コンテナ型仮想化による機械学習支援を目的とした計算機環境の実現

織田康太郎*　小高知宏**　黒岩丈介**　諏訪いずみ***　白井治彦****

Realization of Containerized Virtualization of Computing Environments to Support Machine Learning

Kotaro ODA*, Tomohiro ODAKA**, Jousuke KUROIWA**
Izumi SUWA*** , Haruhiko SHIRAI****

(Received January 26, 2024)

Using container virtualization, we have developed a system that provides a computing environment capable of running machine learning to a larger number of users. Conventional GPU servers do not provide sufficient separation of the computing environment between users, and when a GPU server is shared by multiple users, many things need to be taken care of. This system provides a computing environment as a container, making it difficult for users to be affected by the operations of other users, and enabling the stable provision of computing resources such as GPUs to a larger number of users.

Key words : Kubernetes, GPGPU, Machine Learning, Deep Learning, JupyterHub

1. 緒言

近年, 深層学習は様々な分野で非常に高い注目を浴びている. これは OpenAI が 2022 年に公開した ChatGPT の影響が非常に大きいと考えられる. ChatGPT は大規模言語モデルの一種である GPT-3 を使用したチャットボットであり, 回答の正確さや柔軟性から世界的に注目されている. その注目度の高まりは 1 年以上が経過した 2023 年 12 月現在でも継続している.

このように近年の深層学習は大規模化の流れにあるが, 同時にこれらのモデルをより低性能な計算機で学習できるようにしようという流れも存在する. 例えば,2022 年 8 月にオープンソース化し一般向けに公開され, 非常に高い注目を集めている画像生成モデルである Stable Diffusion は, 厳密には条件によって異なると思うが,GPU の VRAM が 12GB 以上であれば動作するため家庭用の GPU でも画像の生成を行うことが可能である. Stable Diffusion は Stability AI が提供する画像生成モデルである. 主に txt2img は Text to Image の略であり, 文章から画像を生成することが可能であるという特徴から誰でも簡単に利用できる. 更に img2img は Image to Image の略であり, 画像から画像を生成するという場合においても, 最低 1 枚の画像から新たに画像を生成することが可能であるなど, こちらも非常に簡単に利用できる. また,txt2img と img2img のどちらも様々パラメータの調整を行うことで細かな修正を行うことができ, それらのパラメータの修正も専用の Stable Diffusion Web UI を利用することで行いやすくなっている. こういった理由から Stable Diffusion は公開直後から非常に高い人気があった. 最も簡単に GPU を利用できる環境の 1 つである Google Colaboratory では, かつて無料で NVIDIA T4 という GPU が割り当てに失敗することもあるが, 基本的に無制限に利用できた. この NVIDIA T4 は GPU の VRAM 容量が 16GB あり, Stable Diffusion を実行する

[*1]大学院工学研究科 知識社会基礎工学専攻
[*1]Fundamental Engineering for Knowledge-Based Society, Graduate School of Engineering
[**]知能システム工学講座
[**]Department of Human and Artificial Intelligent Systems
[***]仁愛女子短期大学 生活科学学科
[***]Jin-ai Women ’s College
[****]工学部 技術部
[****]Technical Division

ことができたため, 画像生成が非常に多く行われた. そのため,2022 年の 9 月 30 日以降, Google Colaboratory の有料プランで従来の優先的に上位のランタイムや GPU を割り当てるという定額で使い放題という形態から, 毎月一定のコンピューティングユニットが割り当てられるという形態に変わった. 使用するランタイムや GPU によって消費するコンピューティングユニットは異なるが, 使い切ると追加でコンピューティングユニットを購入しない限り, それ以上 GPU の利用ができなくなった. さらに,2023 年の 4 月に Google Colaboratory のプロダクトリーダーである Chris Perry が「Stable Diffusion Web UI の利用量が非常に多くなり, Google Colab チームの予算では対応できなくなった」という趣旨の投稿を旧 Twitter（現 X）にて行っており, そのあと Google Colaboratory の無料プランでは, Stable Diffusion Web UI の利用に制限がかかるようになった.

