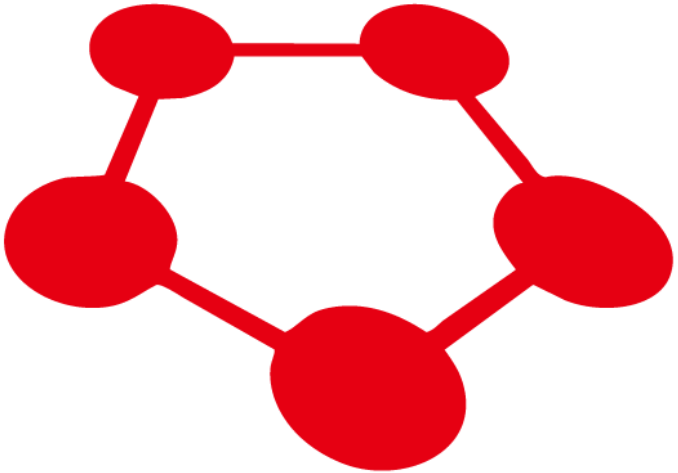
GTC Japan, 2018/09/14

得居 誠也, Preferred Networks

Chainer における深層学習の高速化

Optimizing Deep Learning with Chainer



Chainer

Preferred Networks

Chainer のミッション

Deep Learning とその応用の研究開発を加速させる

- 環境セットアップが速い
- すぐ習熟
- 素早いコーディング
- 実験の高速化
- 結果をさっと公開・論文化
- モデルをすぐ応用へ

Preferred Networks

Chainer のミッション

Deep Learning とその応用の研究開発を加速させる

- 環境セットアップが速い
- すぐ習熟
- **素早いコーディング**
- **実験の高速化**
- 結果をさっと公開・論文化
- モデルをすぐ応用へ

Preferred Networks

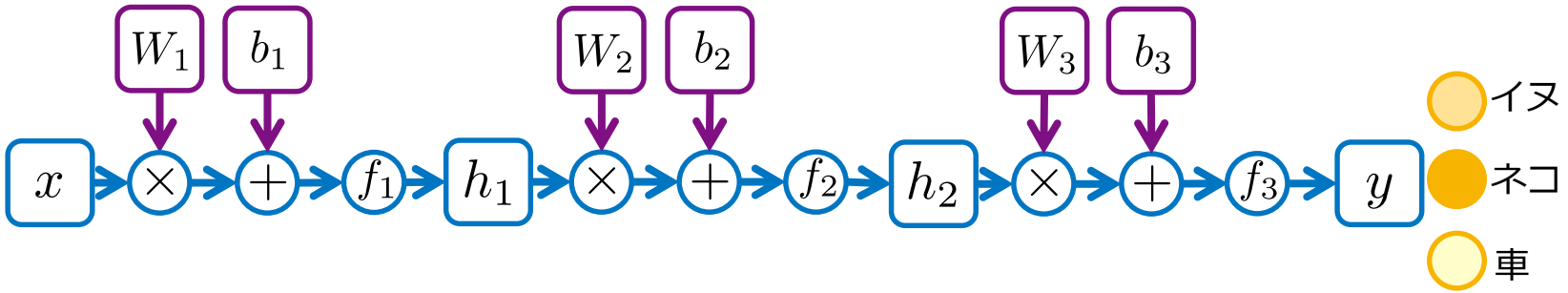
Agenda

1. Chainer の基本
2. Chainer での深層学習の高速化

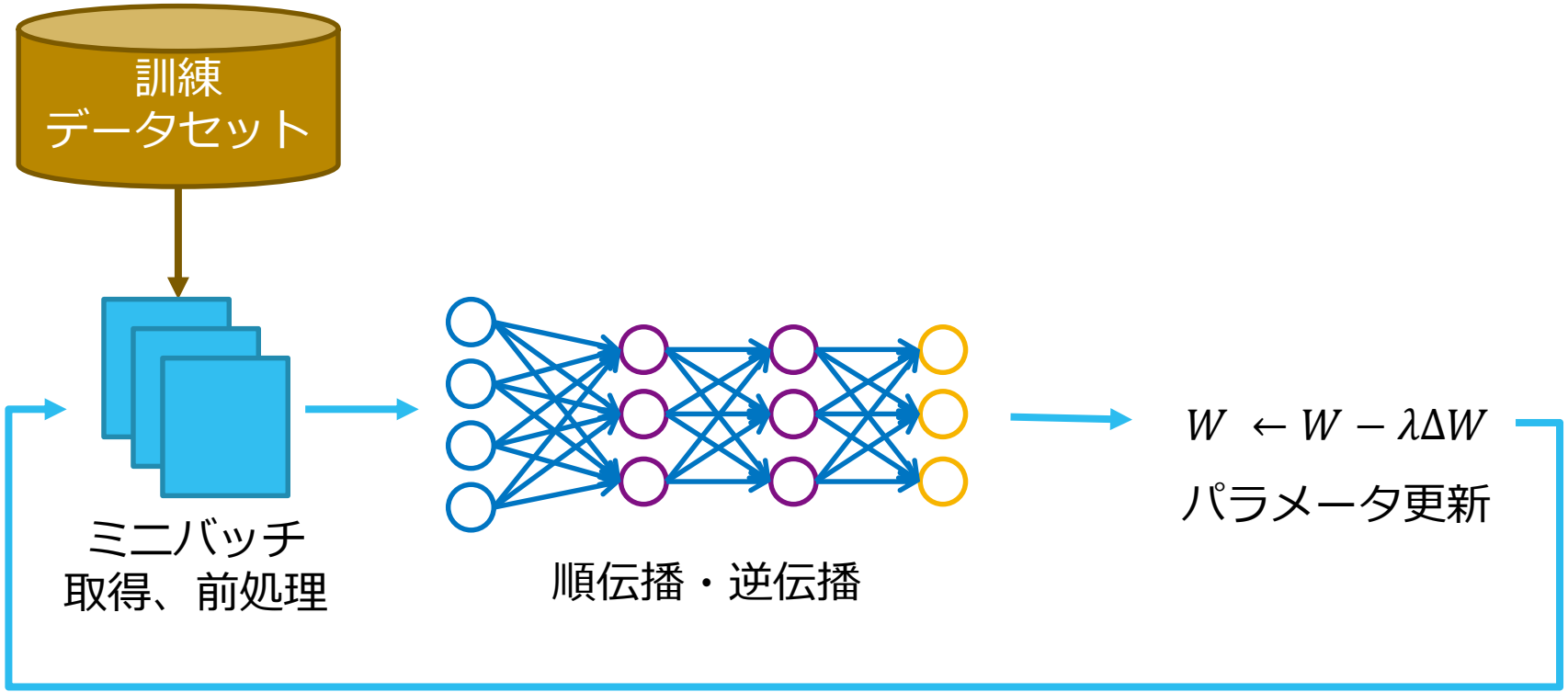
