communication, and data partitioning, but hides the NDP hardware details. The NDP runtime greatly simplifies porting analytics frameworks to this architecture. The end-user application code is unmodified: It is the same as if these analytics frameworks were running on a conventional system.

Second, we want to explore balance and scalability for NDP systems. Specifically, we want to quantify trade-offs on the following issues: what is the right balance of compute-to-memory throughput for NDP systems; what is the efficient communication model for the NDP threads; how do NDP systems scale; what software optimizations matter most for efficiency; what are the performance implications for the host processors in the system.

Our study produces the following insights: First, simple hardware support for coherence and synchronization and a runtime that hides their implementation from higher-level software make NDP systems efficient and practical to use with popular analytics frameworks. Specifically, using a pull-based communication model for NDP threads that utilizes the hardware support for communication provides a 2.5x efficiency improvement over previous NDP systems. Second, NDP systems can provide up to 16x overall advantage for both performance and energy efficiency over

1089-795X/15 $31.00 © 2015 IEEE
DOI 10.1109/PACT.2015.22

conventional systems. The performance of the NDP system scales well to multiple memory stacks and hundreds of memory-side cores. Third, a few (4-8) in-order, multi-threaded cores with simple caches per vertical memory channel provide a balanced system in terms of compute and memory throughput. While specialized engines can produce some additional energy savings, most of the improvement is due to the elimination of data movement. Fourth, to achieve maximum efficiency in an NDP system, it is important to optimize software frameworks for spatial locality. For instance, an edge-centric version of the graph framework improves performance and energy by more than 2.9x over the typical vertex-centric approach. Finally, we also identify additional hardware and software issues and opportunities for further research on this topic.

II. Background and Motivation

Processing-in-Memory (PIM): Several efforts in the 1990s and early 2000s examined single-chip logic and DRAM integration. EXECUBE, the first PIM device, integrated 8 16-bit SIMD/MIMD cores and 4 Mbits of DRAM [5], [6]. IRAM combined a vector processor with 13 Mbytes of DRAM for multimedia workloads [7]. DIVA [8], Active Pages [9], and FlexRAM [10] were drop-in PIM devices that augmented a host processor, but also served as traditional DRAM. DIVA and FlexRAM used programmable cores, while Active Pages used reconfigurable logic. Several custom architectures brought computation closer to data on memory controllers. The Impulse project added application-specific scatter and gather logic which coalesced irregularly-placed data into contiguous cachelines [14], and Active Memory Operations moved selected operations to the memory controller [15].

3D integration: Vertical integration with through-silicon vias (TSV) allows multiple active silicon devices to be stacked with dense interconnections [16], [17]. 3D stacking promises significant improvements in power and performance over traditional 2D planar devices. Despite thermal and yield challenges, recent advances have made this technology commercially viable [11], [12]. Two of the most prominent 3D-stacked memory technologies today are Micron's Hybrid Memory Cube (HMC) [18] and JEDEC's High Bandwidth Memory (HBM) specification [19], both of which consist of a logic die stacked with several DRAM devices. Several studies have explored the use of 3D-stacked memory for caching [20]–[24]. We focus on applications with no significant temporal locality and thus will not benefit from larger caches.

From PIM to NDP: 3D-stacking with TSVs addresses one of the primary reasons for the limited success on past PIM projects: the additional cost as well as the performance or density shortcomings of planar chips that combined processing and DRAM. Hence, there is now renewed interest in systems that use 3D integration for near-data processing [13]. Pugsley et al. evaluated a daisy chain of modified HMC devices with simple cores in the logic layers for MapReduce workloads [25]. NDA stacked Coarse-Grained Reconfigurable Arrays on commodity DRAM modules [26]. Both designs showed significant performance and energy improvements, but they also relied on host processor to coordinate data layout and necessary communication between NDP threads. Tesseract was a near-data accelerator for large-scale graph processing that provided efficient communication using message passing between memory partitions [27]. The Active Memory Cube focused on scientific workloads and used specialized vectorized processing elements with no caches [28]. PicoServer [29] and 3D-stacked server [30] focused on individual stacks for server integration, and targeted web applications and key-value store which are not as memory-intensive. Other work has studied 3D integration with GPGPUs [31], non-volatile memory [32], and other configurations [33]–[36].

However, implementation technology is not the only challenge. PIM and NDP systems are highly parallel but most do not support coherent, shared memory. Programming often requires specialized models or complex, low-level approaches. Interactions with features such as virtual memory, host processor caches, and system-wide synchronization have also been challenging. Our work attempts to address these issues and design efficient yet practical NDP systems.

The biggest opportunity for NDP systems is with emerging "big data" applications. These data-intensive workloads scan through massive datasets in order to extract compact knowledge. The lack of temporal locality and abundant parallelism suggests that NDP should provide significant improvements over conventional systems that waste energy on power-hungry processor-to-memory links. Moreover, analytics applications are typically developed using domain-specific languages for domains such as MapReduce, graphs, or deep neural networks. A software framework manages the low-level communication and synchronization needed to support the domain abstractions at high performance. Such frameworks are already quite popular and perform very well in cluster (scale-out) environments, where there is no cluster-scale cache coherence. By optimizing NDP hardware and low-level software for such analytics frameworks, we can achieve significant gains without exposing end programmers to any details of the NDP system.

III. NDP Hardware Architecture

NDP systems can be implemented with several technology options, including processing on buffer-on-board (BoB) devices [37], edge-bonding small processor dies on DRAM chips, and 3D-stacking with TSVs [11], [12]. We use 3D-stacking with TSVs because of its large bandwidth and energy advantages. Nevertheless, most insights we draw on NDP systems, such as the hardware/software interface

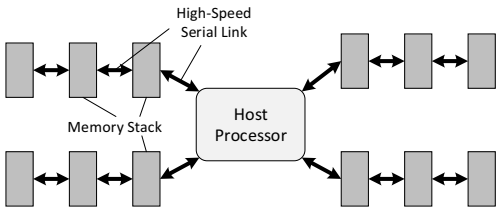


Figure 1. The Near Data Processing (NDP) architecture.

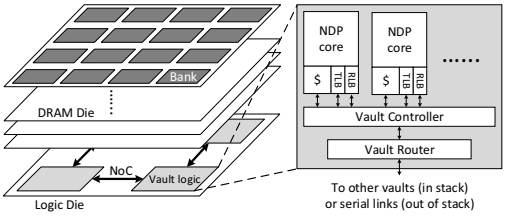


Figure 2. 3D memory stack and NDP components.

and the runtime optimizations for in-memory analytics, are applicable to NDP systems that use alternative options.

Figure 1 provides an overview of the NDP architecture we study. We start with a system based on a high-end host processor chip with out-of-order (OoO) cores, connected to multiple memory stacks. This is similar to a conventional system where the host processor uses multiple DDR3 memory channels to connect to multiple memory modules, but high-speed serial links are used instead of DDR interface. The memory stacks integrate NDP cores and memory using 3D stacking, as shown in Figure 2. The portions of applications with limited temporal locality execute on the NDP cores in order to minimize the energy overhead of data movement. The portions of applications with sufficient temporal locality execute on the host processor as usual. The NDP cores and the host processor cores see the same physical address space (shared memory).

Recently-proposed NDP hardware architectures use a similar base design. To achieve significant performance and energy improvements for complex analytics workloads, we need to further and carefully design the communication and memory models of the NDP system. Hence, after a brief overview of the base design (section III-A), we introduce the new hardware features (section III-B) that enable the software runtime discussed in section IV.

A. Base NDP Hardware

The most prominent 3D-stacked memory technologies today are Micron's Hybrid Memory Cube (HMC) [18] and JEDEC's High Bandwidth Memory (HBM) [19], [38]. Although the physical structures differ, both HMC and HBM integrate a logic die with multiple DRAM chips in a single stack, which is divided into multiple independent channels (typically 8 to 16). HBM exposes each channel as raw DDR-like interface; HMC implements DRAM controller in the logic layer as well as SerDes links for off-stack communication. We use the term *vault* to describe the vertical channel in HMC, including the memory banks and its separate DRAM controller. 3D-stacked memory provides high bandwidth through low-power TSV-based channels, while latency is close to normal DDR3 chips due to their similar DRAM core structures [38]. In this study, we use an HMC-like 4 Gbyte stack[1] with 8 DRAM dies by default, and investigate different vault organization in section VI.

We use general-purpose, programmable cores for near-data processing to enable a large and diverse set of analytics frameworks and applications. While vector processors [7], reconfigurable logic [9], [26], or specialized engines [39], [40] may allow additional improvements, our results show that most of the energy benefits are due to the elimination of data movement (see section VI). Moreover, since most of the NDP challenges are related to software, starting with programmable cores is an advantage.

Specifically, we use simple, in-order cores similar to the ARM Cortex-A7 [41]. Wide-issue or OoO cores are not necessary due to the limited locality and instruction-level parallelism in code that executes near memory, nor are they practical given the stringent power and area constraints. However, as in-memory analytics relies heavily on floating-point operations, we include one FPU per core. Each NDP core also has private L1 caches for instructions and data, 32 Kbytes each. The latter is used primarily for temporary results. There is not sufficient locality in the workloads to justify the area and power overheads of private or even shared L2 caches.

For several workloads—particularly for graph processing—the simple cores are underutilized as they are often stalled waiting for data. We use fine-grained multithreading (cycle-by-cycle) as a cost-effective way to increase the utilization of simple cores given the large amount of memory bandwidth available within each stack [42]. Only a few threads (2 to 4) are needed to match the short latency to nearby memory. The number of threads per core is also limited by the L1 cache size.

B. NDP Communication & Memory Model

The base NDP system is similar to prior NDP designs that target simple workloads, such as the embarrassingly-parallel map phase in MapReduce [25]. Many real-world applications, such as graph processing and deep learning, require more complex communication between hundreds to thousands of threads [43], [44]. Relying on the host processor to manage all the communication will not only turn it into the performance bottleneck, but will also waste

[1]Higher capacity stacks will become available as higher capacity DRAM chips are used in the future.

energy moving data between the host processor and memory stacks (see section VI). Moreover, the number of NDP cores will grow over time along with memory capacity. To fully utilize the memory and execution parallelism, we need an NDP architecture with support for efficient communication that scales to thousands of threads.

Direct communication for NDP cores: Unlike previous work [25], [26], we support direct communication between NDP cores within and across stacks because it greatly simplifies the implementation of communication patterns for in-memory analytics workloads (see section IV). The physical interconnect within each stack includes a 2D mesh network-on-chip (NoC) on the logic die that allows the cores associated with each vault to directly communicate with other vaults within the same stack. Sharing a single router per vault is area-effective and sufficient in terms of throughput. The 2D mesh also provides access to the external serial links that connect stacks to each other and to the host processor.

This interconnect allows all cores in the system, NDP and host, to access all memory stacks through a unified physical address space. An NDP core sends read/write accesses directly to its local vault controller. Remote accesses reach other vaults or stacks by routing based on physical addresses (see Figure 2). Data coherence is guaranteed with the help of virtual memory (discussed later). Remote accesses are inherently more expensive in terms of latency and energy. However, analytics workloads operate mostly on local data and communicate at well-understood points. By carefully optimizing the data partitioning and work assignment, NDP cores mostly access memory locations in their own local vaults (see section IV).

Virtual memory: NDP threads access the virtual address space of their process through OS-managed paging. Each NDP core contains a 16-entry TLB to accelerate translation. Similar to an IOMMU in conventional systems, TLB misses from NDP cores are served by the OS on the host processor. The runtime system on the NDP core communicates with the OS on a host core to retrieve the proper translation or to terminate the program in the case of an error. We use large pages (2 Mbyte) for the entire system to minimize TLB misses [45]. Thus, a small number of TLB entries is sufficient to serve large datasets for in-memory analytics, given that most accesses stay within the local vault.

We use virtual memory protection to prevent concurrent (non-coherent) accesses from the host processor and NDP cores to the same page. For example, while NDP threads are working on their datasets, the host processor has no access or read-only access for those pages. We also leverage virtual memory to implement coherence in a coarse-grained per-page manner (see below).

Software-assisted coherence: We use a simple and coarse-grained coherence model that is sufficient to support the communication patterns for in-memory analytics, namely limited sharing with moderate communication at well-specified synchronization points. Individual pages can be cached in only one cache in the system (the host processor's cache hierarchy or an NDP core's cache), which is called the *owner cache*. When a memory request is issued, the TLB identifies the owner cache, which may be local or remote to the issuing core, and the request is forwarded there. If there is a cache miss, the request is sent to the proper memory vault based on the physical address. The data is placed in the cache of the requesting core only if this is the owner cache. This model scales well as it allows each NDP core to hold its own working set in its cache without any communication. Compared to conventional directory-based coherence at cacheline granularity using models like MOESI [46], [47], our model eliminates the storage overhead and the lookup latency of directories.