以上をまとめると, 生成 AI の登場によって深層学習に対する注目度はこれまで以上に高まっているが, それを実行するために大きな企業などのプラットフォーマーが提供する計算機環境は突然の料金プランの変更などを受け, 利用方法を変える必要が生じることがある. そういった事態を避けるために最も効果的な方法はやはり自前の計算機環境を用意することであり, 本研究では主に家庭用 GPU を搭載した複数のコンピュータを利用することでユーザーがより安定して機械学習や深層学習を実行できる計算機環境提供システムの実現を目指す. また, ユーザーの利用に際して, より機械学習や深層学習の実行を支援するために, コンテナ型仮想化を使用した計算機環境の生成を行う.

2. 機械学習と計算機環境

本章では, 機械学習と計算機環境の関係について, 歴史を踏まえつつ説明する.

2.1 機械学習とは

一言に機械学習と言っても, この言葉の持つ意味は回帰分析や帰納学習, 進化的計算など多数あるが, 今回はその中でもニューラルネットワークを使用した手法について説明する.

ニューラルネットワークとは, 生物の神経細胞によるネットワークをモデル化したものとして人口ニューラルネットワークとも呼ばれる. ニューラルネットワークは, ニューロンという神経細胞をモデル化したものを複数組み合わせたものとなる. ニューロンは主にパーセプトロンを使ってモデル化される. パーセプトロンとは, 複数の入力に対して一つの出力を行う関数である. このパーセプトロンを用いることで, 線形分離可能な問題は学習することが可能である. しかし, 一つのパーセプトロンでは排他的論理和のような線形分離することができないものは学習することができないという問題がある. 複数のパーセプトロンを用いてニューラルネットワークを構成することで, 線形分離不可能なものであっても学習が可能となる. 実際に排他的論理和は 3 つのパーセプトロンを使って 2 層のニューラルネットワークを構成することで学習可能である. つまり, ニューラルネットワークは層を重ね, ニューラルネットワークを構成するパーセプトロンの数を多くすることでより複雑な問題にも対応することが可能であると言える. このように複数のパーセプトロンを使って 2 層以上のニューラルネットワークを構成したものを多層パーセプトロンとも言う. 余談ではあるが, 反対に一つのパーセプトロンのみにより構成されるニューラルネットワークを単純パーセプトロンという.

単純パーセプトロンよりも多層パーセプトロンの方がパラメータ数が多く, より複雑な問題に対応できるということから非常に注目度は高かったが, 単純に多層パーセプトロンの一層あたりのパーセプトロンの数を増やしたり, 層の数を増やしたりなどしても学習がうまくいかないという問題が発生した. それが, 局所最適化と勾配消失問題である. 局所最適化は最適化分野の問題とも言える問題である. 具体的には, 多層パーセプトロンの中間層や学習データを増やしていくと, 学習データと出力との誤差を減らす際に, 局所最小値にはまってしまうという問題である. この現象は過学習とも言われる. 勾配消失問題は活性化関数であるシグモイド関数によって学習データと出力の誤差を減らそうとしても中間層を増やすほど誤差の勾配がシグモイド関数の端によってしまい, 誤差の減少が思うように進まなくなるという問題である. これらの問題によって多層パーセプトロンは機械学習手法としてあまり注目されなくなった.

2.2 深層学習と GPGPU の活用

多層パーセプトロンによるニューラルネットワークを用いた機械学習に転機が訪れたのは, 局所最適化と勾配消失問題を解決する方法として, ヒントンがオートエンコーダとディープビリーフネットワークを提案した 2006 年である. これらの方法によって中間層が 2 層以上であっても学習が継続できるようになった. そのため, これ以降の 3 層以上の多層パーセプトロン, つまりディープニューラルネットワークを使った学習における問題点は計算量の多さと学習データの収集になった. ディープニューラルネットワークは層と学習データを増やせば増やすほど性能が向上すると言えるので, それ