1. Chainer の基本

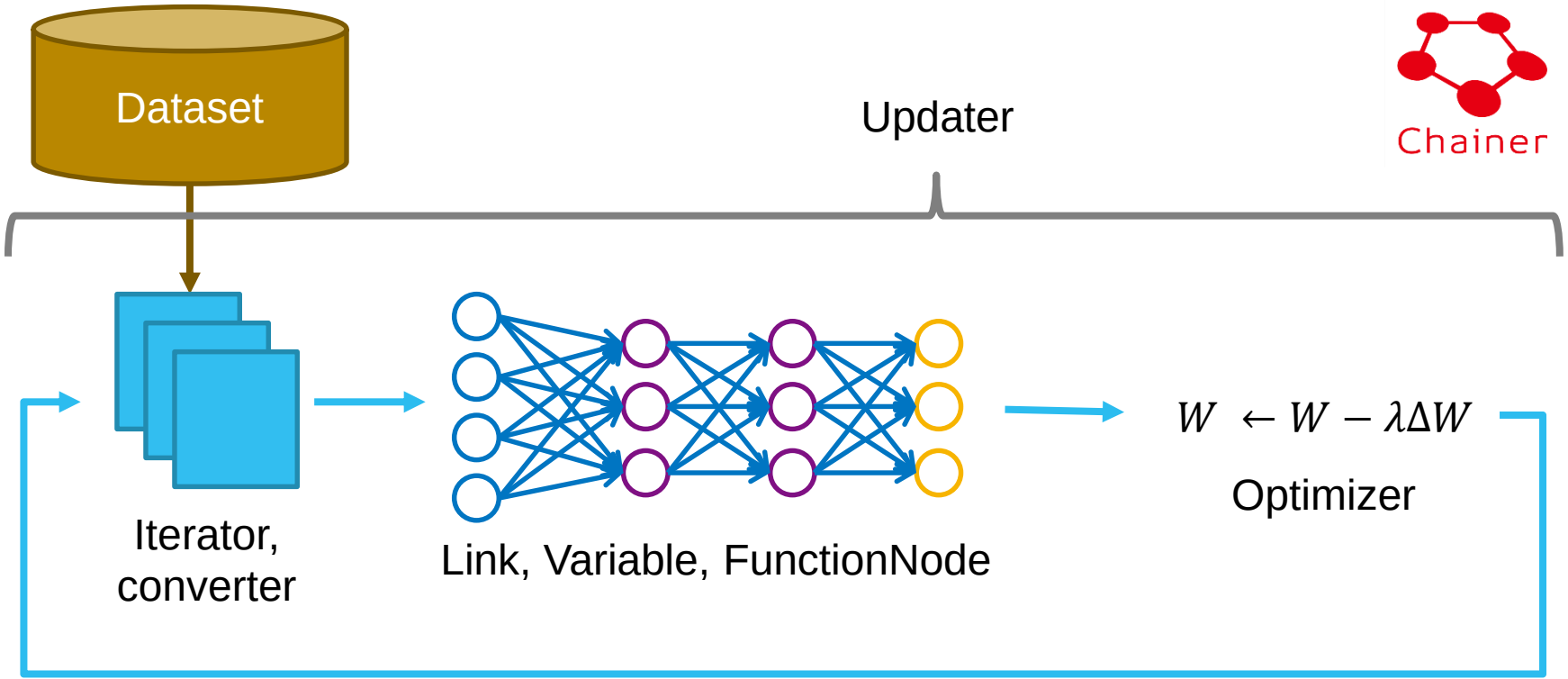
Preferred Networks

…の前に Deep Learning の基本

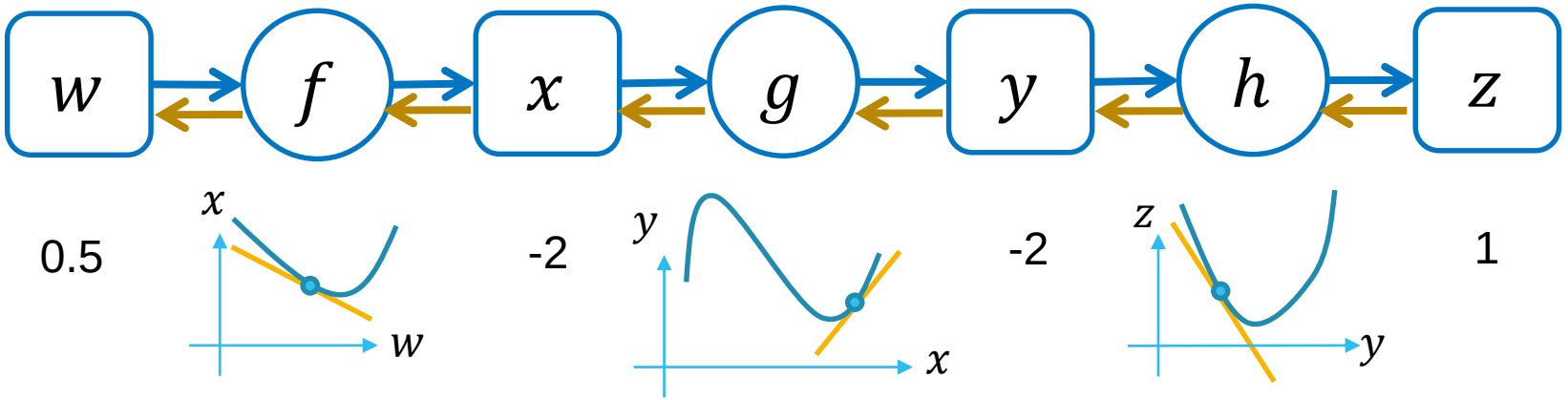


入力とパラメータから予測を計算する**プログラム**を記述して、予測が合うようにパラメータを**学習**する


訓練
データセット
ミニバッチ
取得、前処理
順伝播・逆伝播
$W \leftarrow W - \lambda \Delta W$
パラメータ更新


Dataset
Updater
Chainer
Iterator,
converter
Link, Variable, FunctionNode
$W \leftarrow W - \lambda \Delta W$
Optimizer

誤差逆伝播



誤差逆伝播の実装方法: Define and Run

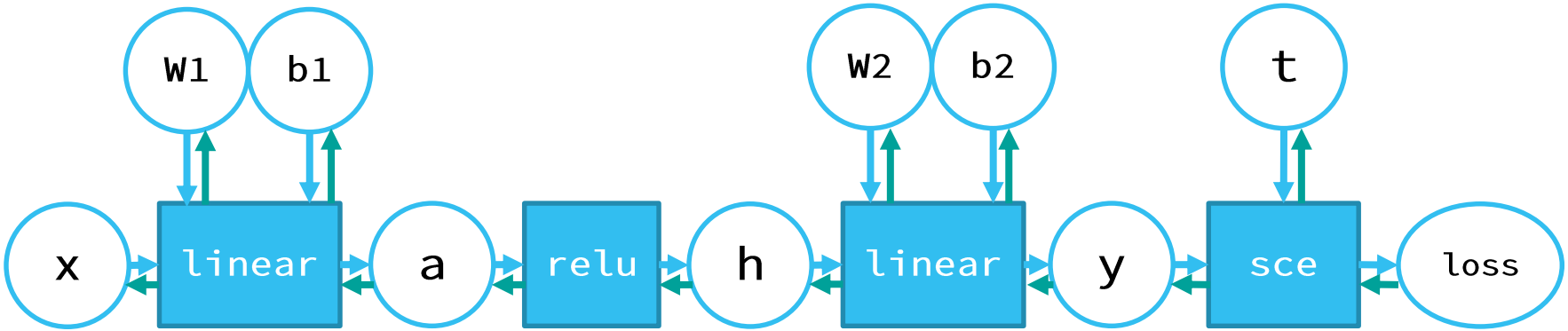
- 計算グラフを作る
- データをグラフに流す
- グラフを逆方向にたどって、勾配を計算する

誤差逆伝播の実装方法: Define by Run

- 順伝播を計算する Python コードを実行
  - 実行中に計算の過程をグラフに記録