To achieve high performance with our coherence model, it is critical to carefully assign the owner cache. A naive static assignment which evenly partitions the vault physical address space to each cache will not work well because two threads may have different working set sizes. We provide full flexibility to software by using an additional field (using available bits in PTEs) in each TLB entry to encode and track this page's owner cache. The NDP runtime provides an API to let the analytics framework configure this field when the host thread partitions the dataset or NDP threads allocate their local data (see section IV). In this way, even if an NDP core's dataset spills to non-local vault(s), the data can still be cached.

By treating the cache hierarchy in the host processor as a special owner cache, the same coherence model can also manage interactions between host and NDP cores. Note that the page-level access permission bits (R/W/X) can play an important role as well. For example, host cores may be given read-only access to input data for NDP threads but no access to data that are being actively modified by NDP cores. When an NDP or host core attempts to access a page for which it does not have permission, it is up to the runtime and the OS to handle it properly: signal an error, force it to wait, or transfer permissions.

Remote load buffers: While the coherence model allows NDP cores to cache their own working sets, NDP cores will suffer from latency penalties while accessing remote data during communication phases. As communication between NDP cores often involves streaming read-only accesses (see section IV), we provide a per-core *remote load buffer* (RLB), that allows sequential prefetching and buffering of a few cachelines of *read-only* data. Specifically, 4 to 8 blocks with 64 bytes per block are sufficient to accommodate a few threads. Remote stores will bypass RLBs and go to owner caches directly. RLBs are not kept coherent with remote memories or caches, thus they require explicit flushing through software. This is manageable because flushes are only necessary at synchronization points such as at barriers,

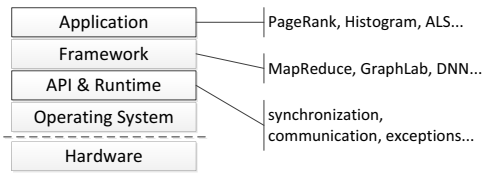


Figure 3. The NDP software stack.

and no writeback is needed. Note that caches do not have to be flushed unless there is a change in the assigned owner cache. Hence, synchronization overhead is low.

Remote atomic operations and synchronization: We support several remote atomic operations, including remote fetch-and-add and remote compare-and-swap, implemented at the vault controllers similarly to [48]. All NDP communication support is the same whether communication occurs within or across stacks. These remote atomic operations can be used to build higher-level synchronization primitives, specifically user-level locks and barriers. We use a hierarchical, tree-style barrier implementation: all threads running inside a vault will first synchronize and only one core signals an update to the rest of the stack. After all of the vaults in the stack have synchronized, one will send a message for cross-stack synchronization.

IV. Practical Software for NDP Systems

The NDP architecture in section III supports flexible inter-thread communication and scalable coherence. We exploit these features in a software infrastructure that supports a variety of popular analytics frameworks. As shown in Figure 3, a lightweight runtime interfaces between the OS and user-level software. It hides the low-level NDP hardware details and provides system services through a simple API. We ported three popular frameworks for in-memory analytics to the NDP runtime: MapReduce, graph processing, and deep neural networks. *The application developer for these frameworks is unaware that NDP hardware is used and would write exactly the same program as if all execution happened in a conventional system.* Due to the similarity between our low-level API and distributed (cluster) environments, we expect that other scale-out processing frameworks would also achieve high performance and energy efficiency on our NDP system.

A. NDP Runtime

The NDP runtime exposes a set of API functions that initialize and execute software on the NDP hardware. It also monitors the execution of NDP threads and provides runtime services such as synchronization and communication. Finally, it coordinates with the OS running on host cores for file I/O and exception handling.

Data partitioning and program launch: The NDP runtime informs applications running on the host cores about the availability of NDP resources, including the number of NDP cores, the memory capacity, and the topology of NDP components. Applications, or more typically frameworks, can use this information to optimize their data partitioning and placement strategy. The runtime provides each NDP thread a private stack and a heap, and works with the OS to allocate memory pages. Applications can optionally specify the owner cache for each page (by default the caller thread's local cache), and preferred stacks or vaults in which to allocate physical memory. This is useful when the host thread partitions the input dataset. The runtime prioritizes allocation on the vault closest to the owner cache as much as possible before spilling to nearby vaults. However, memory allocations do not have to perfectly fit within the local vault since if needed an NDP thread can access remote vaults with slightly worse latency and power. Finally, the runtime starts and terminates NDP threads with an interface similar to POSIX threads, but with hints to specify target cores. This enables threads to launch next to their working sets and ensures that most memory accesses are to local memory.

Communication model: In-memory analytics workloads usually execute iteratively. In every iteration, each thread first processes an independent sub-dataset in parallel for a certain period (*parallel phase*), and then exchanges data with other threads at the end of the current iteration (*communication phase*). MapReduce [49], graph processing [43], [50], and deep neural networks [44] all follow this model. While the parallel phase is easy to implement, the communication phase can be challenging. This corresponds to the shuffle phase in MapReduce, scatter/gather across graph tiles in graph, and forward/backward propagation across network partitions in deep neural networks.

We build on the hardware support for direct communication between NDP threads to design a *pull* model for data exchange. A producer thread will buffer data locally in its own memory vault. Then, it will send a short message containing an address and size to announce the availability of data. The message is sent by writing to a predefined mailbox address for each thread. The consumer will later process the message and use it to pull (remote load) the data. The remote load buffers ensure that the overheads of remote loads for the pull are amortized through prefetching. We allocate a few shared pages within each stack to implement mailboxes. This pull-based model of direct communication is significantly more efficient and scalable than communication through the host processor in previous work [25], [26]. First, communication between nearby cores does not need to all go through the host processor, resulting in shorter latency and lower energy cost. Second, all threads can communicate asynchronously and in parallel, which eliminates global synchronization and avoids using only a limited number of host threads. Third, consumer threads can directly use or apply the remote data

while pulling, avoiding the extra copy cost.

Exception handling and other services: The runtime initiates all NDP cores with a default exception handler that forwards to the host OS. Custom handlers can also be registered for each NDP core. NDP threads use the runtime to request file I/O and related services from the host OS.

B. In-memory Analytics Frameworks

While one can program straight to the NDP runtime API, it is best to port to the NDP runtime domain-specific frameworks that expose higher-level programming interfaces. We ported three analytics frameworks. In each case, the framework utilizes the NDP runtime to optimize access patterns and task distribution. The NDP runtime hides all low-level details of synchronization and coherence.

First, we ported the Phoenix++ framework for in-memory MapReduce [51]. Map threads process input data and buffer the output locally. The shuffle phase follows the pull model, where each reduce thread will remotely fetch the intermediate data from the map threads' local memory. Once we ported Phoenix++, all Phoenix workloads run without modification. Second, we developed an in-memory graph framework that follows the gather-apply-scatter approach of GraphLab [43]. The framework handles the gather and scatter communication, while the high-level API visible to the programmer is similar to GraphLab and does not expose any of the details of NDP system. Third, we implemented a parallel deep neural network (DNN) framework based on Project Adam [44]. This framework supports both network training and prediction for various kinds of layers. Each layer in the network is vertically partitioned to minimize cross-thread communication. Forward and backward propagation across threads are implemented using communication primitives in the NDP runtime.

Applications developed with the MapReduce framework perform mostly sequential (streaming) accesses as map and reduce threads read their inputs. This is good for energy efficiency as it amortizes the overhead of opening a DRAM row (most columns are read) and moving a cacheline to the NDP core (most bytes are used). This is not necessarily the case for the graph framework that performs random accesses and uses only a fraction of the data structure for each vertex. To explore the importance of optimizing software for spatial locality for NDP systems, we developed a fourth framework, a second version of the graph framework that uses the same high-level API. While the original version uses a vertex-centric organization where computation accesses vertices and edges randomly, the second implementation is modeled after the X-Stream system that is edge-centric and streams edges which are arranged consecutively in memory [50]. We find that the edge streaming method is much better suited to NDP (see section VI).

C. Discussion

The current version of the NDP runtime does not implement load balancing. We expect load balancing to be handled in each analytics framework and most frameworks already do this (e.g., MapReduce). Our NDP runtime can support multiple analytics applications and multiple frameworks running concurrently by partitioning NDP cores and memory. The virtual memory and TLB support provide security isolation.

The NDP hardware and software are optimized for executing the highly-parallel phases with little temporal locality on NDP cores, while less-parallel phases with high temporal locality run on host cores. Division of labor is managed by framework developers given their knowledge about the locality and parallelism. Our experience with the frameworks we ported shows that communication and coordination between host and NDP cores is infrequent and involves small amounts of data (e.g., the results of a highly-parallel memory scan). The lack of fine-grained cache coherence between NDP and host cores is not a performance or complexity issue.

A key departure in our NDP system is the need for coarse-grained address interleaving. Conventional systems use fine-grained interleaving where sequential cachelines in the physical address space are interleaved across channels, ranks, and banks in a DDRx memory system. This optimizes bandwidth and latency for applications with both sequential and random access patterns. Unfortunately, fine-grained interleaving would eliminate most of the NDP benefits. The coarse-grained interleaving we use is ideal for execution phases that use NDP cores but can slow down phases that run on the host cores. In section VI, we show that this is not a major issue. Host cores are used with cache-friendly phases. Hence, while coarse-grained interleaving reduces the memory bandwidth available for some access patterns, once data is in the host caches execution proceeds at full speed. Most code that would benefit from fine-grained partitioning runs on NDP cores anyway.

V. Methodology

A. Simulation Models

We use zsim, a fast and accurate simulator for thousand-core systems [52]. We modified zsim to support fine-grained (cycle-by-cycle) multithreading for the NDP cores and TLB management. We also extended zsim with a detailed memory model based on DDR3 DRAM. We validated the model with DRAMSim2 [53] and against a real system. Timing parameters for 3D-stacked memory are conservatively inferred from publicly-available information and research literature [18], [38], [54]–[56].

Table I summarizes the simulated systems. Our conventional baseline system (Conv-DDR3) includes a 16-core OoO processor and four DDR3-1600 memory channels with 4 ranks per channel. We also simulate another system, Conv-3D, which combines the same processor with eight 3D

Host Processor	
Cores	16 x86-64 OoO cores, 2.6 GHz
L1I cache	32 KB, 4-way, 3-cycle latency
L1D cache	32 KB, 8-way, 4-cycle latency
L2 cache	private, 256 KB, 8-way, 12-cycle latency
L3 cache	shared, 20 MB, 8 banks, 20-way, 28-cycle latency
TLB	32 entries, 2 MB page, 200-cycle miss penalty
NDP Logic	
Cores	in-order fine-grained MT cores, 1 GHz
L1I cache	32 KB, 2-way, 2-cycle latency
L1D cache	32 KB, 4-way, 3-cycle latency
TLB	16 entries, 2 MB page, 120-cycle miss penalty
DDR3-1600	
Organization	32 GB, 4 channels $\times$ 4 ranks, 2 Gb, x8 device
Timing Parameters	$t_{CK} = 1.25$ ns, $t_{RAS} = 35.0$ ns, $t_{RCD} = 12.5$ ns $t_{CAS} = 12.5$ ns, $t_{WR} = 15.0$ ns, $t_{RP} = 12.5$ ns
Bandwidth	12.8 GBps $\times$ 4 channels
3D Memory Stack	
Organization	32 GB, 8 layers $\times$ 16 vaults $\times$ 8 stacks
Timing Parameters	$t_{CK} = 1.6$ ns, $t_{RAS} = 22.4$ ns, $t_{RCD} = 11.2$ ns $t_{CAS} = 11.2$ ns, $t_{WR} = 14.4$ ns, $t_{RP} = 11.2$ ns
Serial links	160 GBps bidirectional, 8-cycle latency
On-chip links	16 Bytes/cycle, 4-cycle zero-load delay

Table I
THE KEY PARAMETERS OF THE SIMULATED SYSTEMS.

memory stacks connected into four chains. The serial links of each chain require roughly the same number of pins from the processor chip as a 64-bit DDR3 channel [18], [37]. Both systems use closed-page policy. Each serial link has an 8-cycle latency, including 3.2 ns for SerDes [55]. The on-chip NoC in the logic layer is modeled as a 4×4 2D-mesh between sixteen vaults with 128-bit channels. We assume 3 cycles for router and 1 cycle for wire as the zero-load delay [57], [58]. Finally, the NDP system extends the conventional 3D memory system by introducing a number of simple cores with caches into the logic layer. We use 64-byte lines in all caches by default.