らの計算を効率よく行うことが次の課題になるからである. その問題に対する有効な手段は 2012 年に開催された画像認識 AI の競技「ImageNet Large Scale Visual Recognition Challenge」(ILSVRC)[1] においてトロント大学のアレックスらが,2 基の GPGPU による学習で平均的な人間の正答率を超える結果を出したことで示された. この時に行われた学習には畳み込みニューラルネットワークが使われている. 畳み込みニューラルネットワークとは, 畳み込み演算を加えたニューラルネットワークの総称であり, ディープニューラルネットワークが DNN と呼ばれるように CNN と呼ばれる. これは畳み込み演算が英語で,「Convolution」というためである. 畳み込み演算とは, 与えられた 2 つの数列の要素同士を全組合せで乗算して加算する計算のことである. 例えば画像であれば,1 つの画素だけ見ていてもそれがなんであるかわからないため, 周囲の画素との関係を見ることが重要である. 畳み込み演算はそういった状況において 5x5 の範囲の特徴などを計算する際に有効な演算である. CNN を使った学習ではこの畳み込みの計算が学習過程で大量に行うことになり, その計算には非常に時間がかかる. そこで GPGPU を使用することが重要になる. GPGPU とは「General-purpose computing on graphics processing units」の略称であり, 日本語に訳すと「GPU による汎用的な目的における計算」となる. GPU とは元々画像処理に使われていた計算装置であるが, それを画像処理以外に利用する技術がこの GPGPU である. 畳み込み演算は, 行列同士の掛け算が主な処理内容であり, この計算は GPU が画像処理用に持っている持つプログラマブルシェーダー実行装置によってそのまま計算可能である. つまり,GPU を GPGPU として活用することで, 大量のデータの畳み込み演算を高速に実行可能になり,CNN による学習を促進することになった.

3. 計算機環境提供システムの構築

本章では, コンテナ仮想化を使った機械学習環境の構築について, ハードウェアやソフトウェアなどをどのように使用したか具体的に述べる. 主に家庭用 GPU を搭載した複数のコンピュータを利用することでユーザーがより安定して機械学習や深層学習を実行できる計算機環境提供システムの構築を行う. また, ユーザーの利用に際して, より機械学習や深層学習の実行を支援するために, コンテナ型仮想化を使用した計算機環境の生成を行う.

3.1 ハードウェアの準備

今回は 2 つの研究室で別々に複数のコンピュータを使用した. それぞれの研究室で使用したコンピュータのスペックは表 1 と表 2 の通りである.

これらのコンピュータを用いてクラスタリングを行うことで機械学習用の計算機環境を用意する. どのようにクラスタリンを行うかは 3.3 節に記載する.

3.2 コンテナ型仮想化

本研究では, コンテナ仮想化を用いて計算機環境を用意する. コンテナ型仮想化はほかの仮想化技術とどのように異なるのか, 具体的にどのようなコンテナ型仮想化技術を用いたのかなどについて述べる.

まず,IT 分野における仮想化とは論理的にハードウェアなどのリソースを分離したり, 複数のハードウェアで共有したりする技術を指す言葉である. 仮想化を利用することで,1 台のコンピュータで複数の異なる OS を動作させることが可能となったり, 別のコンピュータに OS などのコンピュータの動作に必要なデータを移すことで OS を停止させずにコンピュータの停止を行うライブマイグレーションが可能となったりする. 仮想化技術にはハイパーバイザー型とコンテナ型の 2 種類が存在する.

- ハイパーバイザー型仮想化技術

  ハイパーバイザー型は物理コンピュータにインストールされた OS の上で仮想ハードウェアをエミュレーションする. そのため, ホストとなっている物理コンピュータで動作可能な OS であれば仮想ハードウェアにインストールすることができる. つまり,1 台のコンピュータ上で Windows や Linux など多種多様な OS を同時に動作させることが可能となる. VMware,ProxmoxVE,KVM などがハイパーバイザー型仮想化ソフトウェアとして挙げられる.

- コンテナ型仮想化技術

  コンテナ型はホストマシンの OS とコンテナで動作するゲスト OS とでカーネルを共有するため, ハイパーバイザー型と比較してリソースのロスが少なく, リソースを最大限活用することが可能であるが, カーネルを共有するという性質上, ゲスト OS として活用することのできる OS には制限があり, カーネルのアップデートもゲスト OS では行うことができないという制限もある. 代表的なソフトウェアとして Docker が挙げられるが,Docker 以外にも,containerd や cri-o,podman,runc などがある.