- グラフを逆順にたどって、勾配を計算

```
a = F.linear(x, W1, b1)
h = F.relu(a)
y = F.linear(h, W2, b2)
loss = F.softmax_cross_entropy(y, t)

loss.backward()
```



Define by Run の利点

順伝播が Python プログラムとして自然に書ける

- デバッグが容易
  - print デバッグ、デバッガ (pdb) の使用、etc.
- Python 制御構文がそのまま使える
  - 入力に応じて計算を変える、というのが Python プログラムとして自然に書ける

Define by Run の実行特性

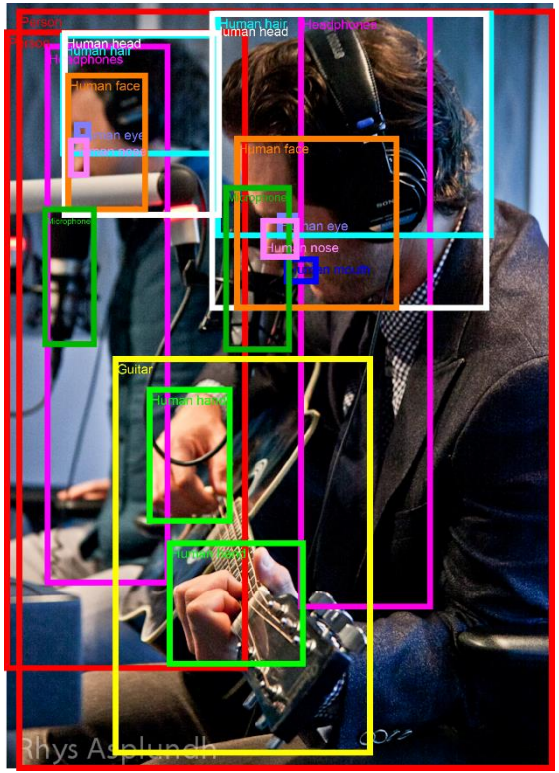
計算グラフが毎 iteration で構築される
→ CPU オーバーヘッドを考慮する必要
→ グラフ最適化はしづらい
→ アドホックな高速化はしやすい

Chainer はスケールする

- Python は遅いが、GPU の計算コストが十分高ければオーバーヘッドは隠蔽できる
- ChainerMN を使うことで複数 GPU にスケールできる

Google AI Open Images Challenge (Object Detection Track)