B. Power and Area Models

We assume 22 nm technology process for all logic, and that the area budget of the logic layer in the 3D stack is 100 mm^2. We use McPAT 1.0 to estimate the power and area of the host processor and the NDP cores [59]. We calculate dynamic power using its peak value and core utilization statistics. We also account for the overheads of the FPUs. We use CACTI 6.5 for cache power and area [60]. The NDP L1 caches use the ITRS-HP process for the peripheral circuit and the ITRS-LSTP process for the cell arrays. The use of ITRS-LSTP transistors does not violate timing constraints due to the lower frequency of NDP cores.

We use the methodology in [61] to calculate memory energy. DDR3 IDD values are taken from datasheets. For 3D memory stacks, we scale the static power with different bank organization and bank numbers, and account for the replicated peripheral circuits for each vault. For dynamic power, we scale ACT/PRE power based on the smaller page size and reduced latency. Compared to DDR3, RD/WR power increases due to the wider I/O of 3D stacking, but drops due to the use of smaller banks and shorter global wires (TSVs). Overall, our 3D memory power model results in roughly 10 to 20 pJ/bit, which is close to but more conservative than the numbers reported in HMC literature [54], [62].

We use Orion 2.0 for interconnect modeling [63]. The vault router runs at 1 GHz, and has 4 I/O ports with 128-bit flit width. Based on the area of the logic layer, we set wire length between two vaults to 2.5 mm. We assume that each serial link and SerDes between stacks and the host processor consume 1 pJ/bit for idle packets, and 3 pJ/bit for data packets [25], [55], [62]. We also model the overheads of routing between stacks in the host processor.

C. Workloads

We use the frameworks discussed in section IV: Phoenix++ for in-memory MapReduce analytics [51], the two implementations of the graph processing based on the gather-apply-scatter model [43], [50], and a deep neural network framework [44]. We choose a set of representative workloads for each framework described in Table II. Graph applications compute on real-world social networks and online reviews obtained from [64]. Overall, the workloads cover a wide range of computation and memory patterns. For instance, histogram (Hist) and linear regression (LinReg) do simple linear scans; PageRank is both read- and write-intensive; ALS requires complex matrix computation to derive the feature vectors; ConvNet needs to transfer lots of data between threads; MLP and dA (denoising autoencoder) have less communication due to combined propagation data.

We also implement one application per framework that uses both the host processor and NDP cores in different phases with different locality characteristics. They are similar to real-world applications with embedded, memory-intensive kernels. FisherScoring is an iterative logistic regression method. We use MapReduce on NDP cores to calculate the dataset statistics to get the gradient and Hessian matrix, and then solve the optimal parameters on the host processor. KCore decomposition first computes the maximal induced subgraph where all vertices have degree at least k using NDP cores. Then, the host processor calculates connected components on the smaller resultant graph. ConvNet-Train trains multiple copies of the LeNet-5 Convolutional Neural Network [65]. It uses NDP cores to process the input images for forward and backward propagation, and the host processor will periodically collect the parameter updates and send new parameters to each NDP worker [44].

VI. EVALUATION

We now present the results of our study, focusing on the key insights and trade-offs in the design exploration. Unless otherwise stated, the graph workloads use the edge-centric implementation of the graph framework. Due to space

Framework	Application	Data type	Data element	Input dataset	Note
MapReduce	Hist	Double	8 Bytes	Synthetic 20 GB binary file	Large intermediate data
	LinReg	Double	8 Bytes	Synthetic 2 GB binary file	Linear scan
	grep	Char	1 Bytes	3 GB text file	Communication-bound
	FisherScoring	Double	8 Bytes	Synthetic 2 GB binary file	Hybrid and iterative
Graph	PageRank	Double	48 Bytes	1.6M nodes, 30M edges social graph	Read- and write-intensive
	SSSP	Int	32 Bytes	1.6M nodes, 30M edges social graph	Unbalanced load
	ALS	Double	264 Bytes	8M reviews for 253k movies	Complex matrix computation
	KCore	Int	32 Bytes	1.6M nodes, 30M edges social graph	Hybrid
DNN	ConvNet	Double	8 Bytes	MNIST dataset, 70k 32×32 images	Partial connected layers
	MLP	Double	8 Bytes	MNIST dataset, 70k 32×32 images	Fully connected layers
	dA	Double	8 Bytes	Synthetic 500-dimension input data	Fully connected layers
	ConvNet-Train	Double	8 Bytes	MNIST dataset, 70k 32×32 images	Hybrid and iterative

Table II
THE KEY CHARACTERISTICS OF MAPREDUCE, GRAPH, AND DNN WORKLOADS AND THEIR DATASETS.

limitations, in some cases we present results for the most representative subset of applications: Hist for MapReduce, PageRank for graph, and ConvNet for DNN.

A. NDP Design Space Exploration

Compute/memory bandwidth balance: We first explore the balance between compute and memory bandwidth in the NDP system. We use a single-stack configuration with 16 vaults and vary the number of cores per vault (1 to 16), their frequency (0.5 or 1 GHz), and the number of threads per core (1 to 4). The maximum bandwidth per stack is 160 Gbytes/s. Figure 4 shows both the performance and maximum vault bandwidth utilization.

Performance scales almost linearly with up to 8 cores and benefits significantly from higher clock frequency and 2 threads per core. Beyond 8 cores, scaling slows down for many workloads due to bandwidth saturation. For PageRank, there is a slight drop due to memory congestion. The graph and DNN workloads saturate bandwidth faster than MapReduce workloads. Even with the edge-centric scheme, graph workloads still stress the random access bandwidth, as only a fraction of each accessed cacheline is utilized (see Figure 6). DNN workloads also require high bandwidth as they work on vector data. In contrast, MapReduce workloads perform sequential accesses, which have high cacheline utilization and perform more column accesses per opened DRAM row.

Overall, the applications we study need no more than eight 2-threaded cores running at 1 GHz to achieve balance between the compute and memory resources. Realistic area constraints for the logic die further limit us to 4 cores per vault. Thus, for the rest of the paper, we use four 2-threaded cores at 1 GHz. This configuration requires 61.7 mm^2 for the NDP cores and caches, while the remaining 38.3 mm^2 are sufficient for the DRAM controllers, the interconnect, and the circuitry for external links.

NDP memory hierarchy: Figure 5 shows the relative performance for using different stack structures. HMC-like stack uses 16 vaults with 64-bit data bus, and HBM-like stack uses 8 vaults with 128-bit data bus. We keep the same total number of data TSVs between the two stacks. Performance is less sensitive to the number of vaults per stack and the width of the vault bus. HBM is preferred for DNN workloads because they usually operate on vectors where wider buses could help with prefetching.

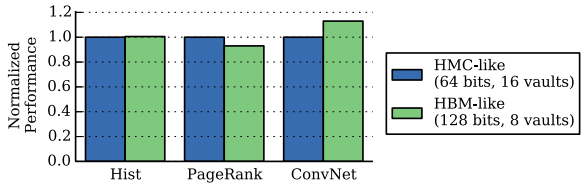


Figure 5. Performance impact of stack design.

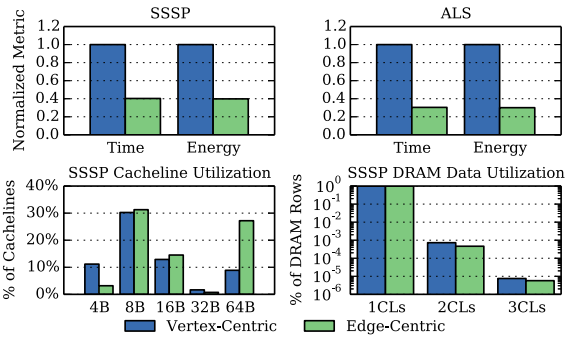


Figure 6. Vertex-centric and edge-centric graph frameworks.

We have also looked into using different cache structures. When varying L1 cacheline sizes, longer cachelines are always better for MapReduce workloads because their streaming nature (more spatial locality) amortizes the higher cache miss penalty in terms of time and energy. For graph and DNN workloads, the best cacheline size is 64 bytes; longer cachelines lead to worse performance due to the lack of spatial locality. Using a 256-Kbyte L2 cache per core leads to no obvious improvement for all workloads, and even introduces extra latency in some cases.

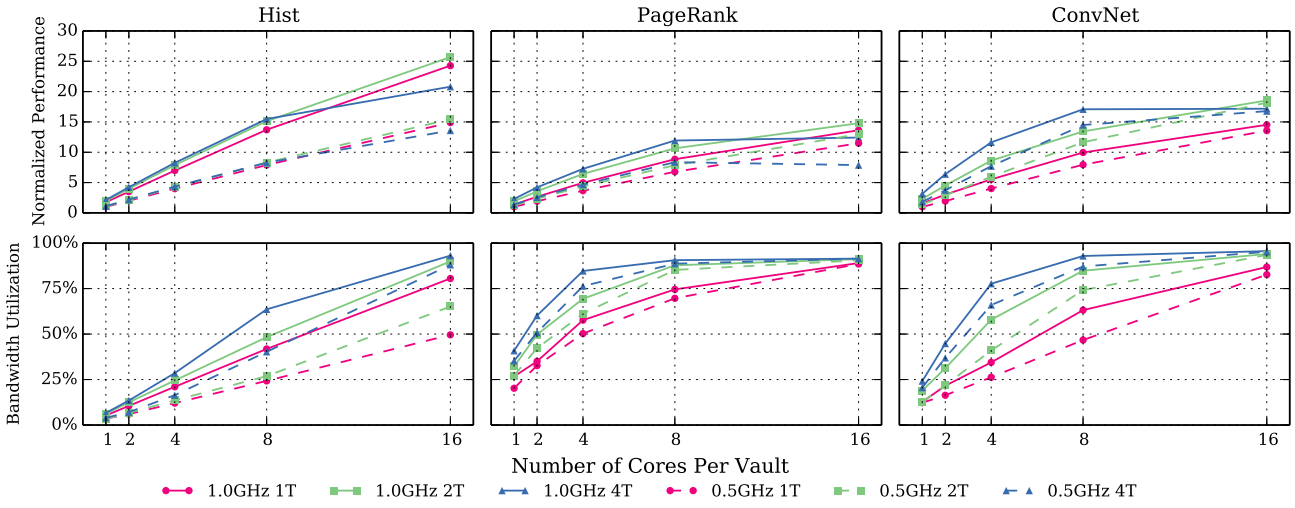


Figure 4. Performance scaling and bandwidth utilization for a single-stack NDP system.

The impact of software optimizations: The most energy-efficient way to access memory is to perform sequential accesses. In this case, the energy overhead of fetching a cacheline and opening a DRAM row is amortized by using (nearly) all bytes in the cacheline or DRAM row. While the hardware design affects efficiency as it determines the energy overheads and the width of the row, it is actually the software that determines how much spatial locality is available during execution.

Figure 6 compares the performance, energy, cacheline utilization, and DRAM row utilization for the two implementations of the graph framework (using open-page policy). The edge-centric implementation provides a 2.9x improvement in both performance and energy over the vertex-centric implementation. The key advantage is that the edge-centric scheme optimizes for spatial locality (streaming, sequential accesses). The cacheline utilization histogram shows that the higher spatial locality translates to a higher fraction of the data used within each cacheline read from memory, and a lower total number of cachelines that need to be fetched (not shown in the figure).

Note that the number of columns accessed per DRAM row is rather low overall, even with the edge-centric design. This is due to several reasons. First, spatial locality is still limited, usually less than column size. Second, these workloads follow multiple data streams that frequently cause row conflicts within banks. Finally, the short refresh interval in DDR3 (7.8μs) prevents the row buffer from being open for the very long time needed to capture further spatial locality. Overall, we believe there is significant headroom for software and hardware optimizations that further improve the spatial locality and energy efficiency of NDP systems.