今回の環境構築ではクラスタリングに Kubernetes を

表 1 研究室 A

	A-01	A-2	A-3
CPU	Intel Core i5 10400	Intel Core i7 13700	AMD Ryzen 7 1800X
Memory	DDR4 16GB	DDR4 48GB	DDR4 16GB
Storage	HDD 6TB	SSD 512GB	SSD 256GB + HDD 1TB
GPU	(none)	NVIDIA GeForce RTX 4080 NVIDIA A2	NVIDIA GeForce GTX 1080 Ti

表 2 研究室 B

	B-0	B-1	B-2	B-3	B-4
CPU	Intel Core i5 13400	Intel Core i7 3770	AMD Ryzen 7 1800X	AMD Ryzen 5 2600X	Intel Core i7 10700
Memory	DDR5 16GB	DDR3 16GB	DDR4 16GB	DDR4 16GB	DDR4 64GB
Storage	SSD 512GB	HDD 2TB x4 (RAID10)	SSD 256GB	SSD 256GB	SSD 1TB
GPU	(none)	(none)	NVIDIA GeForce GTX 1080 Ti	NVIDIA GeForce RTX 2080 Ti	NVIDIA GeForce RTX 3070

利用している関係上,containerd をコンテナソフトウェアとして利用する.

3.3 Kubernetes によるクラスタリング

今回は計算機環境をユーザーに対して提供するにあたって, クラスタリングを行うことにする. クラスタリングを行う利点は大きく 3 点ある.

1. ユーザーが複数のコンピュータを一つのサービス, 例えば Web ページから選択して利用することが可能になる.
2. クラスタにコンピュータを追加することで計算機の数を拡張できる.
3. クラスタを構成するコンピュータの役割を分けることで負荷分散ができる.

Kubernetes をクラスタリングソフトウェアとして選んだ理由は,1 つ目と 2 つ目の利点を実現するためである.

Kubernetes は複数のコンテナを管理するためのソフトウェアであり,JupyterHub を Kubernetes にインストールすることでユーザーは Web ブラウザ上の操作によって Jupyter Notebook のコンテナを任意のコンピュータに作成し, 利用することが可能となる.

また, クラスタの拡張については,Kubernetes は後からクラスタにコンピュータを追加しても, それを計算用コンピュータとして利用するために必要なソフトウェアをコンテナとして自動で作成し機能を追加する仕組みを持っているので, 計算機環境提供システムを必要に応じてスケールアウトすることが非常に容易である.

そして,3 つ目の利点を実現するためにクラスタリングを行うコンピュータの数を最低でも（計算用コンピュータの数+1）台となるようにした. これはクラスタの管理を行うコンピュータとデータを保存するコンピュータを計算用コンピュータとは別に用意するためである. この役割の分離についても Kubernetes を使用することである程度容易に行うことができる. また, できればクラスタの管理とデータ保存のコンピュータも分けたほうが無難ではあるため, コンピュータの数に余裕があった研究室 B ではクラスタに使用したコンピュータの数を（計算用コンピュータの数+2）台としている.

ここで表 1 と表 2 について補足すると, それぞれのコンピュータの名前について, 最初のアルファベットがどの研究室のものであるかを示しており,「-」の後に続く数字がそのコンピュータのクラスタにおける役割を示している. 具体的には「0」がクラスタの管理を行うコンピュータであり,「1」がデータを保存するコンピュータ,「2」以降が計算用のコンピュータである. つまり, 表 1 の研究室 A では,A-01 がクラスタの管理とデータ保存の両方を担っている.

3.4 JupyterHub によるノートブックコンテナの提供

JupyterHub は Jupyter Notebook を作成し, ユーザーに提供するための Web アプリケーションである. Jupyter Notebook は IPython[2] をブラウザから利用することを可能にする.