- 500 クラスのオブジェクト
- 1.7M 枚の訓練画像、
  12M オブジェクト (bounding box)
- Kaggle 上で 7-8 月にかけて開催



Mark Paul Gosselaar plays the guitar by *Rhys A* (https://www.flickr.com/photos/rhysasplundh/5738556102). *CC BY 2.0. Annotated image borrowed from https://www.kaggle.com/c/google-ai-open-images-object-detection-track*

Google AI Open Images Challenge
PFDet チーム

- MN-1b: Tesla V100  x 512枚
- 1回の実験で 33 時間、これをくり返し安定して実施
- 83% のスケーリング効率

T. Akiba, T. Kerola, Y. Niitani, T. Ogawa, S. Sano, S. Suzuki. **PFDet: 2nd Place Solution to Open Images Challenge 2018 Object Detection Track** https://arxiv.org/abs/1809.00778

Preferred Networks

2. Chainer での深層学習の高速化

まず高速化の前に……

- 究極的に、高い精度のモデルを素早く手に入れるのが目標であることを忘れない
- 学習プログラムの完了が速くなっても、得られるモデルの予測性能が落ちていたら意味がない
- （という claim をつけた上で、今日のトークでは予測性能の話はほとんどしません）

Chainer におけるパフォーマンスの勘所

- GPU の計算が効率的かどうか
- GPU カーネルがちゃんと詰まっているかどうか (CPU ボトルネック)

ResNet 50

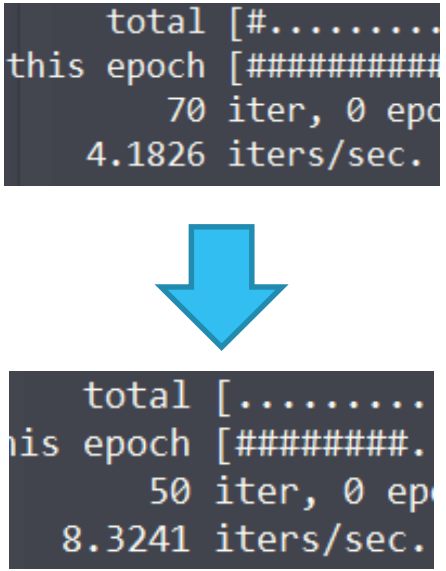
- Tesla V100 (16GB RAM)
- ImageNet
- 前処理：random crop + flip
- バッチサイズ 32
- SerialIterator, StandardUpdater

```
     total [#.........
this epoch [##########
        70 iter, 0 epo
   4.1826 iters/sec.
```


Preferred Networks

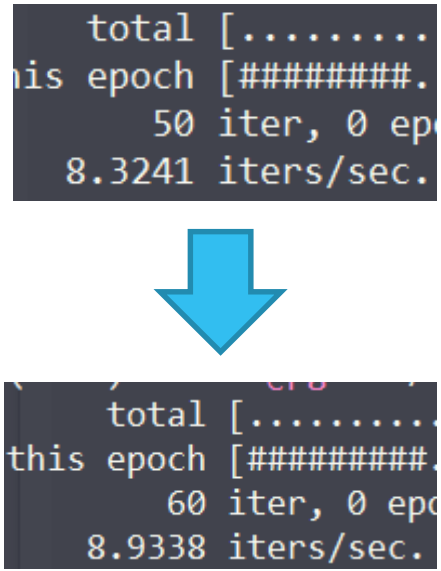
GPU 計算の高速化

- cuDNN を使う
  - `pip install cupy-cuda92` で cuDNN 同梱パッケージが入る (92 の部分は使用している CUDA toolkit のバージョンに合わせる)



GPU 計算の高速化

- cuDNN autotune 機能を使う
  - `chainer.config.autotune = True`
  - 各 convolution の初回実行時に最適なカーネルを探す（初回のみオーバーヘッドがある）



GPU 計算の高速化

- FP16 で計算する
  - メモリ使用量・速度ともに改善する
  - TensorCore が使えればさらに高速に
- **計算精度が落ちるので、最終的なモデルの予測性能が保たれることを確認する**