NDP scaling: We now scale the number of stacks in the NDP system to observe scaling bottlenecks due to communication between multiple stacks. We use two cores per vault with 16 vaults per stack. The host processor can directly connect with up to 4 stacks (limited by pin constraints). Hence, for the configurations with 8 and 16 stacks, we connect up to 4 stacks in a chain as shown in Figure 1. Figure 7 shows that applications scale very well up to 16 stacks. With 8 stacks, the average bandwidth on inter-stack links is less than 2 Gbytes/s out of the peak of 160 Gbytes/s per link. Even in communication-heavy phases, the traffic across stacks rarely exceeds 20% of the peak throughput. Most communication traffic is handled within each stack by the network on chip, and only a small percentage of communication needs to go across the serial links.

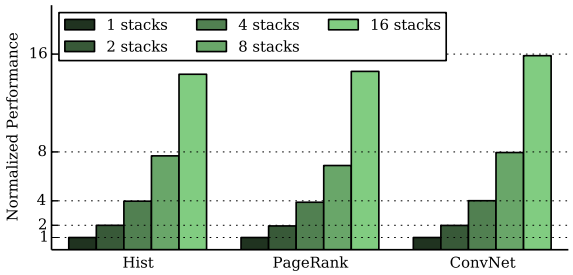


Figure 7. Performance as a function of the number of stacks.

B. Performance and Energy Comparison

We now compare the NDP system to the baseline Conv-DDR3 system, the Conv-3D system (3D stacking without processing in the logic layer), and the base NDP system in section III-A that uses the host processor for communication between NDP threads [25]. We use four 2-threaded cores per

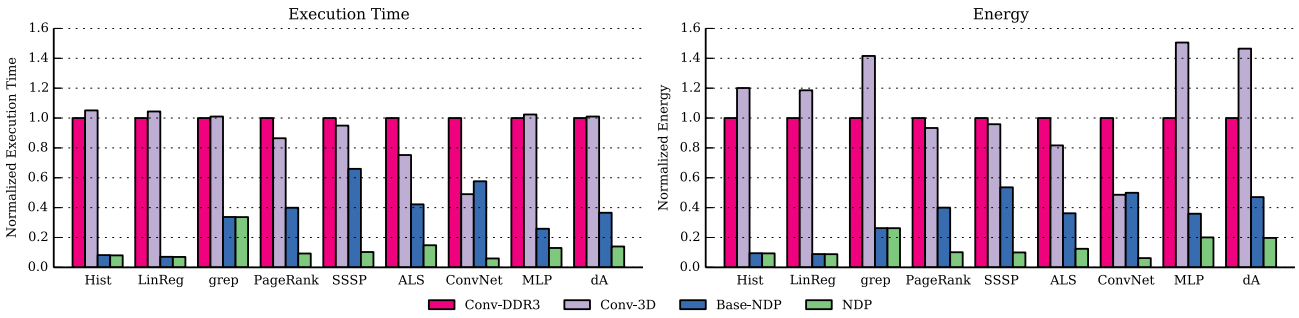


Figure 8. Performance and energy comparison between Conv-DDR3, Conv-3D, Base-NDP and NDP systems.

vault and 16 vaults per stack. This means 512 NDP cores (1024 threads) across 8 stacks.

Figure 8 shows the performance and energy comparison between the four systems. The Conv-3D system provides significantly higher bandwidth than the DDR3 baseline due to the use of high-bandwidth 3D memory stacks. This is particularly important for the bandwidth-bound graph workloads that improve by up to 25% in performance and 19% in energy, but less critical for the other two frameworks. The Conv-3D system is actually less energy-efficient for these workloads due to the high background power of the memory stacks and the underutilized high-speed links. The slight performance drop for Hist, MLP, etc. is because the sequential accesses are spread across too many channels in Conv-3D, and thus there are fewer requests in each channel that can be coalesced and scheduled to utilize the opened row buffers.

The two NDP systems provide significant improvement over both the Conv-DDR3 and the Conv-3D systems in terms of performance *and* energy. The base-NDP system is overall 3.5x faster and 3.4x more energy efficient over the DDR3 baseline. Our NDP system provides another 2.5x improvement over the base-NDP, and achieves 3-16x better performance and 4-16x less energy over the base-DDR3 system. This is primarily due to the efficient communication between NDP threads (see section III and IV). The benefits are somewhat lower for grep, MLP and dA. Grep works on 1-byte char data which requires less bandwidth per computation. MLP and dA are computationally intensive and their performance is limited by the capabilities of the simple NDP cores.

Figure 9 provides further insights into the comparison by breaking down the average power consumption. Note that the four systems are engineered to consume roughly the same power and utilize the same power delivery and cooling infrastructure. The goal is to deliver the maximum performance within the power envelope and avoid energy waste. Both conventional systems consume half power in the memory system and half in the processor. The cores are mostly stalled waiting for memory, but idleness is not

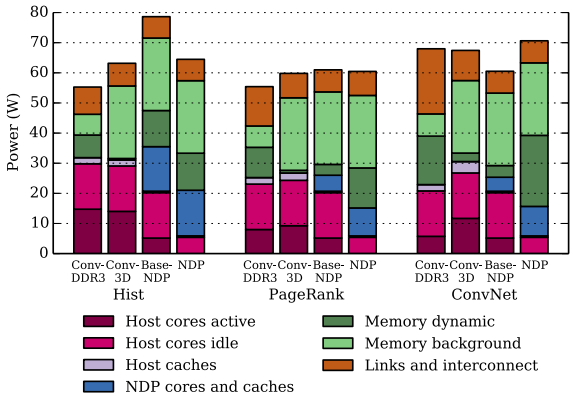


Figure 9. System power breakdown (host processor + 8 stacks).

sufficiently long to invoke deeper low power modes (e.g., C3 to C6 modes). The DDR3 channels are burning dynamic energy to move data with little reuse for amortization, with relatively low background energy due to the modest capacity. In the Conv-3D memory system, background power is much higher due to the large number of banks and replicated peripheral circuitry. Dynamic energy decreases due to efficient 3D structure and TSV channels, but bandwidth is seriously underutilized.

For the NDP system, dynamic memory power is higher as there are now hundreds of NDP threads issuing memory accesses. The host cores in our NDP system are idle for long periods now and can be placed into deep low-power modes. However, for the base NDP system, workloads which have iterative communication require the host processor to coordinate, thus the processor spends more power. The NDP cores across the 8 stacks consume significant power, but remain well-utilized. Replacing these cores with energy-efficient custom engines would provide further energy improvement (up to an additional 20%), but these savings are much lower than those achieved by eliminating data movement with NDP. A more promising direction is to focus the design of custom engines on performance improvements (rather than

power), e.g., by exploiting SIMD for MapReduce and DNN workloads. This approach is also limited by the available memory bandwidth (see Figure 4).

We do not include a comparison with baseline systems replacing OoO cores with many small cores at the host processor side. The Conv-DDR3 system is already bandwidth-limited, therefore using more cores would not change its performance. A Conv-3D system would achieve performance improvements with more small cores but would still spend significant energy on the high-speed memory links. Moreover, as we can infer from Figure 9 it would lead to a power problem by significantly increasing the dynamic power spent on the links. Overall, it is better to save interconnect energy by moving a large number of small cores close to memory *and* maintaining OoO cores on the host for the workloads that actually need high ILP [66].

Regarding thermal issues, our system is nearly power-neutral compared with the Conv-3D system. This can be further improved with better power management of the links (e.g., turn off some links when high bandwidth is not needed), as they draw significant static power but are seriously underutilized. Also, our power overhead per stack is close to previous work [25], which has already demonstrated the feasibility of adding logic into HMC [67].

C. System Challenges

The NDP system uses coarse-grained rather than fine-grained address interleaving. To understand the impact of this change, we run the SPEC CPU2006 benchmarks on the Conv-3D system with coarse-grained and fine-grained interleaving. All processing is performed on the host processor cores. For the benchmarks that cache reasonably well in the host LLC (perlbench, gcc, etc.), the impact is negligible (<1%). Among the memory-intensive benchmarks (libquantum, mcf, etc.), coarse-grained interleaving leads to an average 10% slowdown (20.7% maximum for GemsFDTD). Overall, this performance loss is not trivial but it is not huge either. Hence, we believe it is worth it to use coarse-grained interleaving to enable the large benefits from NDP for in-memory analytics, even if some host-side code suffers a small degradation. Nevertheless, we plan to study adaptive interleaving schemes in future work.

Finally, Figure 10 compares the performance of the hybrid workloads on the four systems. Energy results are similar. The memory-intensive phases of these workloads execute on NDP cores, leading to overall performances gain of 2.5x to 13x over the DDR3 baseline. The compute-intensive phases execute on the host processor on all systems. ConvNet-Train has negligible compute-intensive work. The baseline NDP system uses the host processor for additional time in order to coordinate communication between NDP cores. In our NDP system, in contrast, NDP cores coordinate directly. While this increases their workload (see KCore), this work is parallelized and executes faster than on the host processor.

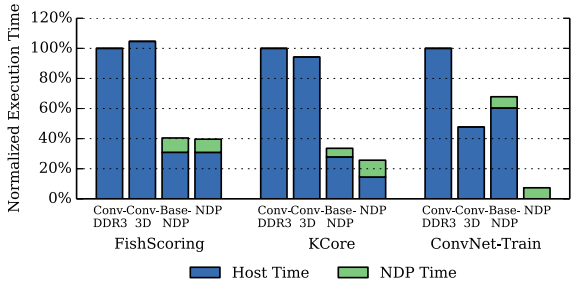


Figure 10. Hybrid workload performance comparison.

VII. Conclusion

We presented the hardware and software features necessary for efficient and practical near-data processing for in-memory analytics (MapReduce, graph processing, deep neural networks). By placing simple cores close to memory, we eliminate the energy waste for data movement in these workloads with limited temporal locality. To support non-trivial software patterns, we introduce simple but scalable NDP hardware support for coherence, virtual memory, communication and synchronization, as well as a software runtime that hides all details of NDP hardware from the analytics frameworks. Overall, we demonstrate up to 16x improvement on both performance and energy over the existing systems. We also demonstrate the need of the coherence and communication support in NDP hardware and the need to optimize software for spatial locality in order to maximize NDP benefits.

Acknowledgments

The authors want to thank Mark Horowitz, Heonjae Ha, Kenta Yasufuku, Makoto Takami, and the anonymous reviewers for their insightful comments on earlier versions of this paper. This work was supported by the Stanford Pervasive Parallelism Lab, the Stanford Experimental Datacenter Lab, Samsung, and NSF grant SHF-1408911.

References

[1] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *ISCA-38*, 2011, pp. 365–376.
[2] S. Keckler, "Life After Dennard and How I Learned to Love the Picojoule," Keynote in MICRO-44, Dec. 2011.
[3] M. Horowitz *et al.*, "Scaling, power, and the future of CMOS," in *IEDM-2005*, Dec 2005, pp. 7–15.
[4] Computing Community Consortium, "Challenges and Opportunities with Big Data," http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf, 2012.
[5] P. M. Kogge, "EXECUBE-A New Architecture for Scaleable MPPs," in *ICCP-1994*, 1994, pp. 77–84.
[6] P. M. Kogge *et al.*, "Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies," in *FMPC-6*, 1996, pp. 88–97.
[7] D. Patterson *et al.*, "A Case for Intelligent RAM," *Micro, IEEE*, vol. 17, no. 2, pp. 34–44, 1997.
[8] M. Hall *et al.*, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *SC'99*, 1999, p. 57.
[9] M. Oskin *et al.*, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA-25*, 1998, pp. 192–203.

[10] Y. Kang *et al.*, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD-30*, 2012, pp. 5–14.

[11] M. Motoyoshi, "Through-Silicon Via (TSV)," *Proceedings of the IEEE*, vol. 97, no. 1, pp. 43–48, Jan 2009.

[12] A. W. Topol *et al.*, "Three-Dimensional Integrated Circuits," *IBM Journal of Research and Development*, vol. 50, no. 4.5, pp. 491–506, July 2006.

[13] R. Balasubramonian *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *Micro, IEEE*, vol. 34, no. 4, pp. 36–42, July 2014.

[14] J. Carter *et al.*, "Impulse: Building a Smarter Memory Controller," in *HPCA-5*, 1999, pp. 70–79.

[15] Z. Fang *et al.*, "Active Memory Operations," in *ICS-21*, 2007, pp. 232–241.

[16] S. J. Souri *et al.*, "Multiple Si Layer ICs: Motivation, Performance Analysis, and Design Implications," in *DAC-37*, 2000, pp. 213–220.

[17] S. Das *et al.*, "Technology, Performance, and Computer-Aided Design of Three-dimensional Integrated Circuits," in *ISPD-2004*, 2004, pp. 108–115.

[18] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification 1.0," 2013.