仕組みを簡単に説明すると,JupyterHub はスポナーによって Jupyter Notebook 環境を提供する. 動作する環境によってスポナーの種類は異なり,Kubernetes 環境では専用のスポナーである KubeSpawner が動作している. KubeSpawner にはリソース制限をかけたり,GPU などを利用したり, コンテナのもとになるコンテナイメージを設定したりするための設定を行うことが可能となっている. ユーザーが HTTP Proxy 経由で Hub にアクセスし Jupyter Notebook 環境の作成を要求すると,KubeSpawner

はコンテナ上で動作する Jupyter Notebook をユーザーが指定したコンピュータ上に作成するという処理の流れで計算機環境が環境が構築されてる.

このような機能を持つ JupyterHub であるが, 今回計算機環境の提供にあたって選択した理由は主に下記の 2 つである.

1. ノートブック環境をブラウザ上で作成し, 利用できる使いやすさ.
2. コンテナイメージを追加できる拡張性.

まず,1 つ目の理由について説明すると,JupyterHub は JupyterHub によるノートブック環境の構築と構築したノートブック環境の利用をブラウザ上で完結することができ, システム全体を通してユーザーが利用しやすいようにシンプルな操作だけで使うことが可能となっている. 例えば, コンテナに必要なパッケージをインストールしたいという場合でも, ブラウザ上の操作でインストール操作を完結することが可能である.

次に 2 つ目の理由について説明すると,JupyterHub にはノートブックコンテナを作成するためのコンテナイメージを複数登録することが可能である. しかし, それらのコンテナイメージは Jupyter 公式によって提供される特定のスクリプトを持っている必要がある. ただ, それらのスクリプトや公式の提供するイメージのもとになっている Dockerfile は GitHub[3] で公開されており, それらのファイルを GitHub に登録されている Dockerfile を参考にしてコンテナイメージ内部にコピーすれば, オリジナルのコンテナイメージを作成することが可能となる. つまり, 必要に応じてコンテナイメージを自作することが可能である. コンテナイメージを自作することのメリットは多数あり, 例えば,Jupyter 公式が提供するノートブックコンテナのもとになるコンテナイメージに含まれる Tensorflow では GPU が利用できないが,Tensorflow のインストールや深層学習に必要なライブラリ類や GPU ドライバー類を内包したコンテナイメージを自作することで GPU の利用が可能となっている. また, これらの作業は先に述べたようにコンテナ作成後にブラウザからの操作で行うことも可能といえば可能であるが, 非常に時間のかかる作業である. そういった際に, 予め GPU の利用に必要なソフトウェアを持ったコンテナイメージからコンテナを作成することで時間の節約につながるという利点もある. ただし, そういった大きなファイルを含んだコンテナイメージは必然的にそのファイルサイズも大きくなるので, どこまで事前にソフトウェアをインストールしておくかは場合によるとしか言えない. 今回は 3 つのオリジナルイメージを作成した.

- **Simple Container Image**：Jupyter 公式のイメージのベースイメージを通常の Ubuntu から NVIDIA が提供する CUDA 関係のファイルを含んだものに変更してビルドし直したコンテナイメージ.
- **AutoML Container Image**：上記の Simple Container Image に AutoML 関係のライブラリをインストールしたコンテナイメージ.
- **Multi Kernel Container Image**：Jupyter Kernel として複数のプログラミング言語をノートブックから実行できるようにしたコンテナイメージ.

以上の理由から今回は JupyterHub を利用することとした. また今回は,Kubernetes に JupyterHub などアプリケーションをインストールするにあたって Helm という Kubernetes 用パッケージマネージャを利用した.

3.5 KubeSpawner のカスタマイズ

JupyterHub にはプロファイルリストとして複数のコンテナ実行条件を設定できる. 例えば,「CPU やメモリ,GPU をどのように利用するか」,「コンテナのもととなるイメージには何を使用するか」などである. これによって事前に作成したオリジナルイメージをもとにノートブックコンテナを作成する選択肢を追加することが可能となるが, それと同時にユーザーが利用するリソースの利用量の予測しづらくなり, 結果としてリソース超過が発生する恐れが高まることが考えられる.

今回のシステム中でリソース超過が発生する状況には以下の 2 つが考えられる. なお, ここで言う「リソース」は CPU コア数やメモリ容量などのコンピュータリソースを指す.