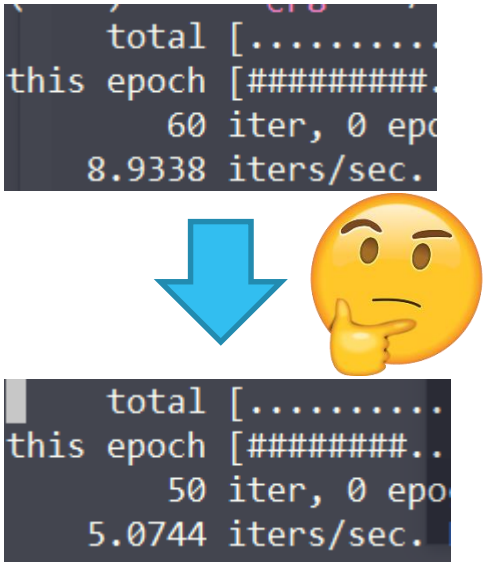
GPU 計算の高速化

FP16 で精度を上げる手段

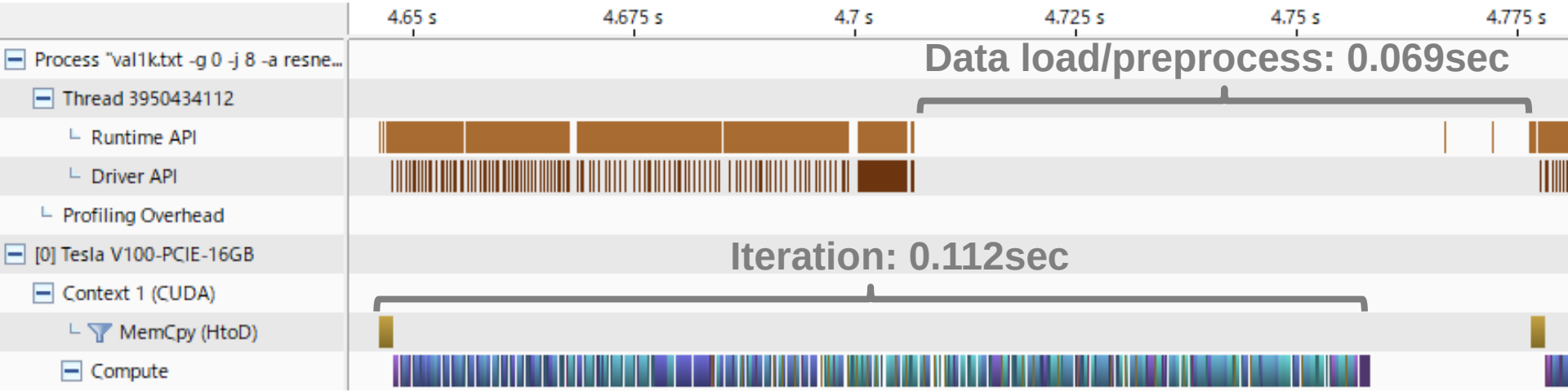
- TensorCore を使う（gemm 内の accumulation が fp32）
- FP32 update
  - 勾配をパラメータの fp32 コピーに足し込む
  - Optimizer の .use_fp32_update() を呼ぶ
- Loss scaling
  - Backprop 時に定数倍されたエラーを扱うことで underflow を防ぐ
  - StandardUpdater 等の loss_scale= や Optimizer の set_loss_scale() で設定可能

GPU 計算の高速化

- FP16 モードを使う（**beta 版機能**）
  - `$ CHAINER_DTYPE=float16`
  - or `chainer.global_config.dtype = np.dtype('float16')`
  - デフォルトで float16 が使われる
  - `cuda.set_max_workspace_size()` と組合せると TensorCore も有効に

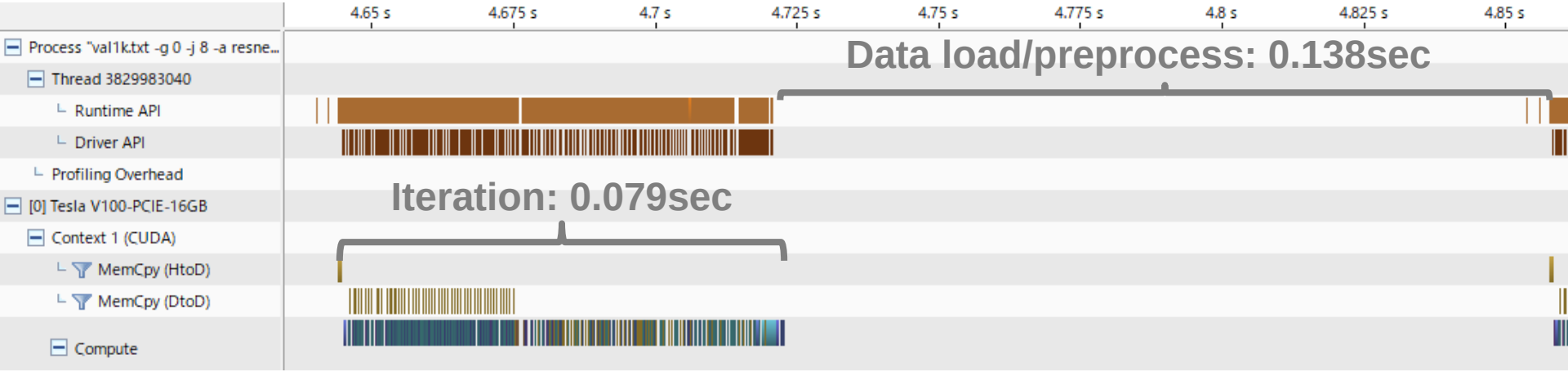


Visual Profiler で見てみる (fp32 mode)



データの読み込み・前処理が間に合っていない

Visual Profiler で見てみる (fp16 mode)