[19] JEDEC Standard, "High Bandwidth Memory (HBM) DRAM," JESD235, 2013.

[20] K. Puttaswamy and G. H. Loh, "Implementing Caches in a 3D Technology for High Performance Processors," in *ICCD-2005*, 2005, pp. 525–532.

[21] G. H. Loh, "Extending the Effectiveness of 3D-stacked DRAM Caches with an Adaptive Multi-Queue Policy," in *MICRO-42*, 2009, pp. 201–212.

[22] X. Jiang *et al.*, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *HPCA-16*, 2010, pp. 1–12.

[23] G. H. Loh and M. D. Hill, "Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap," *Micro, IEEE*, vol. 32, no. 3, pp. 70–78, May 2012.

[24] D. Jevdjic *et al.*, "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *ISCA-40*, 2013, pp. 404–415.

[25] S. Pugsley *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+ Logic Devices on MapReduce Workloads," in *ISPASS-2014*, 2014.

[26] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *HPCA-21*, Feb 2015, pp. 283–295.

[27] J. Ahn *et al.*, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *ISCA-42*, 2015, pp. 105–117.

[28] R. Nair *et al.*, "Active Memory Cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, March 2015.

[29] T. Kgil *et al.*, "PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor," in *ASPLOS-12*, 2006, pp. 117–128.

[30] A. Gutierrez *et al.*, "Integrated 3d-stacked server designs for increasing physical density of key-value stores," in *ASPLOS-19*, 2014, pp. 485–498.

[31] D. Zhang *et al.*, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC-23*, 2014, pp. 85–98.

[32] P. Ranganathan, "From microprocessors to nanostores: Rethinking data-centric systems," *Computer*, vol. 44, no. 3, pp. 39–48, 2011.

[33] D. Fick *et al.*, "Centip3De: A Cluster-Based NTC Architecture With 64 ARM Cortex-M3 Cores in 3D Stacked 130 nm CMOS," *JSSC*, vol. 48, no. 1, pp. 104–117, Jan 2013.

[34] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *ISCA-35*, June 2008, pp. 453–464.

[35] Y. Pan and T. Zhang, "Improving VLIW Processor Performance Using Three-Dimensional (3D) DRAM Stacking," in *ASAP-20*, July 2009, pp. 38–45.

[36] D. H. Woo *et al.*, "An Optimized 3D-stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth," in *HPCA-16*, Jan 2010, pp. 1–12.

[37] E. Cooper-Balis *et al.*, "Buffer-On-Board Memory Systems," in *ISCA-39*, 2012, pp. 392–403.

[38] D. U. Lee *et al.*, "25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *ISSCC-2014*, Feb 2014, pp. 432–433.

[39] P. Dlugosch *et al.*, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *TPDS*, p. 1, 2014.

[40] T. Zhang *et al.*, "A 3D SoC design for H.264 application with on-chip DRAM stacking," in *3DIC-2010*, Nov 2010, pp. 1–6.

[41] ARM, "Cortex-A7 Processor," http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[42] R. Alverson *et al.*, "The tera computer system," in *ACM SIGARCH Computer Architecture News*, 1990, pp. 1–6.

[43] J. E. Gonzalez *et al.*, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *OSDI-10*, 2012, pp. 17–30.

[44] T. Chilimbi *et al.*, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *OSDI-11*, 2014, pp. 571–582.

[45] J. Navarro *et al.*, "Practical, Transparent Operating System Support for Superpages," in *OSDI-5*, 2002, pp. 89–104.

[46] M. Ferdman *et al.*, "Cuckoo directory: A scalable directory for many-core systems," in *HPCA-17*, 2011, pp. 169–180.

[47] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *HPCA-18*, February 2012.

[48] J. H. Ahn *et al.*, "Scatter-Add in Data Parallel Architectures," in *HPCA-11*, Feb 2005, pp. 132–142.

[49] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI-6*, 2004, pp. 10–10.

[50] A. Roy *et al.*, "X-Stream: Edge-centric Graph Processing Using Streaming Partitions," in *SOSP-24*, 2013, pp. 472–488.

[51] J. Talbot *et al.*, "Phoenix++: Modular MapReduce for Shared-memory Systems," in *MapReduce-2011*, 2011, pp. 9–16.

[52] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems," in *ISCA-40*, 2013, pp. 475–486.

[53] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[54] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," Presented in HotChips 23, 2011.

[55] G. Kim *et al.*, "Memory-Centric System Interconnect Design With Hybrid Memory Cubes," in *PACT-22*, Sept 2013, pp. 145–155.

[56] C. Weis *et al.*, "Design Space Exploration for 3D-stacked DRAMs," in *DATE-2011*, March 2011, pp. 1–6.

[57] B. Grot *et al.*, "Express Cube Topologies for On-Chip Interconnects," in *HPCA-15*, Feb 2009, pp. 163–174.

[58] J. Balfour and W. J. Dally, "Design Tradeoffs for Tiled CMP On-chip Networks," in *ICS-20*, 2006, pp. 187–198.

[59] S. Li *et al.*, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO-42*, 2009, pp. 469–480.

[60] N. Muralimanohar *et al.*, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO-40*, 2007, pp. 3–14.

[61] Micron Technology Inc., "TN-41-01: Calculating Memory System Power for DDR3," http://www.micron.com/products/support/power-calc, 2007.

[62] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *VLSIT*, June 2012, pp. 87–88.

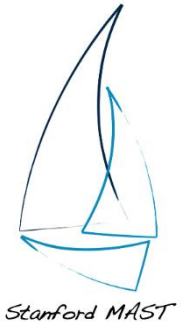
[63] A. B. Kahng *et al.*, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration," in *DATE-2009*.

[64] "Stanford Large Network Dataset Collection," http://snap.stanford.edu/data/index.html, available Online.

[65] Y. Lecun *et al.*, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[66] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE Micro*, vol. 30, no. 4, 2010.

[67] Y. Eckert *et al.*, "Thermal Feasibility of Die-Stacked Processing in Memory," in *2nd Workshop on Near-Data Processing (WoNDP)*, December 2014.



Practical Near-Data Processing for In-Memory Analytics Frameworks

Mingyu Gao, Grant Ayers, Christos Kozyrakis

Stanford University

http://mast.stanford.edu

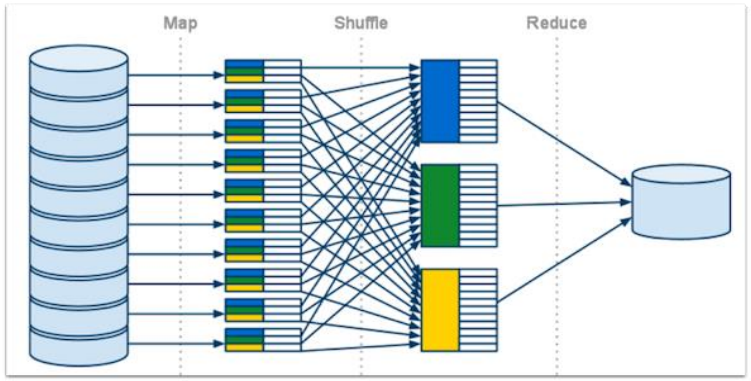
Stanford | ENGINEERING
Electrical Engineering

PACT – Oct 19, 2015

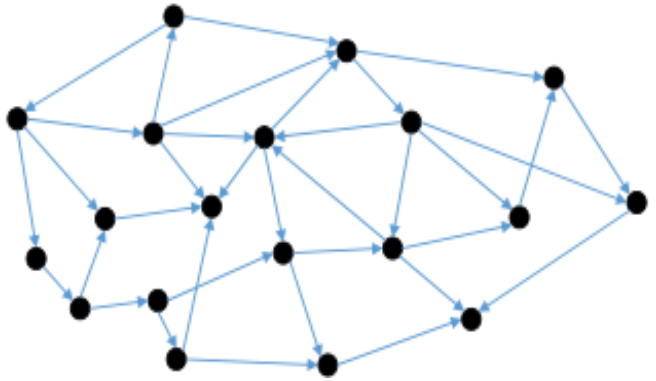
Motivating Trends

Stanford | ENGINEERING
Electrical Engineering

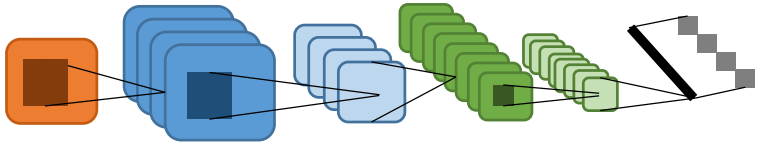
- End of Dennard scaling → systems are energy limited
- Emerging big data workloads
  - Massive datasets, limited temporal locality, irregular access patterns
  - They perform poorly on conventional cache hierarchies
- Need alternatives to improve energy efficiency



MapReduce



Graphs

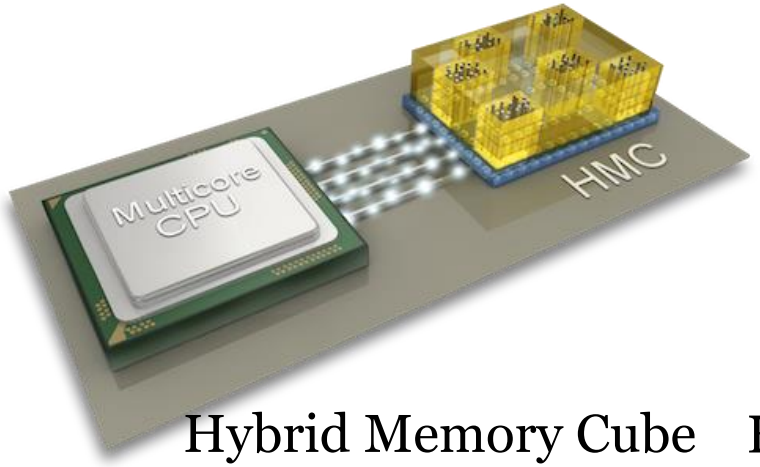


Deep Neural Networks

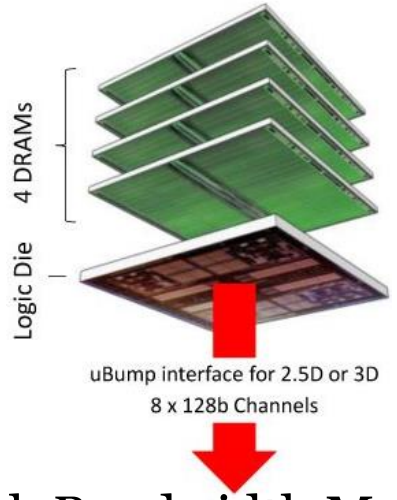
Figs: http://oceanaute.blogspot.com/2015/06/how-to-shuffle-sort-mapreduce.html

PIM & NDP

- Improve performance & energy by avoiding data movement
- Processing-In-Memory (1990's – 2000's)
  - Same-die integration is too expensive
- Near-Data Processing
  - Enabled by 3D integration
  - Practical technology solution
  - Processing on the logic die



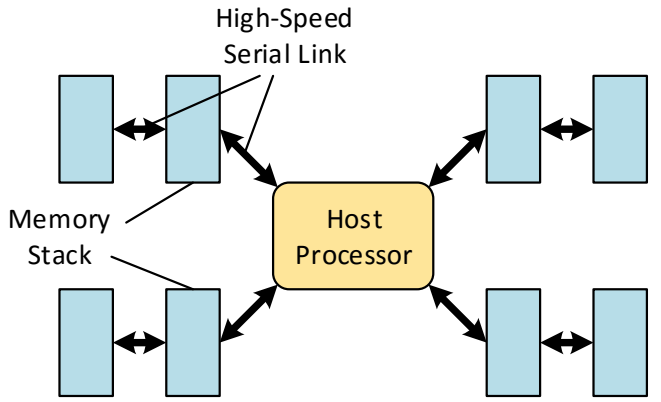
Hybrid Memory Cube (HMC)