- コンテナが要求した最低リソースの合計が実際にコンピュータ上にあるリソースを上回る.
- コンテナに設定されたリソース制限の最大までリソースを利用するコンテナが複数あったり, そもそもリソースを制限されていないコンテナが膨大な量のリソースを使用したりすることでリソース使用量が実際のコンピュータ上にあるリソースを上回る.

通常のシステムであれば, 少なくとも前者の原因については発生しないようにされているのが通常であると思われるが, 今回使用した JupyterHub では特にそういったリソース超過の検知機能などはなかったので, リソースが不足していてもノートブックコンテナの作成を行い, 作成に失敗するという現象が発生していた. 問題は JupyterHub がノートブックコンテナの作成に失敗して

も, それを失敗と認識してユーザーが再度ノートブックコンテナの作成を行うことができるようになるまでに 5〜10 分とかなりの時間がかかることである. また, 適切なエラーメッセージを表示して, 別のもっと低いリソースでノートブックコンテナの作成を促すような機能もないため, ユーザーが再度同じリソースでノートブックコンテナの作成を要求し, 同じ問題に何度も遭遇するという現象も考えられる.

今回はそういった問題を回避するために JupyterHub の KubeSpawner の一部をオーバーライドすることでカスタマイズした CustomSpawner を実装した.

CustomSpawner の機能は次の 2 つである.

- リソース超過の検知
- リソース超過発生の恐れのあるコンテナイメージの不可視化

前者の機能は図 1 のように Kubernetes クラスタのノードのキャパシティとコンテナが要求するリソースの差を調べ, その差よりも大きなリソースを要求していないかを調べる機能である. 後者の機能は前者の機能を利用し, プロファイルリストに登録されているすべてのコンテナ実行条件を調べて, リソース超過が発生すると検知されたものはユーザーから選択できないように不可視化するという機能である.

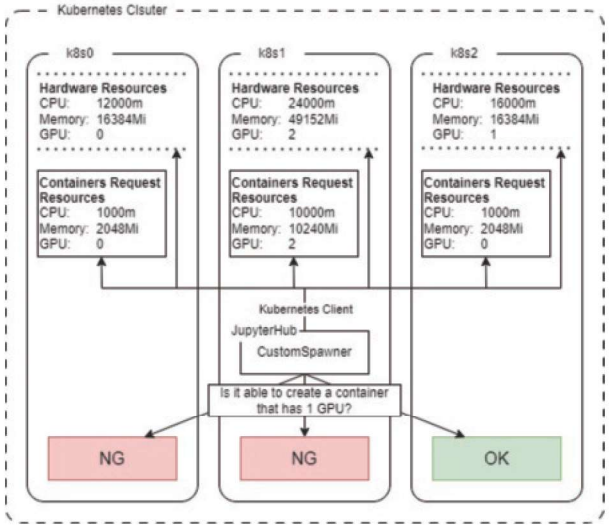

図 1 リソース超過検知の概要

これらの機能によって JupyterHub によるノートブックコンテナの作成においてリソース超過をある程度抑制することができた. また, リソース超過が発生する原因の 2 つ目については, プロファイルリストに登録する際に, コンテナが要求する最低リソースと, コンテナが利用できる限界リソースの差を小さくすることである程度発生を抑制できるのではないかと考え, それらの値の差を 1 から 1.5 倍くらいの値になるように設定した.

4. 計算機環境の性能計測と効果検証

構築した計算機環境提供システムに対していくつかのベンチマークツールを使用して性能の計測を行う. 計測は「段階的にリソース制限をかけた際の性能変化」と「同一コンピュータ上で同時にベンチマークツールを複数実行したときの性能変化」の 2 つを調べるために行う.

効果検証は実際にユーザーに今回構築した計算機環境提供システムを利用してもらった上でアンケートを実施することで調べた.

4.1 性能計測実験

先に述べたように 2 つの方法で性能の計測を行う. それぞれの方法について具体的に説明する.

まず,「段階的にリソース制限をかけたときの性能変化」の調査は,CPU やメモリなどのリソースに制限をかけたときに, その性能変化に GPU の性能が影響を受けて変化するのかを調べるための実験である.