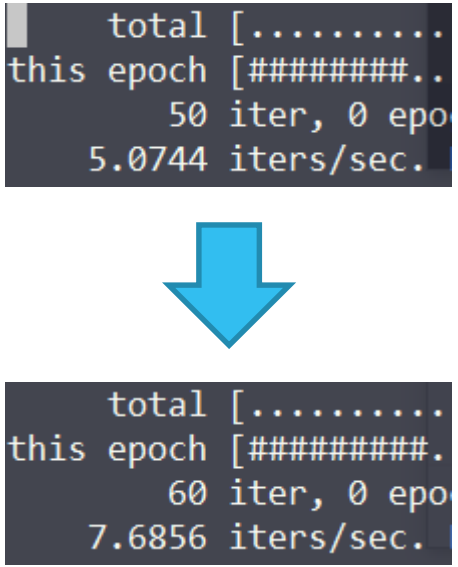


GPU 側が速くなった分カーネル発行の余裕がなくなり、ロード・前処理が隠蔽できなくなっている。ロード・前処理自体にもさらに時間がかかるように。

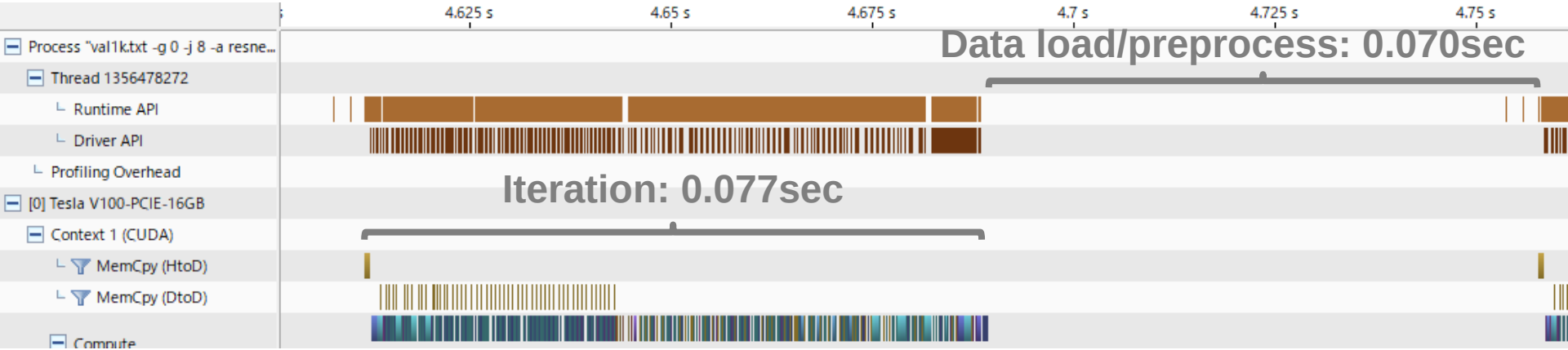
Preferred Networks

データ前処理の高速化

- NumPy の float16 計算は遅い
- 前処理までは float32 でやることにする



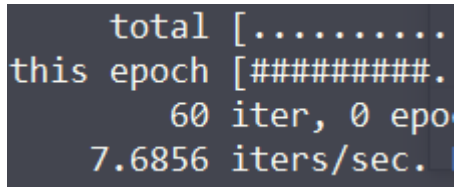
前処理 fp32 化

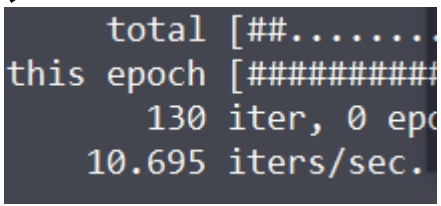


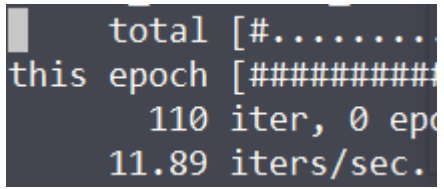
カーネル発行の余裕がなくなった分、まだデータ前処理のオーバーヘッドが露出している