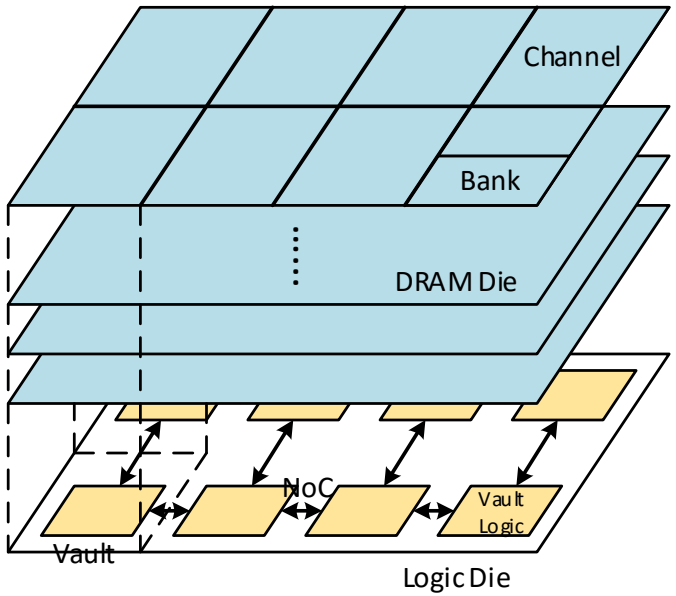


High Bandwidth Memory (HBM)

Figs: www.extremetech.com

Base NDP Hardware





- Stacks linked to host multi-core processor
  - Code with temporal locality: runs on host
  - Code without temporal locality: runs on NDP
- 3D memory stack
  - x10 bandwidth, x3-5 power improvement
  - 8-16 vaults per stack
    - Vertical channel
    - Dedicated vault controller
  - NDP cores
    - General-purpose, in-order cores
    - FPU, L1 caches I/D, no L2
    - Multithreaded for latency tolerance

Challenges and Contributions

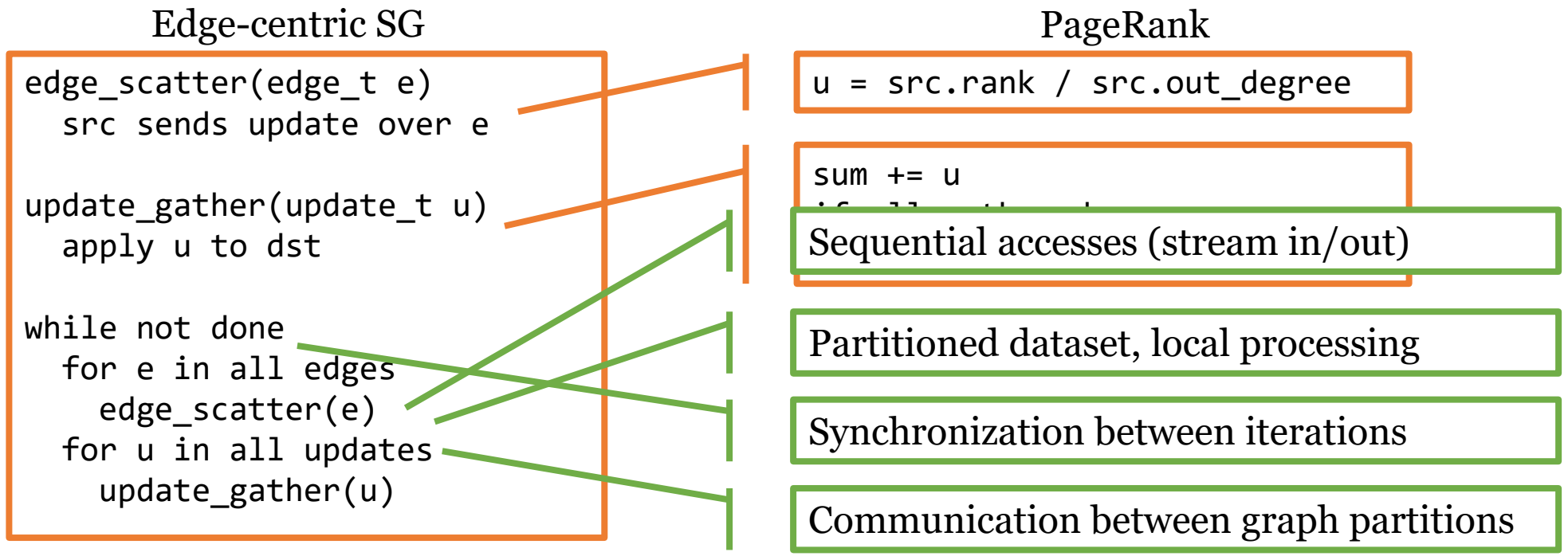
Stanford | ENGINEERING
Electrical Engineering

- NDP for large-scale highly distributed analytics frameworks
  - ? General coherence maintaining is expensive
    - ✓ Scalable and adaptive software-assisted coherence
  - ? Inefficient communication and synchronization through host processor
    - ✓ Pull-based model to directly communicate, remote atomic operations
  - ? Hardware/software interface
    - ✓ A lightweight runtime to hide low-level details to make program easier
  - ? Processing capability and energy efficiency
    - ✓ Balanced and efficient hardware

- A general, efficient, balanced, practical-to-use NDP architecture

Example App: PageRank

- Edge-centric, scatter-gather, graph processing framework
- Other analytics frameworks have similar behaviors



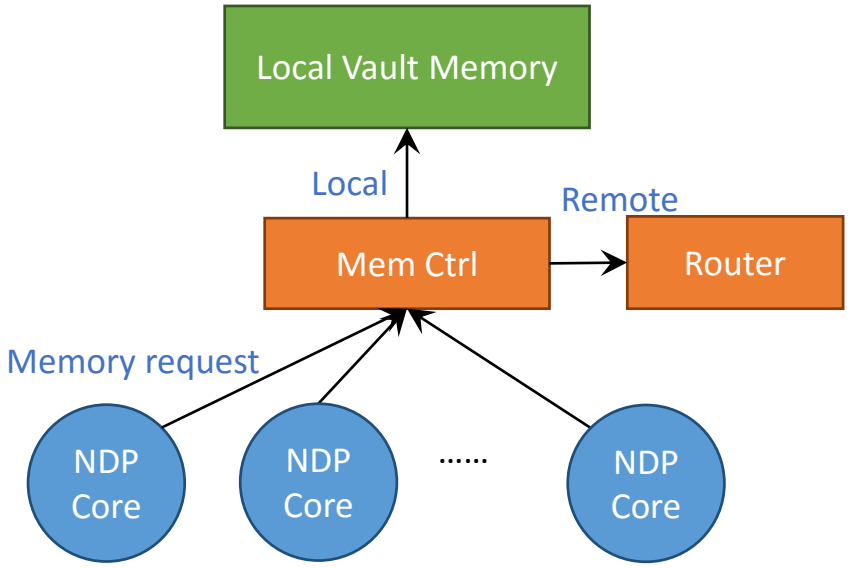
Architecture Design

Memory model, communication, coherence, …

Lightweight hardware structures and software runtime

Shared Memory Model

- Unified physical address space across stacks
  - Direct access from any NDP/host core to memory in any vault/stack
- In PageRank
  - One thread to access data in a remote graph partition
    - For edges across two partitions
- Implementation
  - Memory ctrl forwards local/remote accesses
  - Shared router in each vault



Virtual Memory Support

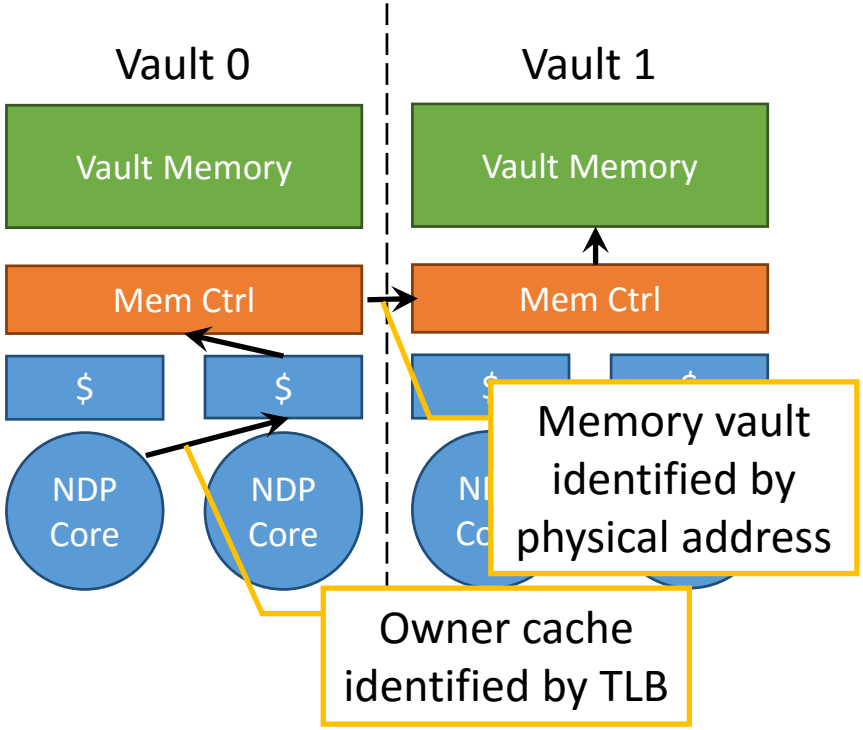
Stanford | ENGINEERING
Electrical Engineering

- NDP threads access virtual address space
  - Small TLB per core (32 entries)
  - Large pages to minimize TLB misses (2 MB)
  - Sufficient to cover local memory & remote buffers
- In PageRank
  - Each core works on local data, much smaller than the entire dataset
  - 0.25% miss rate for PageRank
- TLB misses served by OS in host
  - Similar to IOMMU misses in conventional systems

Software-Assisted Coherence

Stanford | ENGINEERING
Electrical Engineering

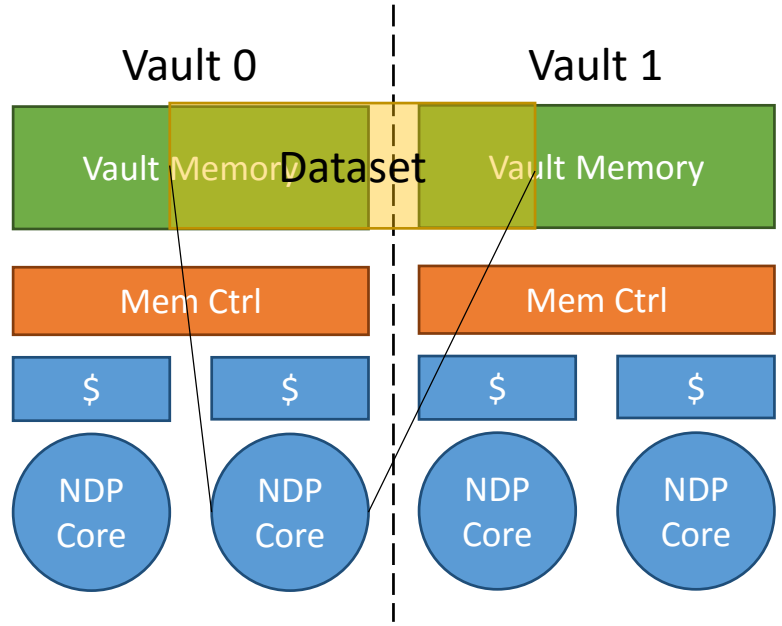
- Maintaining general coherence is expensive in NDP systems
  - Highly distributed, multiple stacks
- Analytics frameworks
  - Little data sharing except for communication
  - Data partitioning is coarse-grained
- Only allow data to be cached in one cache
  - Owner cache
  - No need to check other caches
- Page-level coarse-grained
  - Owner cache configurable through PTE