次に,「同一コンピュータ上で同時にベンチマークツールを複数実行したときの性能変化」の調査は, 複数のユーザーが同時に一つの GPU を使用することを想定して, 実際にどのような性能の変化があるのかを調べるための実験である.

それぞれの結果は次のようになった. なお, ここでは表 1 における「A-3」上で実行した結果のみを載せる.

まずは「段階的にリソース制限をかけたときの性能変化」の結果を図 2, 図 3, 図 4 に載せる. 図は縦軸がメモリの容量を横軸が CPU のコア数を示している.

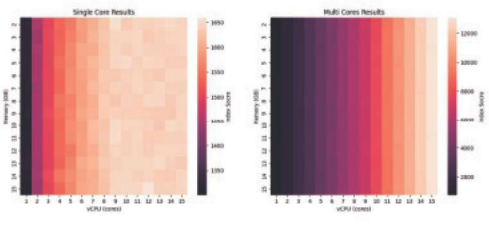

図 2 段階的にリソース制限をかけたときの性能変化 (UnixBench)

次に「同一コンピュータ上で同時にベンチマークツールを複数実行したときの性能変化」の結果を図 5, 図 6 に示す.

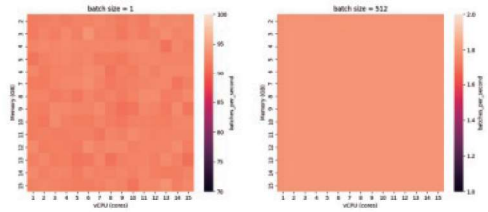

図 3 段階的にリソース制限をかけたときの性能変化（pytorch-benchmark[4]）

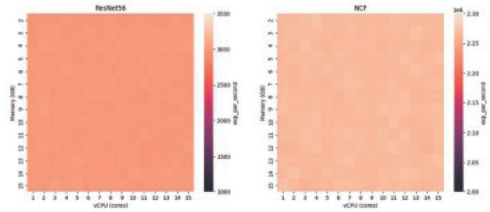

図 4 段階的にリソース制限をかけたときの性能変化（PerfZero[5]）

4.2 効果検証のための評価実験

今回構築した計算機環境提供システムとシステムによって提供された計算機環境がどの程度使えるのか調査するためにアンケートを実施する．アンケートにはシステムを全体的に使用してもらうためにいくつかの作業とユーザーの背景を知るための質問と，システムの評価に関する質問が含まれている．ここではその一部を抜粋して図 7 に示す．

5. 考察

ここでは先の実験についての考察を行う．

まず 4.1 節の「段階的にリソース制限をかけたときの性能変化」の結果は,GPU を使用しない UnixBench では顕著な差がみられたため，特に CPU のリソース制限によって計算機環境の性能は変化するということが分かった．しかし,pytorch-benchmark[4] と PerfZero[5] の結果を見るとほとんど変化していないことから，計算機環境であるコンテナの性能が変化しても GPU の動作にはそれほど大きな影響はないと考えられる．