データ前処理の高速化

- 前処理を並列化する
  - MultiprocessIterator を使う
  - データ間の並列化に加え、データの先読みも行う

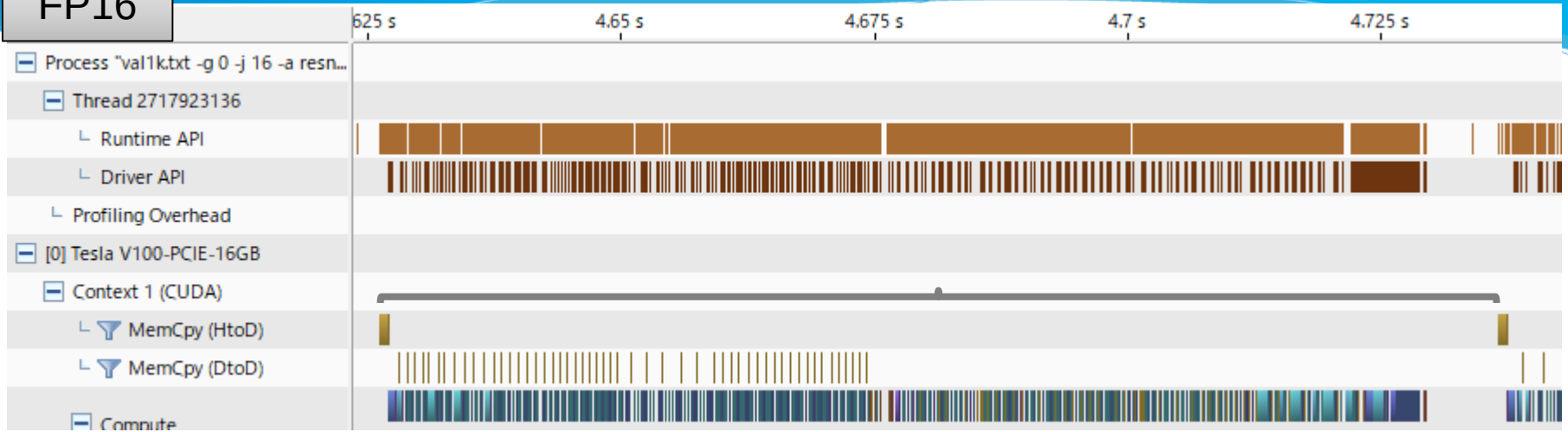




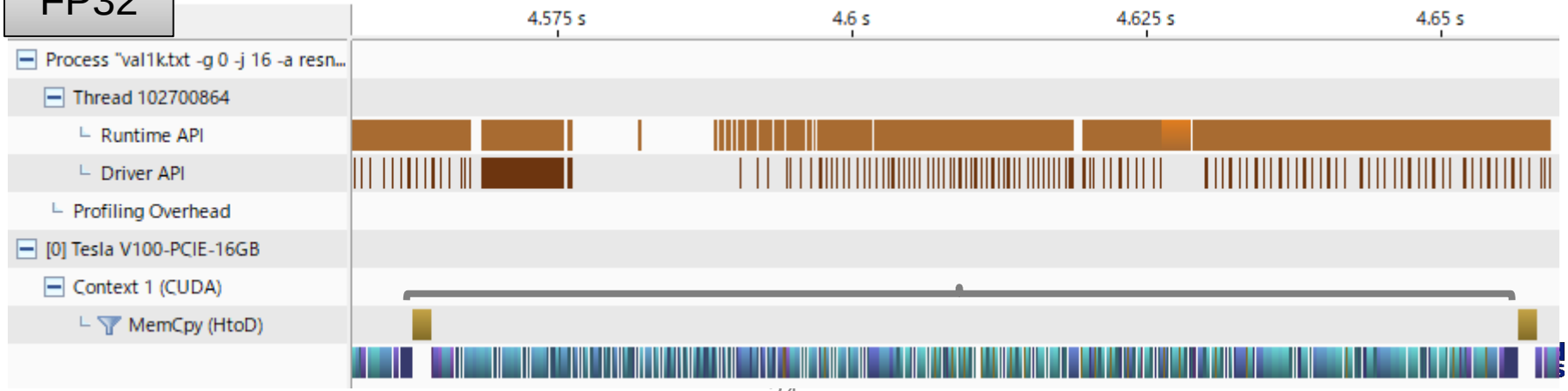


参考：FP32

FP16

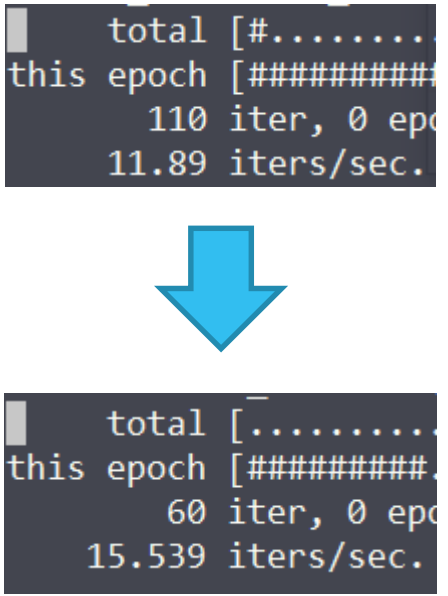

625 s
4.65 s
4.675 s
4.7 s
4.725 s
Process "val1k.txt -g 0 -j 16 -a resn...
Thread 2717923136
Runtime API
Driver API
Profiling Overhead
[0] Tesla V100-PCIE-16GB
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoD)
Compute

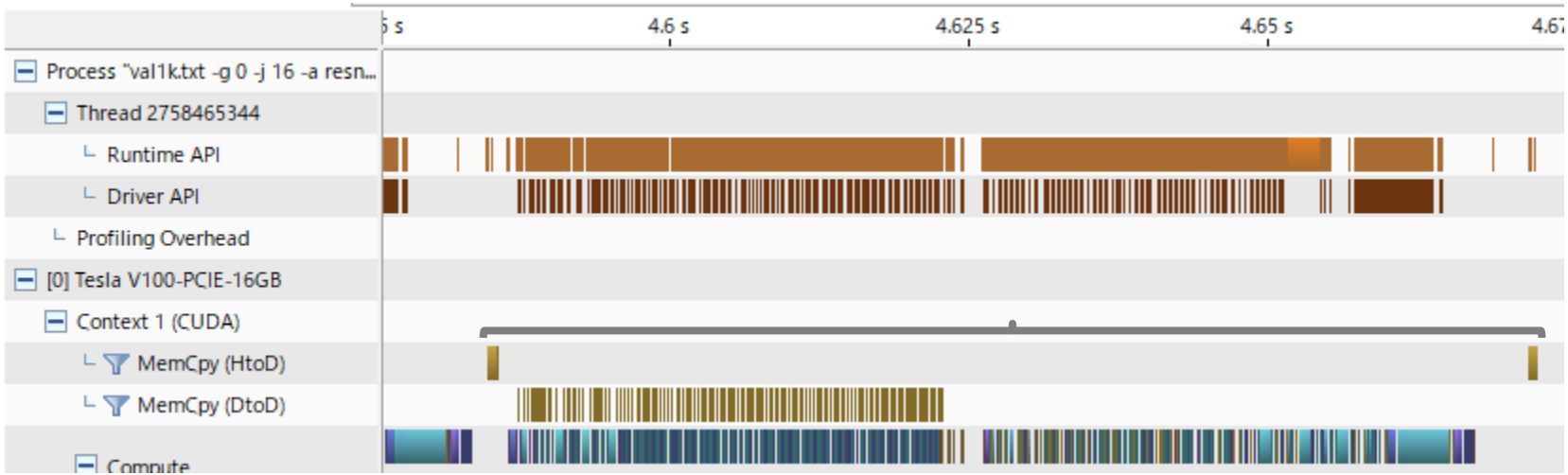
FP32


4.575 s
4.6 s
4.625 s
4.65 s
Process "val1k.txt -g 0 -j 16 -a resn...
Thread 102700864
Runtime API
Driver API
Profiling Overhead
[0] Tesla V100-PCIE-16GB
Context 1 (CUDA)
MemCpy (HtoD)

forward/backward の高速化 **(beta 版機能)**

- 静的グラフ最適化
  （今月中にリリース予定）
  - 計算グラフのうち静的（iteration ごとに不変）な部分をキャッシュし、Python オーバーヘッドを減らす
  - モデルの forward メソッドに @static_graph とつけるだけ

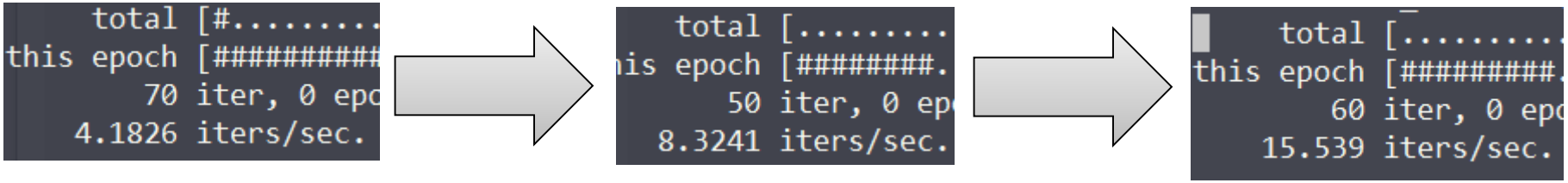




特に backward 側に余裕ができた

cuDNN の使用

様々なテクニック



Chainer の機能を組合せることでおよそ3.7倍の高速化
修正は 8 行

```
chainer.global_config.autotune = True
chainer.global_config.dtype = np.dtype('float16')
chainer.cuda.set_max_workspace_size(512*1024*1024)

train_iter = chainer.iterators.MultiprocessIterator(
        train, args.batchsize, n_processes=16)

@static_graph
# あとは前処理コードで fp32 を強制するための修正
```