Software-Assisted Coherence

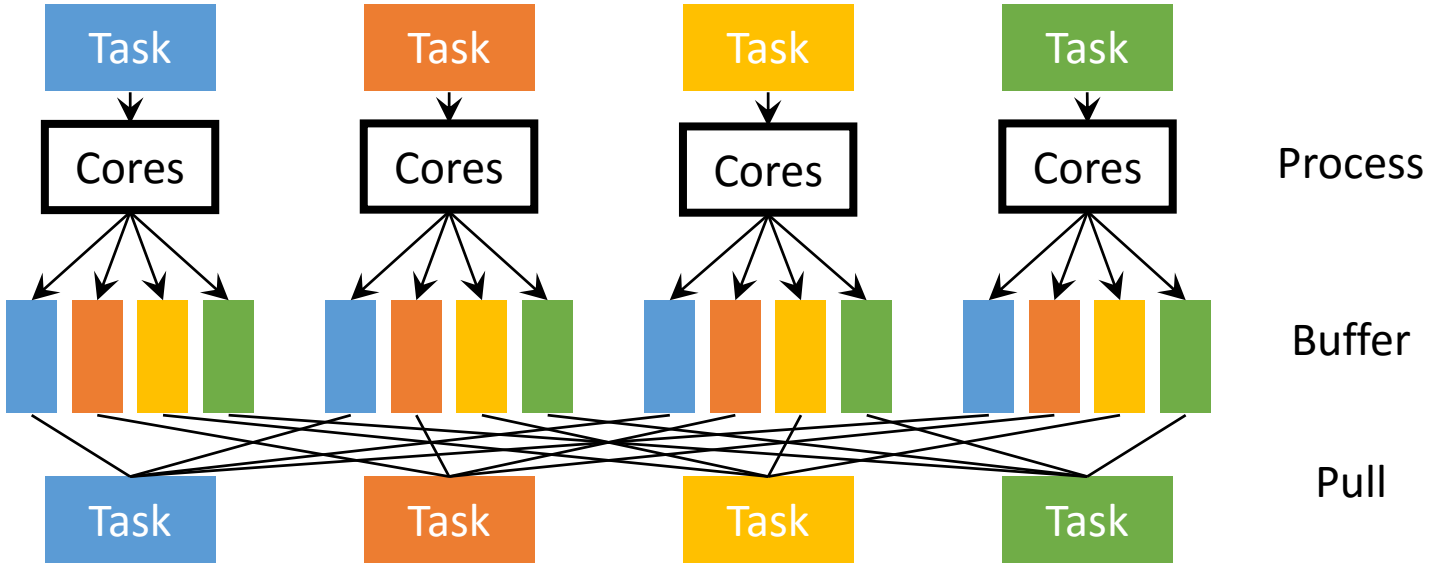
Stanford | ENGINEERING
Electrical Engineering

- Scalable
  - Avoids directory lookup and storage
- Adaptive
  - Data may overflow to other vault
  - Able to cache data from any vault in local cache
- Flush only when owner cache changes
  - Rarely happen as dataset partitioning is fixed



Communication

- Pull-based model
  - Producer buffers intermediate/result data locally and separately
  - Post small message (address, size) to consumer
  - Consumer pulls data when it needs with load instructions



Communication

- Pull-based model is efficient and scalable
  - Sequential accesses to data
  - Asynchronous and highly parallel
  - Avoids the overheads of extra copies
  - Eliminates host processor bottleneck

- In PageRank
  - Used to communicate the update lists across partitions

Communication

- HW optimization: remote load buffer (RLBs)
  - A small buffer per NDP core (a few cachelines)
  - Prefetch and cache remote (sequential) load accesses
    - Remote data are not cache-able in the local cache
    - Do not want owner cache change as it results in cache flush

- Coherence guarantee with RLBs
  - Remote stores bypass RLB
    - All writes go to the owner cache
    - Owner cache always has the most up-to-date data
  - Flush RLBs at synchronization point
    - … at which time new data are guaranteed to be visible to others
    - Cheap as each iteration is long and RLB is small

Synchronization

- Remote atomic operations
  - Fetch-and-add, compare-and-swap, etc.
  - HW support at memory controllers [Ahn et al. HPCA'05]

- Higher-level synchronization primitives
  - Build by remote atomic operations
  - E.g., hierarchical, tree-style barrier implementation
    - Core → vault → stack → global

- In PageRank
  - Build barrier between iterations

Software Runtime

- Hide low-level coherence/communication features
  - Expose simple set of API
- Data partitioning and program launch
  - Optionally specify running core and owner cache close to dataset
  - No need to be perfect, correctness is guaranteed by remote access
- Hybrid workloads
  - Coarsely divide work between host and NDP by programmers
    - Based on temporal locality and parallelism
  - Guarantee no concurrent accesses from host and NDP cores

Evaluation

Three analytics framework: MapReduce, Graph, DNN

Methodology

Stanford | ENGINEERING
Electrical Engineering

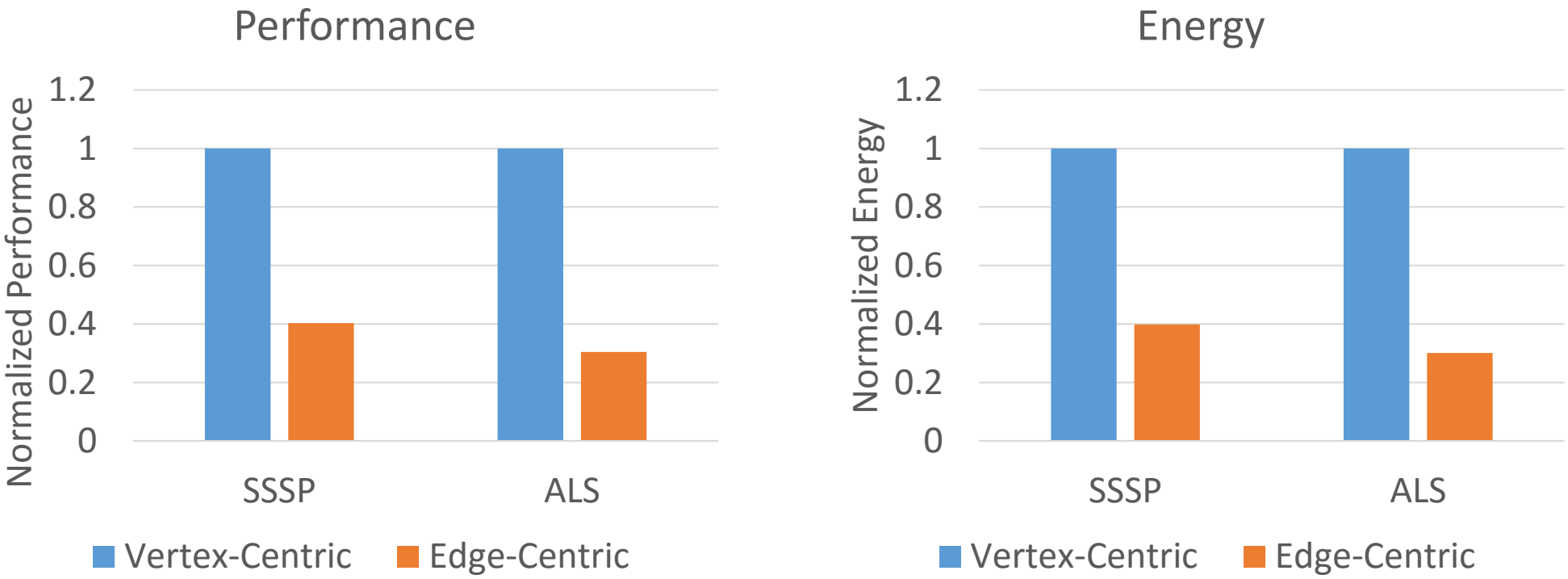
- Infrastructure
  - zsim
  - McPAT + CACTI + Micron's DRAM power calculator
- Calibrate with public HMC literatures
- Applications
  - MapReduce: Hist, LinReg, grep
  - Graph: PageRank, SSSP, ALS
  - DNN: ConvNet, MLP, dA

Porting Frameworks

Stanford | ENGINEERING
Electrical Engineering

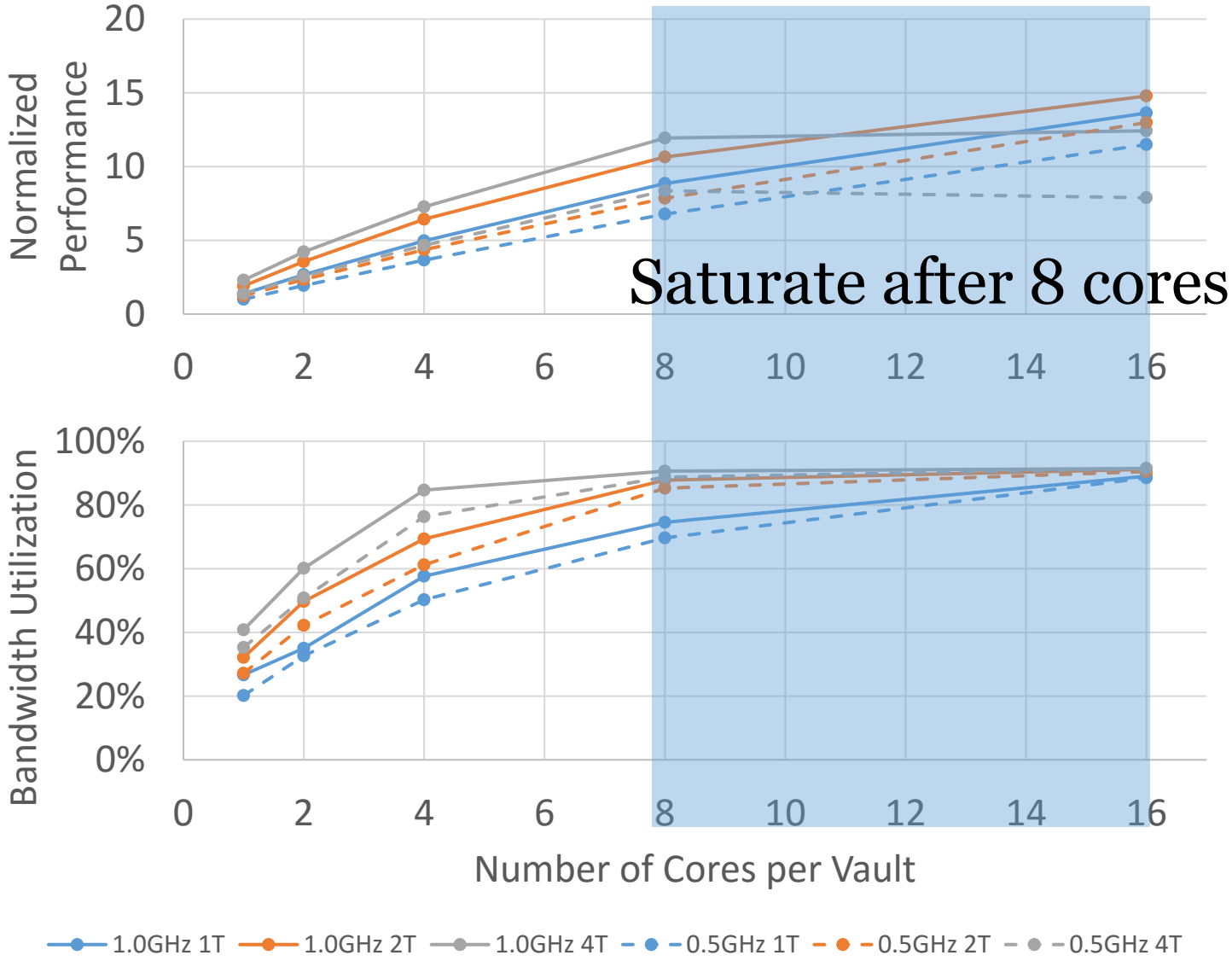
- MapReduce
  - In map phase, input data streamed in
  - Shuffle phase handled by pull-based communication
- Graph
  - Edge-centric
  - Pull remote update lists when gathering
- Deep Neural Networks
  - Convolution/pooling layers handled similar to Graph
  - Fully-connected layers use local combiner before communication
- Once the framework is ported, no changes to the user-level apps