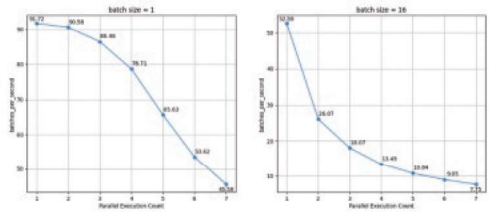

図 5 同時にベンチマークツールを複数実行したときの性能変化（pytorch-benchmark[4]）

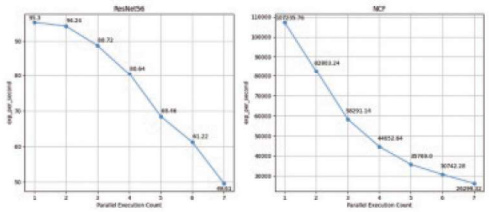

図 6 同時にベンチマークツールを複数実行したときの性能変化（PerfZero[5]）

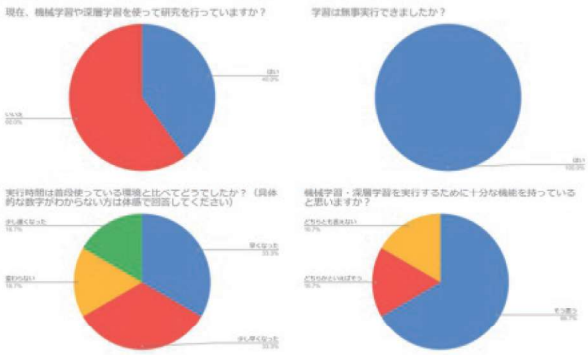

図 7 アンケート結果

次に同じ節の「同一コンピュータ上で同時にベンチマークツールを複数実行したときの性能変化」の結果は，明らかに性能が変化していることが確認できた．一度の実行が比較的短時間で終了するものと長い時間を要するものでスコアの下がり方に違いがみられたが，これはベンチマークツールを本当に同時には実行できておらず，その結果実行タイミングに多少のズレが生じているためであると思われる．つまり複数同時に処理を実行している時間が長いものほど，同時実行数とスコアが反比例のような関係を示していると考えられる．

最後に 4.2 節のアンケート結果は,4 つの結果のみを挙げているが，最初の一つがユーザーの背景調査でありアンケートに回答してくれた 15 名全員に聞いている質問である．内容は「現在深層学習を使用して研究を行っているか」であり，ここで「はい」と答えた 6 名に残りの 3 つの質問に回答してもらっている．残りの 3 つの質問は計算機環境提供システムと提供された計算機環境の機能や性能を回答者の主観を交えて回答してもらう内容となっている．3 つすべてで過半数を超える肯定的な回答を得られたことから，今回のシステムはある程度実用に耐えうる計算機環境を提供できていると考えられる．

6. 結言

本研究では, 計算機環境を提供するシステムを構築し, 多くのユーザーに安定した計算機環境を提供することを目的としている. 計算機環境提供システムは Kubernetes と JupyterHub を使用し, ユーザーのリクエストに応じてノートブックコンテナを提供するという機能を実現した.

計算機環境提供システムは複数のコンピュータ上で動作するが, 実際に計算機環境を使用するユーザーの数と計算機環境提供システムを構成するコンピュータの数を比較すると, ユーザーの数の方が多いことが想定されるため, 一台のコンピュータ上に提供される計算機環境であるノートブックコンテナを複数同時に実行できるようにした.

同時に複数の計算機環境を実行する場合, 各コンテナに対して使用可能なリソースを制限しないと, 実際に存在するリソース以上にリソースを使用しようとしてしまうため,JupyterHub のスポナーに実際に存在する以上のリソースを要求しないようにする機能を追加した. また, リソース制限がコンテナの性能にどのような影響を及ぼすのか, そのコンテナの性能変化が GPU の性能にどのような影響を及ぼすのか, 複数のコンテナが同時に GPU を使用した場合それぞれのコンテナ中の処理の性能はどのように変化するのかなどを調べた. 結果は CPU のリソース制限によってコンテナの性能は低下するが,GPU にはあまり大きな影響はないということがわかり, 複数のコンテナが GPU を同時に使用した場合は同時使用数と性能は反比例するように低下することが確認できた.

また, アンケートではおおむね実用に耐えうる計算機環境が提供できていることが確認できた.

以上のことから, 家庭用の GPU を搭載したコンピュータを複数統合管理して複数のユーザーが利用できるような計算機環境提供システムが実現できたと考えられる.

今後は, アンケートに多数寄せられたシステムの UI に関する改善や, 複数の GPU を利用することでより強力な計算機環境を提供できるような機能の実現を目指したい.

参考文献

[1] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, Vol. 115, pp. 211–252, 2015.

[2] Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in science & engineering*, Vol. 9, No. 3, pp. 21–29, 2007.

[3] Github - jupyter/docker-stacks: Ready-to-run docker images containing jupyter applications. `https://github.com/jupyter/docker-stacks/`(2024/01/23).

[4] Github - lukashedegaard/pytorch-benchmark: Easily benchmark pytorch model flops, latency, throughput, allocated gpu memory and energy consumption. `https://github.com/LukasHedegaard/pytorch-benchmark`(2024/01/24).

[5] Github - tensorflow/benchmarks: A benchmark framework for tensorflow. `https://github.com/tensorflow/benchmarks/`(2024/01/23).