Preferred Networks

おまけ：バッチサイズを増やす

- メモリに余裕がある場合、バッチサイズを増やすことで GPU を相対的に重くできる
  - 学習を速く進めるにはテクニックが必要（学習率を同じ比率で上げるなど）
  - FP16 の方がデータあたりのメモリ消費が少ないのでより大きなバッチサイズを使える

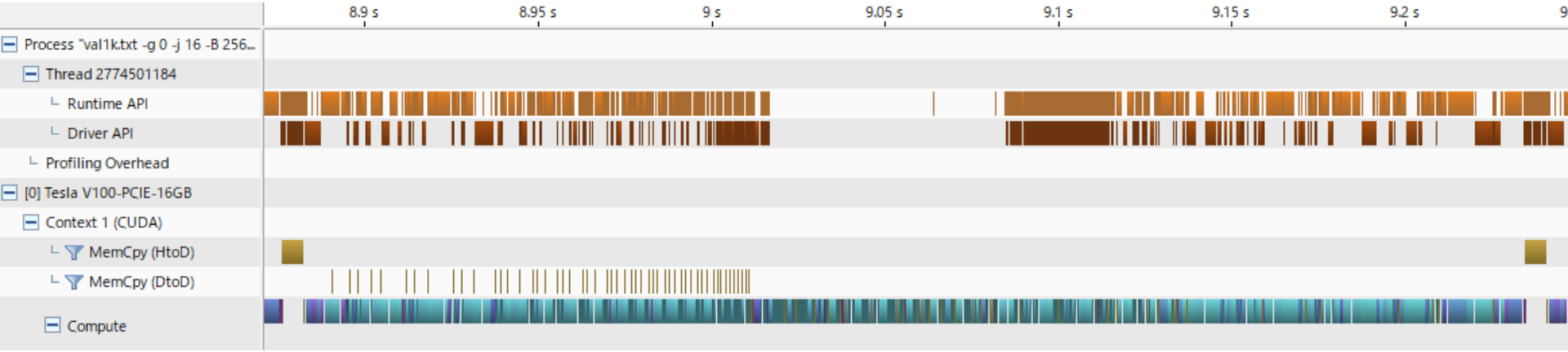
FP32 (batchsize x4)

```
     total [####......
this epoch [##########
        70 iter, 0 epo
     2.908 iters/sec.
```

FP16 **(batchsize x8)**

```
     total [########..
this epoch [##########
        70 iter, 1 epo
    2.7979 iters/sec.
```

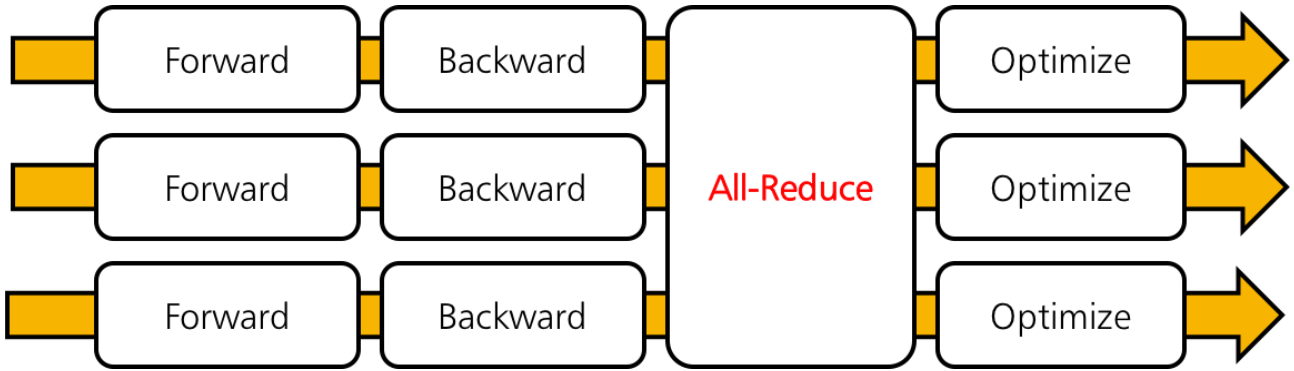
FP16, Large batch (256)



カーネルがちゃんと詰まっている

さらなる高速化

ChainerMN による multi GPU 学習



(次バージョンで Chainer 本体に統合予定)

Chainer 本体のパフォーマンスに対する取り組み

- CuPy の CPU オーバーヘッド削減
  - Kernel launch の高速化
- Chainer のメモリオーバーヘッド削減
  - メモリを削減することで、より大きなモデル・ミニバッチが使えるようになり、GPU 負荷を上げる
- Chainer の CPU オーバーヘッド削減
- （今後）”Static subgraph” の範囲を広げ、グラフ最適化などを盛り込んでいく

Preferred Networks

まとめ

- cuDNN の様々な高速化機能を Chainer から使う
- FP16 + TensorCore を使う
- 思い通りに速くならなかったら Visual Profiler
- CPU ボトルネックを解消するには
  - NumPy での fp16 計算を避ける
  - iterator を並列化する
  - Static subgraph optimization を有効にする
- ChainerMN でさらにスケール
- Chainer/CuPy の高速化にも引き続き取り組んでいきます

Preferred Networks