Graph: Edge- vs. Vertex-Centric



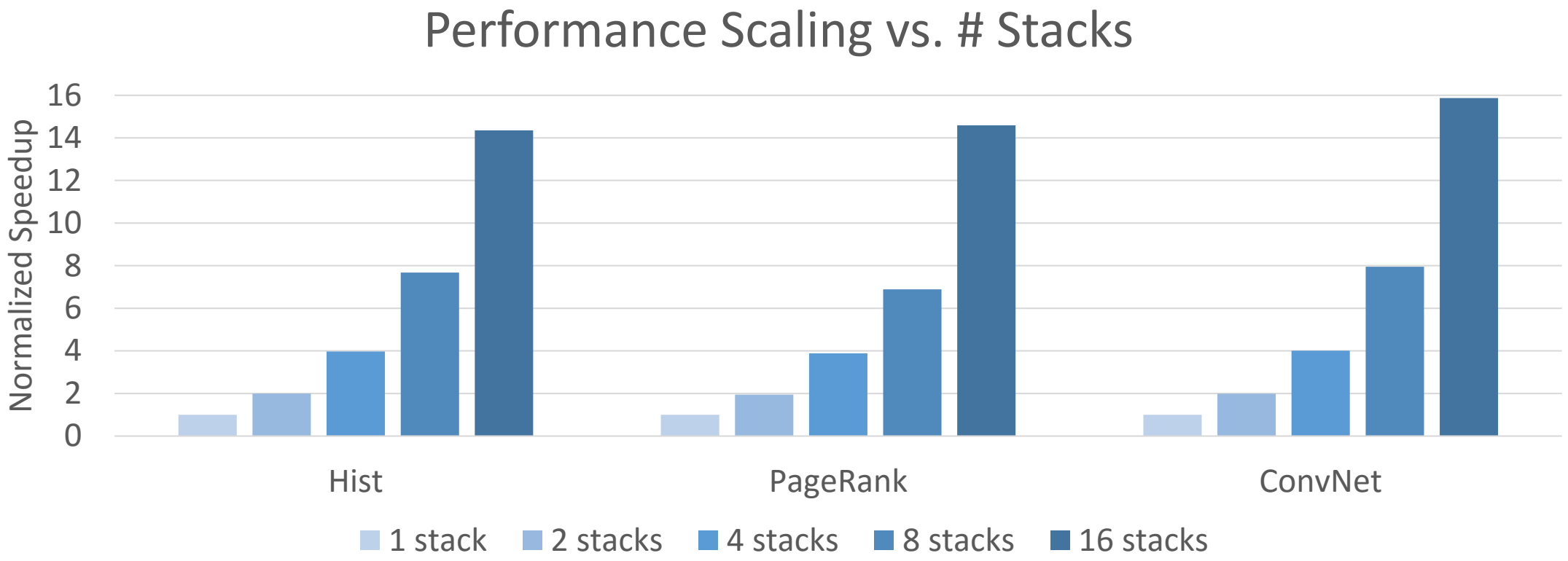
- 2.9x performance and energy improvement
  - Edge-centric version optimize for spatial locality
  - Higher utilization for cachelines and DRAM rows

Balance: PageRank



- Performance scales to 4-8 cores per vault
  - Bandwidth saturates
- Final design
  - 4 cores per vault
  - 1.0 GHz
  - 2-threaded
  - Area constrained

Scalability



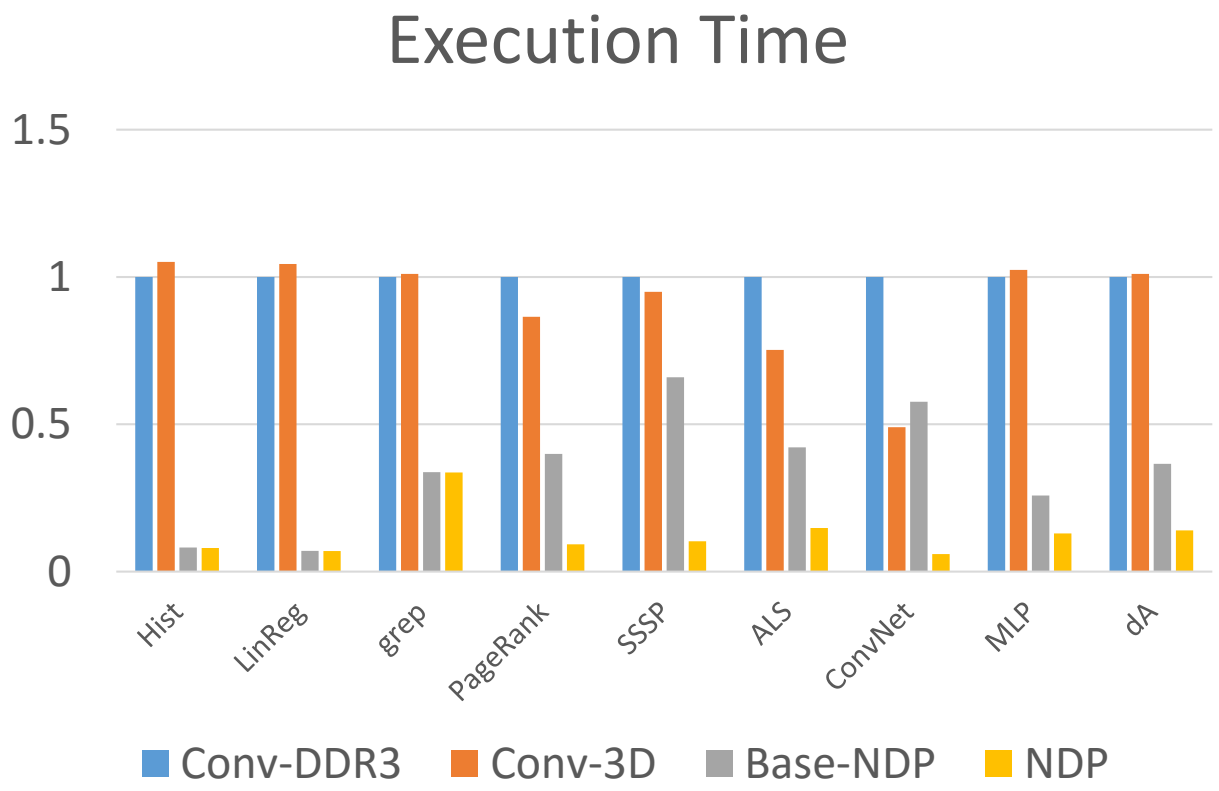
- Performance scales well up to 16 stacks (256 vaults, 1024 threads)
- Inter-stack links are not heavily used

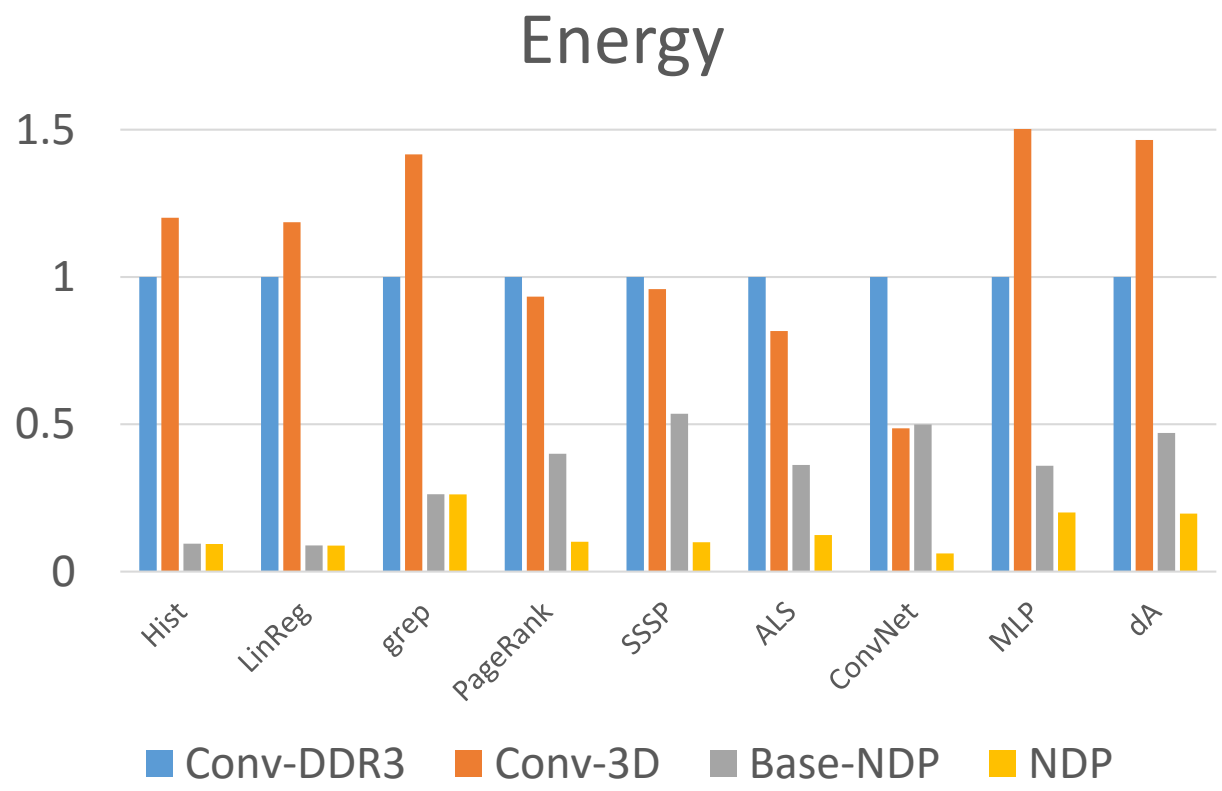
Final Comparison

- Four systems
  - Conv-DDR3
    - Host processor + 4 DDR3 channels
  - Conv-3D
    - Host processor + 8 HMC stacks
  - Base-NDP
    - Host processor + 8 HMC stacks with NDP cores
    - Communication coordinated by host
  - NDP
    - Similar to Base-NDP
    - With our coherence and communication

Final Comparison

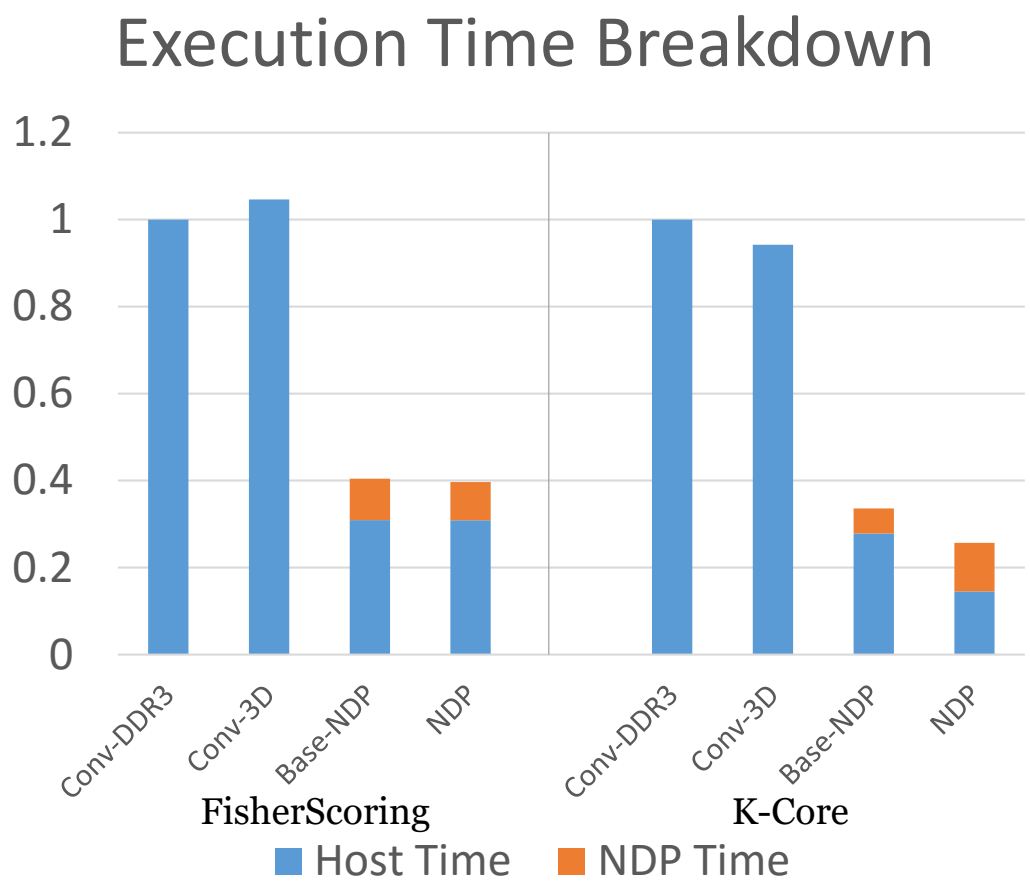
Stanford | ENGINEERING
Electrical Engineering





- Conv-3D: improve 20% for Graph (bandwidth-bound), more energy
- Base-NDP: 3.5x faster and 3.4x less energy than Conv-DDR3
- NDP: up to 16x improvement than Conv-DDR3, 2.5x over Base-NDP

Hybrid Workloads



- Use both host processor and NDP cores for processing
- NDP portion: similar speedup
- Host portion: slight slowdown
  - Due to coarse-grained address interleaving

Conclusion

Stanford | ENGINEERING
Electrical Engineering

- Lightweight hardware structures and software runtime
  - Hides hardware details
  - Scalable and adaptive software-assisted coherence model
  - Efficient communication and synchronization
- Balanced and efficient hardware
- Up to 16x improvement over DDR3 baseline
  - 2.5x improvement over previous NDP systems
- Software optimization
  - 3x improvement from spatial locality

Thanks!

Questions?